Goals. The main purpose of this assignment is for you to get experience in implementing text classification systems, and also to serve as a relatively gentle introduction to PyTorch. It's also an introduction to the kind of coding and problem sets we'll be working with in the course.

Resources

Dependencies. Most assignments in this course will involve programming in Python. **Please use Python 3.9 or higher.** I recommend installing conda, which will help you install Python packages in a modular way for each homework assignment.

This assignment has an autograder for the coding portion. The autograder has the following packages installed: `numpy, nltk, spacy, torch`. Do not use any packages not in this list, or else the autograder will not work.

Data. For this assignment, we'll be doing text classification. The data can be found at this Google Drive link.[1] We'll be training a classifier that takes in news articles and classifies them into one of four categories:

- World news
- Sports
- Business
- Tech/science

For humans, telling the difference is usually very easy. As we'll see, word-based classifiers find this a bit harder. To make things even harder, *the dataset is also highly imbalanced*.[2] We have 1000 examples for world news, but only 50 for sports! It can therefore be easy to learn bad generalizations, and our model may not be able to effectively handle the less-frequent classes of text. This means that we'll need to get creative to build an effective classifier, and even to design a good evaluation metric. In this assignment, the labels are as follows: "0" is world news, "1" is sports, "2" is business, and "3" is tech/science.

The data that you have access to has been split into a training set, development set, and test set. You have access to the labels for the training and development sets, but not the test set.

Code structure. You will see several Python scripts in the provided repository. The main script is `lang_classifier.py`, which you will be calling to load the data and run your classifiers. It's okay to change this file for testing purposes, but **the version of the script we'll use to grade your submission will not differ from the provided version.**

Data handling is done in `dataset.py`. This script iterates over the data file, and loads each example as an `Example` object. Each `Example` contains a string (the "input") and a label.

The `utils.py` script contains some evaluation functions. You will be implementing macro-F1 in this script.

The main file you'll be modifying throughout the homework is `models.py`. It defines some trivial baselines that we'll be using as a sanity check below. It also contains various featurization classes and functions like `train_logistic_regression` that you'll be implementing.

[1] https://tinyurl.com/yeyrvmru

[2] Imbalanced data is very common in real-world scenarios. This is especially true when data is limited and we need to make use of every example we can get.

Assignment

Download the code/data for this homework, and unzip the file. Change into the `hw0/code/` directory. As a sanity check, run this script after you download all of the necessary Python dependencies:

```
python lang_classifier.py --model TECH
```

This loads the data and labels and initializes a classifier `TechClassifier`. This classifier always predicts the label 3 (tech/science). It evaluates on the training and development sets. If everything is working properly, you should see these accuracies:

- Train accuracy: 0.1212
- Dev accuracy: 0.1176

Note that accuracies here lie in the range [0, 1]. As you can see, this isn't a great classifier!

Task 1: Bag-of-words Classification (6 points)

Your first task is to implement a bag-of-words unigram classifier. For this, you will need to modify or implement the following functions in the `BoWFeaturizer` class in `models.py`: `build_vocab` and `get_feature_vector`. First, you will need to count the number of occurrences of each token and filter down the vocabulary to the most frequent tokens.

Then, you will need to map from string inputs to feature vectors. A simple way to do this is by defining a vector equal to the size of your vocabulary, where each vocabulary item has a dedicated index. There is no single best way to do this. We probably don't want to just take all words as-is, because then capitalized and non-capitalized versions of the same word will be considered as completely different features (which would shrink our effective vocabulary). For example, you might consider throwing out all low-frequency words and replacing them with a catch-all low-frequency token. You might also consider lowercasing all of the words before defining the vocabulary. You could also choose to make each vector index binary, indicating whether or not the word is present in the sentence, or you could instead make each index an integer (e.g., to store the number of times the word appeared in the sentence).

Limit your vocabulary size to no more than 5,000. Feel free to filter down the vocabulary further if you wish (so long as performance is not significantly harmed).

Q1 (3 points; AUTOGRADED). Implement the `BoWFeaturizer` class. The vector returned by `get_feature_vector` should contain the counts of all tokens in the vocabulary in a given text. Recall that the vocabulary size must be $\leq$ 5,000; thus, the feature vector should also have length no more than 5,000.

For now, we will use PyTorch's built-in logistic regression function and a pre-provided gradient descent function, which have been provided for you in the `BlackBoxClassifier` class of `models.py`. Once you've implemented the above, run the following command to train and evaluate the classifier:

```
python lang_classifier.py --model BOW
```

To get full credit, you must obtain **a development accuracy of at least 0.63** using the pre-provided hyperparameters (num epochs = 10, vocab_size = 5,000), and **training and evaluation time should be less than 1 minute** on the Gradescope backend. (Our version trained in less than 5 seconds on a CPU on my laptop, and less than 20 seconds on Gradescope.) Please write the training *and* development accuracy you obtained in your written report.

Q2 (1 point). Compare the training and development accuracies of your model. In 1–3 sentences, describe why they are different.

Q3 (2 points). In `train_torch_model` in `models.py`, you are given the weight matrix of your trained logistic regression model. This is a matrix of size $(4, V)$, where V is the vocabulary size. Each weight index corresponds to a sort of "importance score" for each token in your vocabulary for predicting a certain class.

For each class (each row of the weight matrix), list the 5 indices corresponding to the highest weights. Also list the tokens from your vocabulary that correspond to each of these indices. What trends do you notice? Do the most important tokens for each class seem reasonable?

Task 2: Logistic Regression (13 points)

Now, you will implement a logistic regression classifier from scratch. Use the same feature vectors you derived for Task 1.

Q4 (10 points; AUTOGRADED). Implement a multinomial logistic regression classifier from scratch. You will need to implement the following for this to work:

- The `forward` function. The skeleton has been implemented in `LogisticRegressionClassifier` in `models.py`; complete this function. (2 points)
- The `softmax` function. You may not use `torch.nn.Softmax` or any similar pre-implemented function, but you are allowed to use `torch.exp` if you wish. (2 points)
- Stochastic gradient descent. The skeleton of the algorithm has been implemented in `train_logistic_regression`; complete this function. (6 points)

Once you've implemented these, train and evaluate your model using the following command:

```
python lang_classifier.py --model LR
```

Report your model's final training and development accuracy on the dataset in the written report. For full credit, **your model must obtain a dev accuracy of least 0.63** and **finish training and evaluation in 1 minute.**

Q5 (3 points). In `lang_classifier.py`, play around with the hyperparameters of the model. Try changing the maximum vocabulary size, learning rate, and number of training epochs. In your written report, provide the best training and development accuracy you were able to get, and the hyperparameters you used to get them.

In 1–2 sentences, comment on what happened when you made the learning rate too high or too low, in terms of both the number of epochs it took to converge to a stable loss (or not), and the final accuracy. In 1–2 additional sentences, describe why you think this happened.

Task 3: Feature Engineering (14 points)

Now, using your logistic regression classifier, you will implement more sophisticated features.

Q6.

(a) (3 points; AUTOGRADED). Implement a bigram feature extractor in `BigramFeaturizer`. Your vocabulary should contain only pairs of words, like `The|cat` or `went|to`. Use whatever vocabulary size you wish, as long as it's reasonable (i.e., less than the number of bigrams in the entire dataset); we recommend starting at 5,000 and tuning from there. Use this command to train and evaluate your model:

```
python lang_classifier.py --model BIGRAM
```

In the written report, provide the training and development accuracy of the best logistic regression classifier you can train using your bigram features.

(b) (2 points). You will probably find that the performance of your bigram model is worse than your original unigram (bag-of-words) classifier. This might be surprising, because a bigram classifier is more complex and expressive than your earlier unigram classifier. In 2–4 sentences, explain how **feature sparsity** leads to **overfitting** in the bigram classifier. (Hint: You may find Chapter 3.5 and Figure 3.1 in the textbook helpful.)

Q7.

(a) (8 points). Implement at least two new features of your choice. You can add these to an existing featurizer or write a new custom featurizer class. These features could be anything, such as the type/token ratio of the example, average word length, sentence length, TF-IDF scores, among others. They should not just be combinations of your existing unigram or bigram features. **You get 3 points for each custom feature implemented, and 1 point for each custom feature description in your written report.** Be sure to include the development accuracy of your custom featurizer in your written report. It is okay if your custom featurizer does not outperform the original models, but it should not be significantly worse! Include your custom featurizer somewhere in `models.py`, and write where we can find your featurizer in your written report.

(b) (1 point). Why do you think the performance of your custom featurizer differs (or doesn't) from the models we've trained so far?

Task 4: A Better Evaluation Metric (5 points)

Accuracy is a decent metric when a dataset is balanced. As you may have noticed, however, there are many more world news examples than sports examples, and many more business examples than tech/science examples. Thus, accuracy is determined more by the classifier's performance on world news and business than by its performance on all classes. We will now implement a better evaluation metric that takes accuracy on each class into account.

Q8.

(a) (3 points). Implement the macro F_1 score in `utils.py`. As a reminder, this involves computing the F_1 score separately for each class, and then taking their macroaverage. Once you've done this, uncomment the evaluation and print lines in `lang_classifier.py`. Re-evaluate your logistic regression model from Q4 using the macro F_1 score, and put the score on the dev set in your write-up.

(b) (2 point). Is your F_1 score significantly different from your accuracy from Q4? Regardless of your answer, in 1–3 sentences, what does this difference or lack thereof tell you about your classifier?

Submission

Upload the following two items to Gradescope:

1. Your responses to the questions above should be uploaded as a PDF to **HW0: Classification [Written]**. Ideally, your responses should be in a numbered list. **If you were unable to get your code working, upload any written responses that you can anyway so that we can give partial credit.**
2. Zip your modified `models.py` and `utils.py` scripts. Upload your .zip file to **HW0: Classification [Code]**.

For the code, the autograder will grade your submission using the following criteria:

1. Execution time: do your models finish training and evaluating in a reasonable amount of time?
2. Development set performance: do your models achieve reasonably good accuracies on the development set?
3. Test set performance: the autograder does not explicitly assign any points based on your model's test set performance, but we may investigate if the test set performance differs by a large degree from the development set performance (e.g., by more than a few percent).

Note that some additional imported libraries might work, but the autograder may not have everything; we therefore recommend sticking to `numpy`, `nltk`, `spacy`, and `torch`. Also note that **the course staff may give you a higher score than what the autograder gives you at their discretion;** if your models achieve low performance, the course staff will assess what you did manually and assign partial credit.

Make sure that these commands all work before you upload:

```
python lang_classifier.py --model TECH

python lang_classifier.py --model BOW

python lang_classifier.py --model LR

python lang_classifier.py --model BIGRAM
```