Language Identification with Naive Bayes

In this homework assignment, you will use text classification algorithms to identify the language of sentence. For language identification, we will use the Naive Bayes machine learning algorithm with character n-gram features.

Learning Goals

Once you complete this assignment, you should:

- Understand how to classify text using the Naive Bayes algorithm
- Be able to use multiple methods to evaluate your classifier's performance, and be able to compare and contrast the effectiveness of those methods

This assignment is connected to the following overall learning goals of the course:

- Be familiar with NLP methods in three key areas: text classification, text generation, and language understanding

Implementation Details

You will implement a naive bayes model that uses character n-gram probabilities to predict the language that a sentence is written in. You shouldn't edit `test.py` or `test_mini.py` - in other words, you must implement the specific functions that are called in those files.

In your implementation, you should do the following:

1. Your calculations should all be performed in log space to avoid underflow. See the `normalize` function (it will compute log probabilities by default). Remember that *adding* log probabilities is equivalent to multiplying probabilities.
2. You should use add-one smoothing. You'll do this after counting n-grams, before computing probabilities. First, create a set containing all of the character bigrams in the vocabulary. Then, loop through your likelihood counts to add one to each value. Finally, normalize! *How about character bigrams that aren't in the vocabulary at all, for any language?* Their presence shouldn't affect your prediction, so you can safely ignore them entirely.
3. You'll notice that `ngram_size` is an argument to the constructor for your model. Your model should work for any n-gram size. This should be easy to implement using the provided `get_char_ngrams` function.

Resources

Starter Code

model.py This file includes three methods that **you must implement**, `fit`, `predict`, and `predict_one_log_proba`.

`fit` `fit` should train your model on the input sentence/language pairs. These are parallel lists where the language at index `i` in `train_languages` corresponds to the sentence at index `i` in `train_sentences`.

In particular, you should set `self._prior` and `self._likelihood` to the appopriate values (remember to use log probabilities and laplace smoothing). I recommend using dictionaries/dictionaries of dictionaries here for simplicity, but you can use any data structure you want.

`predict` `predict` should return a list of predicted languages for each sentence in `test_sentences`. It should use `self._prior` and `self._likelihood`. Remember to add log probabilities and ignore unknown words!

`predict_one_log_proba` This method is really just a helper method for `predict`, but it might help you to debug and it will help me with automatic grading. This method should return a dictionary corresponding to the log probability of each language for a single sentence.

scoring.py This file includes two functions that **you must implement**, `accuracy_score` and `confusion_matrix`. Here are some sample test cases for each function:

`accuracy_score`

```
y_true = ["spa", "eng", "spa"]
y_pred = ["eng", "eng", "spa"]
print(accuracy_score(y_true, y_pred))
# prints 0.6666666666666666
```

`confusion_matrix` Note that `confusion_matrix` and `print_confusion_matrix` use the `labels` list argument to determine the column labels. This means that there may be additional rows in the matrix that aren't in the true/predicted labels.

```
y_true = ["spa", "eng", "spa"]
y_pred = ["eng", "eng", "spa"]
print_confusion_matrix(confusion_matrix(
    y_true, y_pred,
    ["spa", "eng", "fra"]), ["spa", "eng", "fra"])
# prints this table
# ------------------------------------
# |           |      Actual Label     |
# |           |-----------------------|
# |           |     | spa | eng | fra |
# ------------|-----|-----|-----|-----|
# | Predicted | spa |  1  |  0  |  0  |
# |   Label   |-----|-----|-----|-----|
# |           | eng |  1  |  1  |  0  |
# |           |-----|-----|-----|-----|
# |           | fra |  0  |  0  |  0  |
# ------------------------------------
```

util.py Functions in `util.py` are provided for loading data, creating character n-grams, normalizing, performing argmax, and printing the confusion matrix in an easy-to-read way. You don't *have to* use these functions, but I'd recommend it (it will make your life easier).

Examples of how to use the functions `get_char_ngrams`, `normalize` and `argmax` are provided below:

`get_char_ngrams` In this assignment, we will work with **character ngrams**, which are ngrams made up of individual characters instead of words.

The inputs to `get_char_ngrams` are a string `string` and an integer `n`, which represents the number of characters that should be in each ngram. The function returns a list of strings.

Here are a few examples:

```
string = "blanco"
print(get_char_ngrams(string, 2)) # --> prints ['bl', 'la', 'an', 'nc', 'co']
print(get_char_ngrams(string, 3)) # --> prints ['bla', 'lan', 'anc', 'nco']
```

`normalize` This function takes a dictionary of counts and normalizes them by dividing each value by the total counts. It returns a dictionary of the normalized counts, which are probabilities. It can optionally return log probabilities. **It does not modify the original dictionary.**

Here's an example:

```
counts = {"spa": 50, "eng": 150}
probs = normalize(counts, log_prob=False)
print(probs) # --> prints {'spa': 0.25, 'eng': 0.75}
```

`argmax` The argmax function takes a dictionary with numbers as values, and returns the key with the highest value.

Here's an example:

```
probs = {"spa": 0.7, "eng": 0.3}
highest_key = argmax(probs)
print(highest_key) # --> prints "spa"
```

Data

The provided `train.tsv` and `dev.tsv` are subsets of the Tatoeba sentences dataset.

Allowed External Resources

You may use any built-in python modules (in particular, you may think that Counter and defaultdict from the `collections` module are useful). You may not use external libraries.

While you may not search for code that solves this problem, you may search for ideas to improve your preprocessing or smoothing.

Deliverables

Code

Suggestions for Developing your Model In this assignment, you will be working with a (fairly) large dataset. While you are still working on your implementation, you probably should not use the full dataset, as there are more than 4 million rows in the training data and almost 1 million rows in the test data. Naive Bayes is pretty fast, but running my code with character bigrams on my laptop with the full dataset still takes over a minute. You probably don't want to wait a whole minute to learn that there is a bug in your code.

Therefore, I've built in some methods to get around this. First, you can load a partial dataset using the `--avg_samples_per_language` command line argument (e.g., run your code as `python test.py train.tsv dev.tsv --avg_samples_per_language 50`). This will load approximately 50 random samples per language, rather than loading the full dataset (although the number of samples per language will still be proportional to the presence of that language in the full dataset). Your code should run in a few seconds with this flag - if it's taking over a minute, either your laptop is really slow or you need to make some major modifications to your approach.[1]

Second, it can be hard to debug your code if you are just looking at the predictions and seeing that the accuracy doesn't look as good as you think it should, but you don't know why. I've included a file that should be able to help with this: `test_mini.py`. If you run `python test_mini.py`, you should get the probabilities of the word "able" being in English vs. Spanish, with the same training set that we looked at in class. This is the result you should get for that example:

```
{'eng': 0.0006666666666666675, 'spa': 0.00039506172839506165}
```

[1] Working with a dataset this large is a great way to *really* appreciate big-O. For instance, if you train your model once for every sentence in the test set, your code might take hours to run.

Your tasks

You must complete the following:

- In `model.py`:
  - `NBLangIDModel.fit`
  - `NBLangIDModel.predict` (you'll probably want to call the method below!)
  - `NBLangIDModel.predict_one_log_proba`
- In `scoring.py`:
  - `accuracy_score`
  - `confusion_matrix`

Other parts of this assignment provide details on how you need to implement these functions; there is also some information in the comments!

Possible Extensions (OPTIONAL)

This is our first assignment that includes the opportunity to **compete** for the best extensions to your model on the Gradescope leaderboard. Here are some ideas of how you might extend what you've done for the leaderboard. These extensions should be turned on/off using the `extension` argument in your `NBLangIDModel` constructor. When the `extension` argument is `True`, you may also override `self.ngram_size` - basically, I am going to do the following to test for the leaderboard, with a new test set:

```
model = NBLangIDModel(extension=True)
model.fit(train_sentences, train_labels)
predictions = model.predict(test_sentences)

# we'll use accuracy on the leaderboard for simplicity
score = accuracy_score(test_labels, predictions)
```

You can do things that aren't on this list too, but make sure that you are still in essence implementing a Naive Bayes model. If you've taken deep learning, for instance, you cannot use deep learning for this assignment.

Preprocessing your text If you followed the instructions word-for-word, you won't have done much text preprocessing. Can you add preprocessing steps to improve your predictions, such as converting the text to lowercase?

Multiple n-gram estimates Try computing your probability by using multiple different n-gram sizes.

Improved smoothing Look at Chapter 3 of slp to get some additional ideas about smoothing.

Report

In addition to your code, you must fill out the report in `report.md` (from the starter code).