DD2417 Language Engineering
Assignment 2

Johan Boye
2024-03-25

All programs should be written in Python 3, unless specified otherwise in the problem instructions. Don't use any external libraries (that are not part of the Python 3 distribution) unless otherwise specified.

Mandatory part

1. (Bigram models) We want a program that computes all bigram probabilites from a given (training) corpus, and stores it in a file. For instance, from the file `data/small.txt` we want to produce the contents of the file `small_model_correct.txt`. Note that:

   - The first line contains two numbers, separated by space: The vocabulary size V (= the number of unique tokens, including punctuation), and the size of the corpus N (= the total number of tokens).
   - Then follows V lines, each containing three items: an identifier $(0, 1, \ldots)$, a token, and the number of times that token appears in the corpus.
   - Then follows a number of lines, one for each non-zero bigram probability. Each line contains three numbers: The identifiers of the first and second token of the bigram, respectively, followed by the logarithm of the bigram probability, printed with 15 decimals. The natural logarithm is used (as computed by the `math.log` library method).
   - The final line is "-1" to mark end-of-file.

   The `BigramTrainer.py` program contains a skeleton program for reading a corpus, computing unigram counts and bigram probabilities, and printing the model. **Your task is to extend the code so that the program works correctly** (look for the comments `YOUR CODE HERE` in the program). First go to the folder `LanguageModels` and type:

   ```
   pip install -r requirements.txt
   ```

   Now everything needed for the assignment should be installed. To run the program on test examples use the scripts `run_trainer_small.sh` and `run_trainer_kafka.sh`

   You can use the `-d` option to save the model to file, e.g. :

   ```
   python BigramTrainer.py -f data/kafka.txt -d kafka_model.txt
   ```

   If you are using Windows, printing the model to the terminal will likely lead to character encoding errors.

2. (Text generation) It is now possible to generate words by sampling from the language model. If, for example, the last generated word was `red`, and according to the model `red` can be followed by one of the words `ball` with $p = 0.5$, `toy` with $p = 0.3$, or `car` with $p = 0.2$, then the next generated word should be either `ball`, `toy`, `car` with probabilities 0.5, 0.3 and 0.2, respectively.

   The `Generator.py` program contains a skeleton program for generating words from a language model in the way just described. Extend the code so that the program works correctly (look for the comments `YOUR CODE HERE` in the program). Then generate some words from the various models you have constructed in the preceding problems. (In the rare event where all bigram probabilies from the last generated word are zero, then pick any word at random using a uniform distribution).

3. (Evaluating n-gram models) The `BigramTester.py` program contains a skeleton program for reading a model on the format described in the problem 1, reading a test corpus, and computing the entropy of the test corpus given the model.

   (a) Extend the code so that the program works correctly (look for the comments `YOUR CODE HERE` in the program). The entropy of the test set is computed as the average log-probability:

   $$-\frac{1}{N}\sum_{i=1}^{N}\log P(w_{i-1}w_i)$$

   where N is the number of tokens in the test corpus. To be able to handle missing words and missing bigrams, use linear interpolation:

   $$P(w_{i-1}w_i) = \lambda_1 P(w_i|w_{i-1}) + \lambda_2 P(w_i) + \lambda_3$$

   The values for the constants λ_1, λ_2 and λ_3 are given in the code for the `BigramTester` program. The script `run_tester_small_kafka.sh` tests the model built from `small.txt` using `kafka.txt` as a test corpus, and the script `run_tester_kafka_small.sh` tests the model build from `kafka.txt` on the test corpus `small.txt`. Compare your numbers to the `correct_entropies.txt` file.

   (b) Build a model from the file `data/guardian_training.txt` and another model from the file `data/austen_training.txt`. Compute the entropy of the test file `guardian_test.txt` and the test file `austen_test.txt`, using both models. Report your numbers and your conclusions from these experiments!

4. (Named Entity Recognition) In this problem, we will explore the use of binary logistic regression for doing named entity recognition. You will **extend `BinaryLogisticRegression.py` to make it train a binary logistic regression model from a training set, and to use that model to classify words from a test set as either 'name' or 'not name'**.

   Have a look in the training file `ner_training.csv`. Every line consists of a word and a label. If the label is 'O', then the word is not a name; if it something else, then the word is a name of some kind. Currently we will consider all of these as just 'names'.

   The class `NER.py` reads a corpus on this format, and transforms it to a vector of labels, and a vector of features. The labels are either 1 (if the word is a name), or 0 (if it is not). There are two features: The first feature is 1 if the word is capitalized (starts with an uppercase letter), and 0 if it does not. The second feature is 1 if the word is the first word of a sentence, and 0 if it is not. For instance, from the row

   ```
   Demonstrators,O
   ```

we get the label 0, since the word is not a name, and the feature vector $(1, 1)$, since the word is capitalized and first in a sentence. These features are computed by the methods `capitalized_token` and `first_token_in_sentence`, respectively.

Note that when you call the class `BinaryLogisticRegression.py`, an extra "dummy" feature (which is always 1) is added to each datapoint. The datapoints are thus represented as a matrix `x` of size `DATAPOINTS` $\times$ (`FEATURES` + 1), and the corresponding labels as a vector `y` of length `DATAPOINTS`.

(a) Add code to the class `BinaryLogisticRegression.py`: the method `fit` should implement *batch gradient descent* to compute the model parameter vector θ, where θ_0 is the bias term, and θ_1 and θ_2 are the weights for features 1 and 2, respectively. The method `conditionalProb` should compute the conditional probability $P(label|d)$, where *label* is either 1 or 0, and d is the index of the datapoint. Test your model on the test set `ner_test.csv` by running the script

`run_batch_gradient_descent.sh`.

To view the progress of the algorithm, you may plot the gradient (see problem (b) below).

Batch gradient descent: (m is the number of datapoints, n is the number of features, α is the learning rate). Convergence happens when the absolute value of *all* the partial derivatives `gradient[k]` are below the constant `CONVERGENCE_MARGIN`.

> Repeat until convergence:
>   for k = 0 to n:
>   gradient[k] = $\frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}(h_\theta(x^{(i)}) - y^{(i)})$
>   for k = 0 to n:
> $\theta[k] = \theta[k] - \alpha$*gradient[k]

Recall that $h_\theta(x) = \sigma(\theta^T x) = P(y = 1|x)$.

(b) Track the convergence by inserting the following call at a suitable place in the loop.

`update_plot(np.sum(np.square(self.gradient)))`

However, note that plotting every iteration might slow down the computation considerably. If the learning is slow, try increasing the learning rate.

(c) Add code to the method `stochastic_fit` so that it implements *stochastic gradient descent* to compute θ. Use plotting to track the convergence. Test your code by running the script

`run_stochastic_gradient_descent.sh`.

What is the difference in performance compared to batch gradient descent?

Stochastic gradient descent:

```
Repeat a fixed number of times (e.g. 10m), or until convergence:
     Select i randomly, 0 ≤ i ≤ m:
     for k = 0 to n:
          gradient[k] = x_k^(i)(h_θ(x^(i)) − y^(i))
     for k = 0 to n:
          θ[k] = θ[k] − α*gradient[k]
```

(d) Add code to the method `minibatch_fit` so that it implements minibatch gradient descent. Use plotting to track the convergence. To test your code please run

`run_minibatch_gradient_descent.sh`.

What is the difference in performance compared to the earlier variants of gradient descent?

(e) Compute the **accuracy** of the model given the testset, as well as the **precision** and **recall** of the classes "name" and "no name". Present your numbers, and explain how you computed them.

(f) Try to improve on the results by adding a new feature, or by modifying some existing feature.

Optional part

5. (Classification for transition-based dependency parsing) In assignment 1, you implemented the function `compute_correct_move`, where we used a correct parse tree as an oracle for predicting the next best move given the current parser configuration. In reality, of course, we don't know the correct parse tree for every sentence. To remedy that, we have to create an automatic move classifier that is able to assign probabilities to every possible action, i.e. shift (SH), left-arc (LA) or right-arc (RA). In this problem we will train such a classifier using logistic regression.

   The problem is a 3-way classification problem (one has to predict one of the 3 classes SH, LA, RA), and therefore we are going to employ *multinomial logistic regression*. The skeleton for this problem is located in `DepParser` subfolder of the assignment 2 zip-folder.

   (a) Implement the methods for multinomial logistic regression having the comment `YOUR CODE HERE` in `logreg.py`. Detect when the model is overfitting by checking the loss on a validation set. In this task we'll employ **early stopping**, saying that the model overfits if the validation loss increases for P measurements straight (P is sometimes called **patience**). Incorporate early stopping to the method `fit` of `LogisticRegression`.

   To test your implementation of multinomial logistic regression on a toy problem run the following command:

   `python logreg.py`

   If implemented correctly, you should get 100% accuracy on the held-out validation (development) set.

(b) Implement the `build` and `evaluate` methods of the `TreeConstructor` class in the file `dep_classify.py` to build dependency parsing trees and calculate the ratio of correctly parsed sentences and the ratio of correctly assigned arcs (=unlabeled arc score, UAS), respectively. Use the trained logistic regression model to predict the next move for each parser configuration and select the one with the highest probability **if such move is valid**, otherwise select the move with the second highest probability, and so on.
Note that our move classifier can't predict the end of parsing, so you'll have to check for the terminal condition yourself (the buffer is empty and the stack only contains the ROOT node). To test your implementation run the following command:

```
python dep_classify.py
```

Our reference implementation gets a move-level accuracy of around 80%, the UAS of around 55%, and sentence-level accuracy of around 20% without proper hyper-parameter tuning. Try to push these numbers and see where you get!

To help you with the implementation, here are some hints:

- When running `dep_classify.py`, the trained logistic regression model is stored in the binary file `model.pkl` and then reloaded on the subsequent runs. If you want to train the model again, please remove `model.pkl` and run `dep_classify.py` again.
- When implementing a multinomial logistic regression, first make sure that the gradient computations are implemented correctly by checking whether the training loss steadily decreases. You can plot the value of the loss continuously by simply calling the method `update_plot` of `LogisticRegression` class, i.e. `self.update_plot(loss)`.
- When you have ensured that gradients are computed correctly, do **NOT** compute the training loss, as it will take too much time. Compute and plot only the validation loss to implement early stopping.
- Reuse as much code as possible from your implementation of `BinaryLogisticRegression`.
- Experiment with the number of features by changing the `THRESHOLD` instance variable of the `Dataset` class in `parse_dataset.py`.
- If you want to perform hyper-parameter search (e.g., adjusting learning rate, batch size, etc) in a principled manner, you could try doing a grid search, but this is **NOT** required.