Homework 4: Naive Bayes

Due October 5th

1 Working with the GoodReads dataset

In this assignment, you will build a Naive Bayes classifier for book genres. You should put your code for this assignment in `naive_bayes.py`. Submit it on Gradescope along with a PDF of your write-up.

Our dataset comes from the GoodReads book review website; it was scraped, cleaned and sorted into genres in 2017 by researchers at UCSD.

- Romance
- Poetry
- Young Adult
- Children
- Mystery, Thriller, and Crime
- Fantasy and Paranormal
- History and Biography
- Comics and Graphic Novels

I have provided two subsets of the original data for testing and training purposes. Each is roughly 1/20th of the original data and has been filtered by country (US) and language (English).

Training: `goodreads_US_17sample.json`

Testing: `goodreads_US_19sample.json`

1.1 Loading and Preprocessing the Data (4 pts)

I have given you a function called `load_data` that loads in data and processes it into a list of dictionaries, one for each book.

As it common in NLP, these JSON files are not, in fact, proper JSON: they contain one JSON object per line, rather than one per file. Thus, we have to iterate through the lines in the file and call `json.loads` on each line to process it into a Python dictionary.

We will use just four of the original eight genres:

- Romance
- Young Adult
- Mystery, Thriller, and Crime
- Fantasy and Paranormal

I have given you a function called **`sort_and_filter_books`**. It takes a list of book dictionaries and a list of genres, and filters out books that do not belong to a genre in the genre list. It returns a dictionary where the keys are genres and the values are lists of book dictionaries in that genre.

Use these two functions to load the dataset and filter it to the four genres above.

2 Training Your Classifier

To train our Naive Bayes classifier, we need to calculate two sets of probabilities: the *priors*, reflecting document frequency, and the *likelihoods*, reflecting word frequencies in each class.

We will work with **log probabilities** to avoid numerical stability issues.

2.1 Computing Priors (5pts)

The priors capture the probability of a document belonging to a given class. In our case, this is the probability of a book belonging to a genre, all else being equal. Not all genres are equally well-represented in our dataset. If we know nothing else about a book, we can make an educated guess about its genre simply by guessing the most likely genre.

Write a function called **`compute_priors`**. It should take a dictionary of genre-sorted book dictionaries. It should return a dictionary of log probabilities: the keys should be genres, and the values should be the probability of a book belonging to that genre.

2.2 Counting Word Frequencies (4pts)

Calculating likelihoods is more involved than calculating priors. Start by writing a function called **`count_words_in_books`**. This function should take a list of book dictionaries and a *field*, a key within the book dictionary that we will base our features upon. We will start by using the field "title_without_series".

Your function should iterate through the books, call the **`tokenize`** function on the field for each book, and count how many times each token occurs across all books. It should return a Counter.

You can check that your function is working correctly by searching for an uncommon word in the JSON file, and verifying that the count matches what your program calculates for a particular genre. For instance, "Unicorns" appears only twice in titles in **`goodreads_US_17sample.json`**, both times in the Young Adult category.

2.3 Genre Word Counts (4pts)

Once you are sure your **`count_words_in_books`** function is working, write a function called **`make_genre_counts`**. This wrapper function should take a genre-sorted book dictionary and field. It should call **`count_words_in_books`** to calculate the word counts for each genre, and return a dictionary where the keys are genres and the values are word count dictionaries.

2.4 Building the Vocabulary

It is also convenient to have a list of all words in the training dataset vocabulary. Write a function called **`make_vocab`**. It should take in the genre counts returned by your **`make_genre_counts`**, and it should return a list of all words in the training dataset.

2.5 Computing Likelihoods (6pts)

You now have all the pieces assembled to calculate likelihoods.

Write a function called **`compute_likelihoods`**. It should take two arguments: your vocabulary and your genre count dictionary. It should return a dictionary of log-likelihoods, keyed by genre, with dictionaries of log-likelihoods as values.

You should follow the pseudo-code given in the Jurafsky & Martin chapter to calculate log-likelihoods. This includes using add-1 smoothing.

Hint: it is helpful to calculate the total number of words in each genre once at the top of the function, rather than repeatedly. I have given you a function called **`make_genre_sums`** to help with this. It takes a genre count dictionary and a vocabulary and returns a dictionary where the keys are genre names and the values are the total number of words in that genre.

Check in

The log-likelihoods for "Autumn" should be:

- Romance: -9.31
- Young Adult: -10.33
- Mystery, Thriller, Crime: -9.74
- Fantasy and Paranormal: -9.02

Hint: If you are struggling to think about log-probabilities, remember that you can always convert them back to probabilities when you print by calling math.exp().

2.6 Questions (5 pts)

Congrats! You've now trained your Naive Bayes classifier. Take a moment to explore your data before you move on:

1. What are the top 5 highest-probability features for each genre?

2.7 Packaging Your Classifier

Write a function called **`train`** to package up the steps involved in training your classifier. This function should take 3 parameters: the name of training data file, the list of genres to use, and the field to use.

3 Testing Your Classifier

In this section, we will test out the classifier that you made.

3.1 Processing Training Data

Write a function called **`test`**. This function should take 6 parameters: the name of the test data file; the learned priors; the learned likelihoods; a list of genres; the learned vocabulary; and a field.

We will continue adding to this function for the rest of this section. Start by adding code to load, sort, and filter the test data. You should be able to reuse code from previous sections.

3.2 Computing Genre Probabilities (5pts)

Write a function called **`compute_class_scores`**. This function should take a single book dictionary as an argument, along with the information you need to predict its genre: priors, likelihoods, vocabulary, and field.

For each genre, it should calculate the likelihood of the book belonging to that genre. It should store these probabilities in a dictionary indexed by genre.

To calculate the log-probability that a book belongs to a genre, you should follow the pseudo-code given in the Jurafsky & Martin chapter.

3.3 Classifying Books (7pts)

Write a function called **`classify_book`**. This function should take a book and the data need to classify it, and should call **`compute_class_scores`** to get the log-probabilities that it belongs to each genre. It should extend the score dictionary with some meta-data that will be useful for analysis: the book's title, its field value, and its actual genre. Store the actual genre with key 'gold' and the field with key 'field'.

Now write a wrapper function called **`classify_all_books`**. This should simply iterate through the genre-organized dictionary of books, call **`classify_book`** on each book, and compile the results into a list to return.

Check in

For the second book in the test set, "The Mountain Between Us", the log-probabilities of belonging to each genre should be as follows:

- Romance: -28.076
- Young Adult: -30.717
- Mystery, Thriller, Crime: -32.246
- Fantasy and Paranormal: -30.144

4 Evaluation

4.1 Calculating Metrics (8pts)

To understand how well our classifier is doing, we will need to implement some metrics. Write functions to calculate two evaluation metrics, precision and recall.

Your functions should be called **`by_class_precision`** and **`by_class_recall`**. They should take a list of results and a genre, and return the precision or recall score for predictions within that genre.

4.2 Questions (5 pts)

- Which genre does the model do best on?
- Do you notice any relationship between precision/recall and the genre priors?

4.3 Overall Performance (3pts)

We now have a way to calculate how well the model does on particular genres. But what if we want to compare our classifier to another model?

Write a function called **`macro_average_metric`**. Your function should calculate the *macro-average* for a given metric. (See Jurafsky & Martin for a discussion of micro- and macro-averages.)

Your function should take three parameters: a list of results, a list of genres, and a metric (the name of a function that calculates a metric). It should return the macro-average of that metric over all genres.

4.4 Titles versus Descriptions (5pts)

So far, we have been basing our features on the book titles. We would like to retrain and retest the model using the book's description rather than its title. But, there is an issue: the vocabulary is much larger for descriptions than titles.

To make sure that you can still run your program fairly quickly, we will cap the vocabulary at the 20,000 most common words.

Implement a frequency-based vocabulary cap in your make_vocab function. Make sure you also adjust the per-genre word counts in your compute_likelihoods function.

Now, retrain and retest the model using book descriptions ("description") rather than title.

(If you have written your code in a modular way, this should be trivial. But if you haven't, you may need to do some re-factoring first.)

This is likely to take some time, even though we capped the vocabulary size (around 15 minutes on my laptop). You can test with a tiny vocabulary and re-run with 20,000 words when you are confident everything is working.

4.5 Questions (5 pts)

- How do the models trained on titles and trained on descriptions compare?
- Calculate the *balanced F-measure* for both models. Which is better?

4.6 Most Genre-typical Books (5pts)

Our classifier should be able to tell us what kinds of books are most likely to occur in each genre. Write a function called **`display_top_genre_scores`**. This function should take three parameters: the list of results, the list of genres, and n, the number of top scores to display.

For each genre, it should sort the books by their scores and print out the n most likely books to belong to that genre.

4.7 Questions (2 pts)

- What do you observe about the books? There's an issue with our approach. Can you spot it?

4.8 Most Genre-typical Books, Revisited (3 pts)

Rewrite **`display_top_genre_scores`** to try to fix the issue you identified above. You will need to pass an additional argument to the function.

4.9 Questions (10 pts)

- List the 5 books that your revised function prints for each genre. Discuss any interesting patterns that you notice.
- Do you think there are still problems with our approach to finding the most genre-typical books?
- So far we have tested the classifier only on genres we observed during training. Technically, we could run our classifier on a test set that includes a genre not observed in training. Do you think this would work? If so, describe how you would adapt your program to do this.

5 Intellectual Curiosity (10pts)

As usual, you will receive 90 points for implementing everything described above. To increase your score further, you can extend your investigation in some way.

If you choose to do this, please briefly describe what you've done in your report so that we can make sure to look for it!