DD1418 Language Engineering with Introduction to Machine Learning
Johan Boye, Dmytro Kalpakchi
2019-11-14

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 `train_val_split` should **randomly** split the training data into training and validation set according to the given split ratio, e.g. ratio of 0.9 would mean that 90% of the training data will end up in the training set and 10% – in the validation set;
   - the method `loss` calculating the loss function for Logistic Regression;

- 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 datapoint itself.

When implementing a model, test if the model is correct on the small test set `ner_small_test.csv` by running the script `run_small_batch_gradient_descent.sh`. To ensure the correctness of the implementation you may plot the training loss value (see problem 2). Finish training by using the early stopping technique (see problem 3).

When you think that your implementation is ready, test your model on the larger test set `ner_test.csv` by running the script `run_batch_gradient_descent.sh`. Do **NOT** plot training loss when training on the larger dataset, since it will slow down the training considerably!

Batch gradient descent: (m is the number of datapoints, n is the number of features, α is the learning rate).

Repeat until the stopping criterion:
  for k = 0 to n:
    gradient[k] = $\frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}(\sigma(\theta^T x) - y^{(i)})$
  for k = 0 to n:
    $\theta[k] = \theta[k] - \alpha$*gradient[k]

2. In order to make sure that you've implemented gradient computations correctly, you can plot the training loss value by inserting the call `self.update_plot(self.loss(self.x, self.y))` in the suitable place. Do **NOT** plot training loss on every iteration, since loss computation will slow down the training considerably!

3. Detect model's 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 monotonously increases for P measurements (P is sometimes called **patience**). Checking whether the validation loss increases monotonously should happen in the function `compute_validation_loss` from `BinaryLogisticRegression.py`.

   Incorporate early stopping to the method `fit`. Plot both training and validation set losses by inserting the following call:

```
self.update_plot(
    self.loss(self.x, self.y), self.loss(self.x_val, self.y_val)).
```

   What kind of plot would you expect from theoretical perspective? How well does the plot you get fit your expectations?

4. Add code to the method `stochastic_fit` so that it implements *stochastic gradient descent* to compute θ. Use early stopping to decide when to finish the training. When implementing a method, try testing your code on the smaller dataset by running the script `run_small_stochastic_gradient_descent.sh`. When the implementation is ready, test your code on the larger dataset by running the script `run_stochastic_gradient_descent.sh`. What is the difference in performance compared to batch gradient descent?

   Stochastic gradient descent:

```
Repeat until the stopping criterion:
     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]
```

5. Add code to the method `minibatch_fit` so that it implements minibatch gradient descent. Use early stopping to decide when to finish the training. When implementing a method, try testing your code on the smaller dataset `ner_small_test.csv` by running the script `run_small_minibatch_gradient_descent.sh`. When the implementation is ready, test your code on the larger dataset by running the script `run_minibatch_gradient_descent.sh`. What is the difference in performance compared to the earlier variants of gradient descent?

6. 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.

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

For your reference, the training time for the NER classifier (averaged over 5 runs) trained on `ner_small_training.csv` and testing on `ner_small_test.csv` is given in the table below for different versions of gradient descent. The training-validation split ratio is 0.9, the minibatch size is 1000, the learning rate is 0.1 and the patience value is 5, with validation loss being measured every iteration for mini-batch and batch gradient descents and every 1000 iterations for the stochastic one. **Note, that these hyper-parameters are not necessarily optimal!**

Batch	Mini-batch	Stochastic
900s (9101 it)	110s (2435 it)	195s (789201 it)

Table 1: Training times of the NER classifier on the smaller dataset averaged over 5 runs for different versions of GD ("s" stands for seconds and "it" - for iterations)