Maximum Entropy Model Report

Wei Lu

October 6, 2018

1 Code structure

In order to implement the maximum entropy model, I have defined the following functions in the class MaxEnt:

```
def feature_labels(self,instances):
```

which extracts labels and features and their size from the training set. When extracting features from the reviews data set, I set up the following standard: the frequency of the word should be greater than 3 and the word doesn't appear in the stop words list. So in order to track the frequency of words, I use a dictionary to store the features, which has another benefit: later when I need to featurize an instance, it only costs constant time to check whether a word is in the feature set or not.

```
def featurization(self,instances):
```

which featurize an instance. The returning value is the feature vector which is stored in the field feature_vector of the instance. So the featurization has been done only once.

```
def feature_matrix(self,label,instance):
```

which creates an feature matrix for the given label and instance. During the process, the feature_vector of an instance is initialized.

```
def posterior(self,label,instance):
```

which calculates the posterior probability for the given label and instance. The result will be used to update gradient and to classify an instance.

```
def negative_loglikelihood(self,minibatch):
```

which calculates the negative loglikelihood for a given minibatch. This function is used to track progress of parameter fitting, i.e. the performance on the dev set.

```
def chop_up(self,training_set,batch_size):
```

which chops the training_set into minibatches according to the given batch_size.

```
def compute_gradient(self,minibatch):
```

which calculates the gradient of our objective function. It is done using matrix operation, so much faster than updating parameters one by one.

In the function train_sgd, I determine the convergence according to the performance on the dev set: if the dev loss goes up for two consecutive iterations, we stop training. I gave the function "train(self, instances, dev_instances=None, learning_rate=0.0001):” a default parameter learning_rate in order to accommodate different learning_rate for the two data sets: for the NAME dataset, I choose the learning_rate 0.1, whereas for REVIEW dataset the learning_rate is 0.01.

2 How to run the code and output to expect

Go to the folder which has all the python files in the terminal, run ”python test_maxent.py”. During the training, you will see the following information in the terminal: ”features shown:”, ”dict size:”(the actual number of features I used), ”labels size:”, ”batch_size:”, and the output is like the following:

```
test_names_nltk (__main__.MaxEntTest)
Classify names using NLTK features ... dict size: 412
labels size: 2
batch_size: 30
iter 1 , train loss: 2503.04619661 , dev loss 523.889934082 dev acc: 0.755
iter 2 , train loss: 2238.84321151 , dev loss 471.210789496 dev acc: 0.763
iter 3 , train loss: 2187.49783229 , dev loss 466.956622892 dev acc: 0.771
iter 4 , train loss: 2163.29808033 , dev loss 463.420243738 dev acc: 0.772
iter 5 , train loss: 2095.03054172 , dev loss 456.721868596 dev acc: 0.764
iter 6 , train loss: 2082.58829593 , dev loss 461.399650005 dev acc: 0.769
iter 7 , train loss: 2128.66194013 , dev loss 463.769794422 dev acc: 0.763
75%
ok
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 73094
dict size: 15343
labels size: 3
batch_size: 30
iter 1 , train loss: 5786.04965214 , dev loss 722.045656601 dev acc: 0.688
iter 2 , train loss: 4925.77301547 , dev loss 701.982204 dev acc: 0.717
iter 3 , train loss: 4455.58357859 , dev loss 702.678120673 dev acc: 0.721
iter 4 , train loss: 4088.91384018 , dev loss 705.826536561 dev acc: 0.726
iter 5 , train loss: 3770.0912798 , dev loss 715.319396217 dev acc: 0.718
72%
ok

----------------------------------------------------------------------
Ran 2 tests in 422.427s

OK
```

3 Experimental settings and results

3.1 Experiment 1

In this experiment, I tried minibatch size = 1, 10, 30, 50, 100, 1000. For each mini-batch size, I plotted the number of datapoints used to compute the gradient for so far (x-axis) against the accuracy of the dev set (y-axis). I set the threshold of word frequency as 11, under which the size of the feature set is around 6800. See below for the experiment result:

batch size 1000, learning rate 0.003, feature set size: 6894

batch size 100, learning rate 0.01, feature set size: 6836

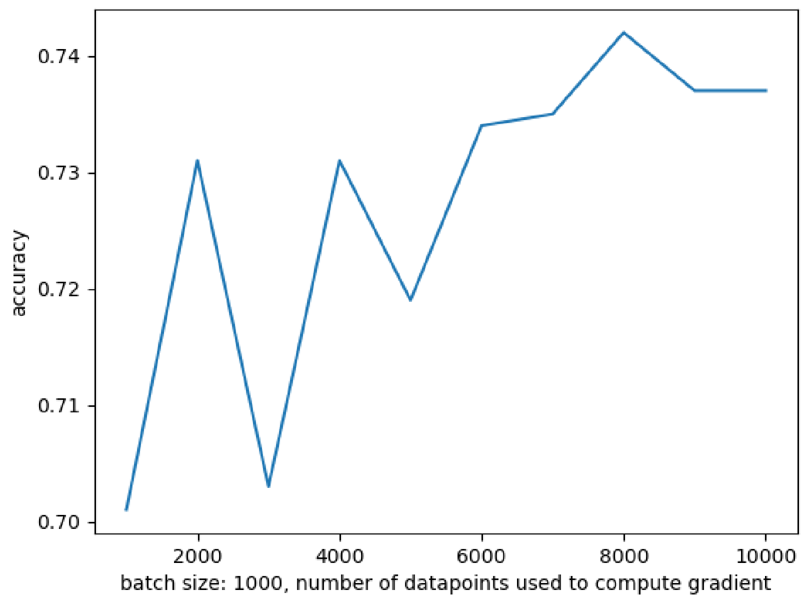

Figure 1: batch size 1000, learning rate 0.003

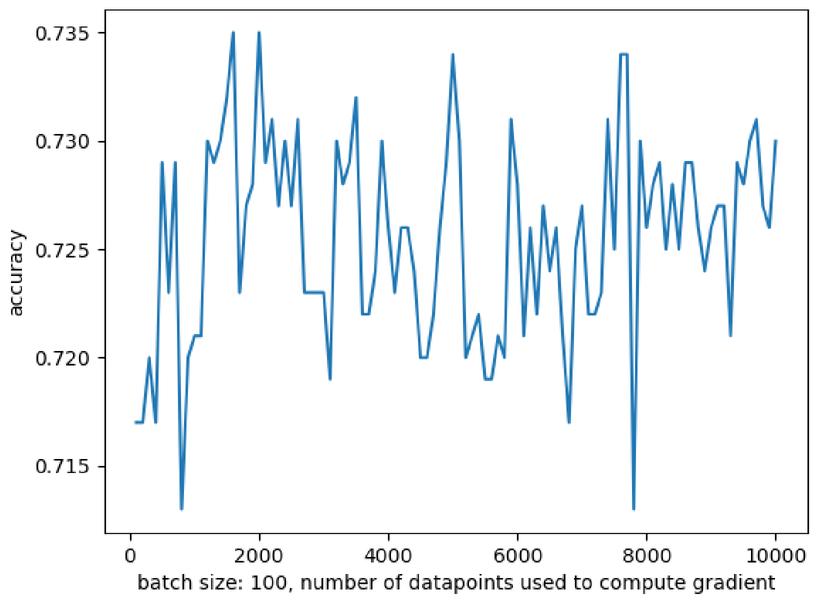

Figure 2: batch size 100, learning rate 0.01

batch size 50, learning rate 0.01, feature set size: 6767

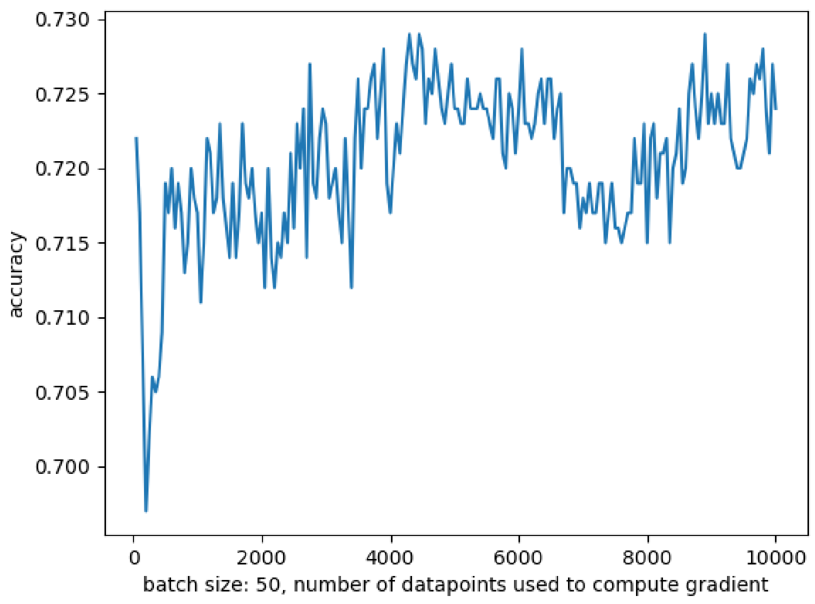

Figure 3: batch size 50, learning rate 0.01

batch size 30, learning rate 0.01, feature set size: 6798

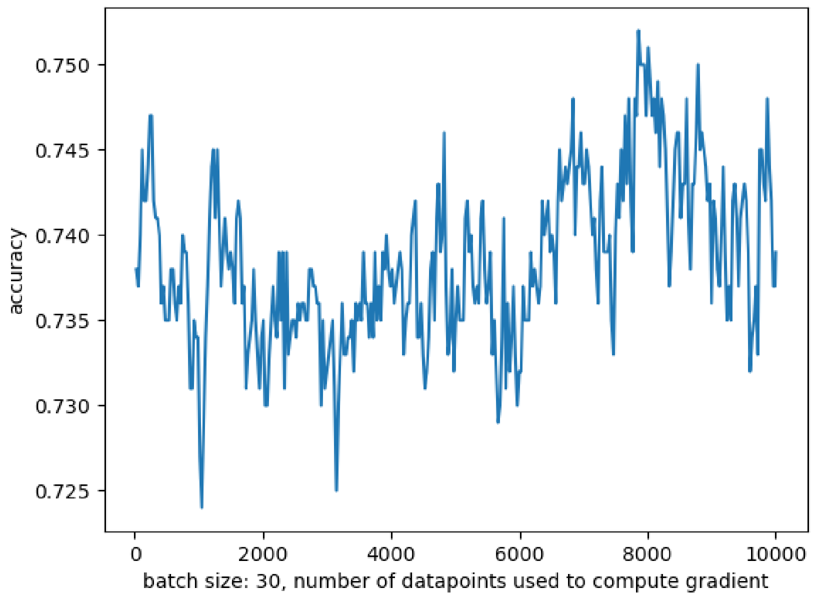

Figure 4: batch size 30, learning rate 0.01

batch size 10, learning rate 0.01, feature set size: 6759

batch size 1, learning rate 0.01, feature set size: 6874

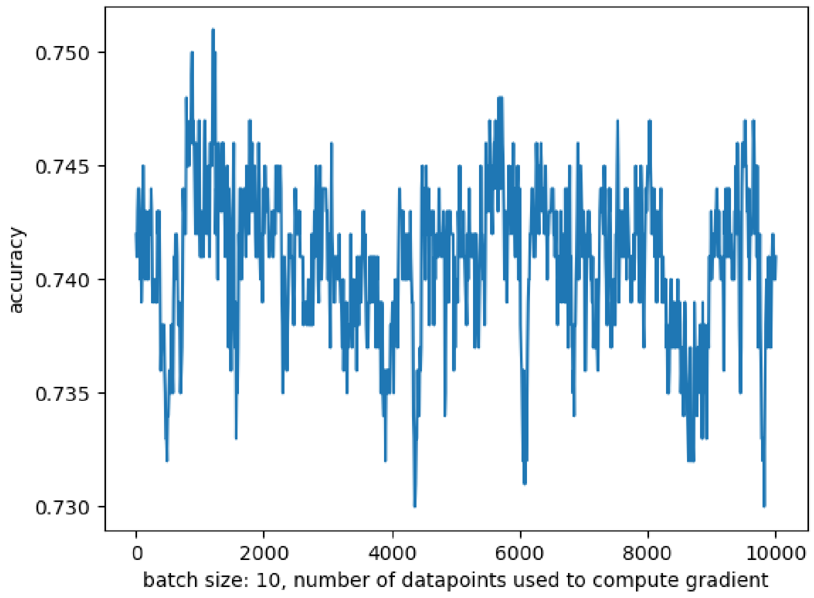

Figure 5: batch size 10, learning rate 0.01

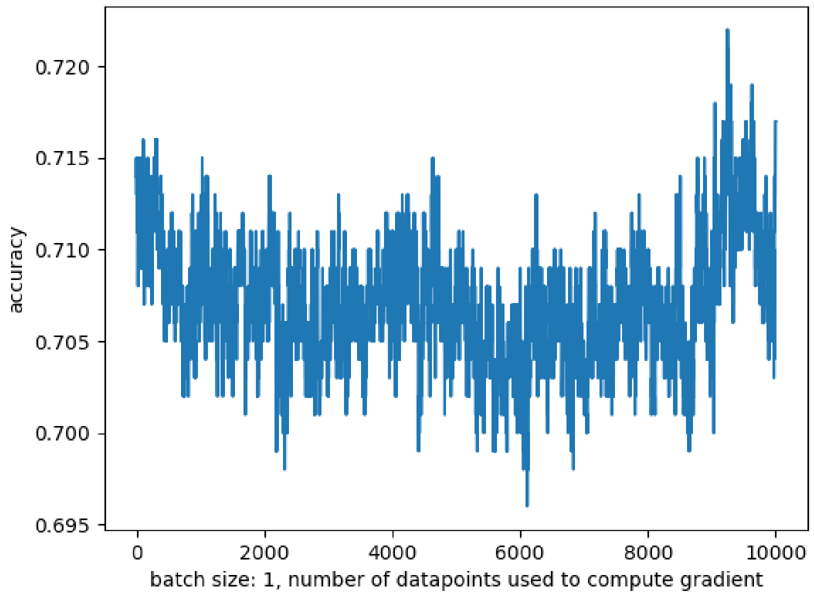

Figure 6: batch size 1, learning rate 0.01

From the experiment, I found that it is faster to update parameters for each iteration when the batch size is small. But the minibatches with bigger size tend to have better performance. Moreover, when the batch size is small, that is, smaller than 50, after we update the parameter for each minibatch, the performance on the dev set oscillate frequently, the performance on the dev set doesn't improve much. But for batch size greater or equal than 50, the performance is improving in the long run. (One exception is batch size 100, the performance isn't improving, I tried a couple of times)

3.2 Experiment 2

In this experiment, I fix the feature set of size about 6800 and fix reviews[100000:101000] as the dev set and reviews[101000:104000] as the test set. I varied the size of the training set 1000, 10000, 20000, 50000, 100000 and compared the (peak) accuracy from each training set size and made a plot of size vs accuracy. The following is the experiment result:

training set size 1000:

```
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 16987
dict size: 6169
labels size: 3
batch_size: 30
iter 1 , train loss: 648.051789943 , dev loss 836.370475392 dev acc: 0.642
iter 2 , train loss: 524.804496597 , dev loss 799.479318432 dev acc: 0.671
iter 3 , train loss: 450.407271385 , dev loss 798.940431263 dev acc: 0.67
iter 4 , train loss: 395.923022086 , dev loss 792.54767615 dev acc: 0.671
iter 5 , train loss: 357.12244324 , dev loss 791.379624828 dev acc: 0.674
iter 6 , train loss: 324.810031086 , dev loss 800.368050962 dev acc: 0.676
iter 7 , train loss: 297.777390612 , dev loss 802.368893151 dev acc: 0.673
67%
ok

----------------------------------------------------------------------
Ran 1 test in 56.199s
```

training set size 10000:

```
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 73805
dict size: 6951
labels size: 3
batch_size: 30
iter 1 , train loss: 5908.21288848 , dev loss 697.790199996 dev acc: 0.69
iter 2 , train loss: 5213.97267004 , dev loss 680.871785599 dev acc: 0.706
iter 3 , train loss: 4751.6789461 , dev loss 683.795939542 dev acc: 0.706
iter 4 , train loss: 4383.54041638 , dev loss 681.713276471 dev acc: 0.706
iter 5 , train loss: 4123.09140121 , dev loss 683.577529871 dev acc: 0.702
iter 6 , train loss: 3884.87383294 , dev loss 688.647181853 dev acc: 0.704
71%
ok

----------------------------------------------------------------------
Ran 1 test in 195.486s
```

training set size 20000:

```
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 111846
dict size: 6825
labels size: 3
batch_size: 30
iter 1 , train loss: 11623.1590429 , dev loss 662.002658271 dev acc: 0.731
iter 2 , train loss: 10450.9683189 , dev loss 656.29519664 dev acc: 0.729
iter 3 , train loss: 9774.46857055 , dev loss 659.417460568 dev acc: 0.736
iter 4 , train loss: 9222.6529428 , dev loss 658.974696618 dev acc: 0.743
iter 5 , train loss: 8849.41199882 , dev loss 673.723112889 dev acc: 0.738
iter 6 , train loss: 8544.56132984 , dev loss 680.12674973 dev acc: 0.752
72%
ok

----------------------------------------------------------------------
Ran 1 test in 343.169s
```

training set size 50000:

```
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 197479
dict size: 6932
labels size: 3
batch_size: 30
iter 1 , train loss: 28419.5647268 , dev loss 694.720915007 dev acc: 0.703
iter 2 , train loss: 26774.7916892 , dev loss 702.526985917 dev acc: 0.713
iter 3 , train loss: 25576.0553234 , dev loss 686.57193638 dev acc: 0.715
iter 4 , train loss: 24781.4426178 , dev loss 689.774602707 dev acc: 0.708
iter 5 , train loss: 24154.5830733 , dev loss 696.688488205 dev acc: 0.713
74%
ok

----------------------------------------------------------------------
Ran 1 test in 725.458s
```

training set size 100000:

```
test_reviews_bag (__main__.MaxEntTest)
Classify sentiment using bag-of-words ... features shown: 303439
dict size: 7026
labels size: 3
batch_size: 30
iter 1 , train loss: 57127.1363542 , dev loss 636.007646605 dev acc: 0.749
iter 2 , train loss: 54626.2039838 , dev loss 631.203539194 dev acc: 0.74
iter 3 , train loss: 53094.2241795 , dev loss 617.661695582 dev acc: 0.757
iter 4 , train loss: 52136.1632346 , dev loss 626.902262282 dev acc: 0.743
iter 5 , train loss: 51918.3559881 , dev loss 631.539481618 dev acc: 0.744
76%
ok

----------------------------------------------------------------------
Ran 1 test in 1498.683s
```

From the experiment result, we can see that the performance is increasing with the size of training set. So it is true that the bigger the training set the better the performance. Another thing to notice is the rate of improvement is decreasing. It means there is an ideal training set size, which can be used to update the parameters efficiently and effectively. Training set with size greater than this value will not

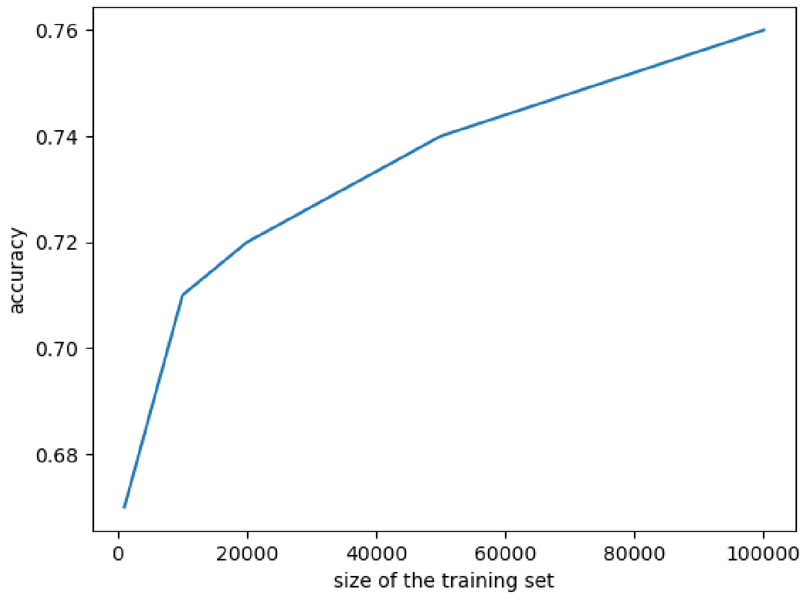

Figure 7: training size vs accuracy

make much difference.