DD1418/DD2418 Language Engineering
Johan Boye
2018-11-20

Assignment 4

Readings: Read chapter 5 in Jurafsky-Martin.

Code: The skeleton code can be downloaded from Canvas or from

`http://www.csc.kth.se/~jboye/teaching/language_engineering/a04/NER.zip`

Unzip the code in your home directory. Go to the folder `NER` and type:

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

Now everything needed for the assignment should be installed.

Problems:
In this problem set, we will explore the use of binary logistic regression for doing named entity recognition. Your main task is to **extend the program `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 1, 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`.

1. 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 2 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 sum of the squares of the partial derivatives `gradient[k]` is below the constant `CONVERGENCE_MARGIN`.

```
Repeat until convergence:
   for k = 0 to n:
      gradient[k] = (1/m) Σ_{i=1}^{m} x_k^(i) (h_θ(x^(i)) − y^(i))
   for k = 0 to n:
      θ[k] = θ[k] − α*gradient[k]
```

$$\text{gradient[k]} = \frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}(h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta[k] = \theta[k] - \alpha * \text{gradient[k]}$$

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

2. Track the convergence by inserting the call `update_plot(np.sum(np.square(self.gradient)))` at a suitable place in the loop (however, note that plotting every iteration might slow down the computation considerably). If the learning is slow, try increasing the learning rate.

3. 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]
```

4. Add code to the method `minibatch_fit` so that it implements minibatch gradient descent. Use plotting to track the convergence. Test your code using `run_minibatch_gradient_descent.sh`. What is the difference in performance compared to the earlier variants of gradient descent?

5. 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.

6. Try to improve on the results by adding some new features, or by modifying some existing feature, and/or adding regularization.