DS222 ML With Large Datasets - Assignment 2

Apoorv Umang Saxena [1]

Abstract

This report shows the results of implementing Logistic regression using SGD for document classification in 2 different setups - one locally on a single machine and the other on the turing cluster.

1. Task and Method

The task is to classify a given document into any of 50 different classes. One document can belong multiple classes, for example it can belong to both "Articles_containing_video_clips" as well as "American_comedy_films".

We train a binary logistic regression classifier for each of the 50 classes separately. That is, the classifier for class *x* predicts whether the document belongs to class *x* or not. This is done using SGD in the local case and parallel SGD in the distributed case.

For prediction, we use each individual classifier to predict whether the given document belongs to that class or not. Then this predicted list is compared to the true list to check accuracy.

2. Parameters

Each of the 50 classifiers has 245389 weights (1 for each word in the vocabulary) plus 1 bias.

$Single\ \ class\ \ parameters\ \ =\ \ 245389\ +\ 1$
$Total\ parameters = 245389 * 50 + 50$

Hyperparameters:

α - Learning rate set to 0.01

Regularization was not used since the dense update required was causing unreasonable slowing of the training process. Testing with regularization was done and no improvement in the generalization ability of the classifier was seen.

3. Document preprocessing

The following steps were taken to reduce the noise present in the documents in the training data and test data. This helped in reducing the size of the vocabulary so that the feature vector size is reduced.

1. Removing URLs, punctuation and unicode character codes
2. Converting to lowercase
3. Stemming - using the Snowball stemming algorithm (Sno)
4. Removing stop words

After this, a dictionary of unique words was created from the training data (*vocab.txt*). Using this dictionary, each document was converted into a set of integers where each integer corresponds to the id of the word in the dictionary.

Finally, this set was used to create a one-hot encoding for a document. So each instance consisted of a sparse vector of size *245389* where the value of an index *i* is the number of times that word appears in the document.

4. Evaluating accuracy

Two different types of accuracy measures are done:

1. *Exact accuracy* - Correct prediction only if the predicted list of classes is exactly the same as the true list
2. *At-least-one accuracy* - Correct prediction if at least one of the predicted classes belongs to the true list

5. Classifier training - Local

Each logistic regression classifier was trained using iterative SGD. Convergence occurred in around 10 epochs for most of the classifiers. Accuracy can be seen in Table 1.

As can be seen in the table, having increasing or decreasing learning rates did not affect the accuracy noticeably. However, with unreasonably increasing learning rates, it was observed that training loss increased rapidly after a

few epochs. Extremely fast decreasing learning rate caused learning to effectively stop after 2 epochs.

Figure 1 shows the average training and validation loss against number of epochs (average over the 50 classifiers).

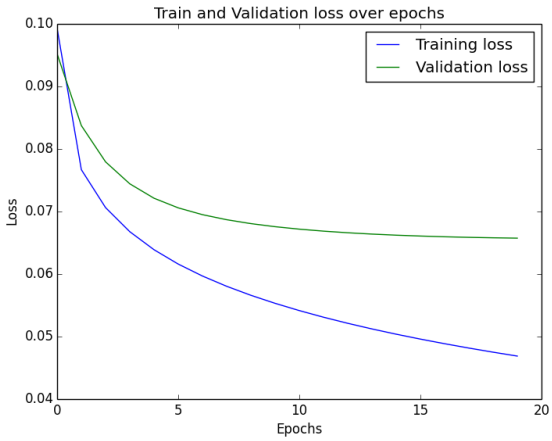

Figure 1. Average training and validation loss over epochs for iterative SGD

6. Classifier training - Distributed

Ray (ray) was used for implementing the parameter server.

Ray is an open-source framework for distributed ML in python. Since the local implementation was done in python using numpy as the only computing library, Ray was the easiest choice to extend this to a distributed setting.

Same methodology was used as in the local setting - 50 different classifiers were trained separately.

6.1. Synchronous SGD

Multiple workers run in parallel but synchronize their weights after each iteration.

6.2. Asynchronous SGD

Multiple workers run in parallel and asynchronously update the weights in the parameter server (global distributed memory).

7. Time Taken

7.1. Local implementation

The local implementation was tested on the *puri* server. On the full dataset, training one classifier took around **35 minutes** (20 epochs). On a single core machine, training all 50 classifiers would take 25 hours. However the multi-core capability of *puri* was utilized to train the 50 classifiers in parallel. This took around **40 minutes** on average.

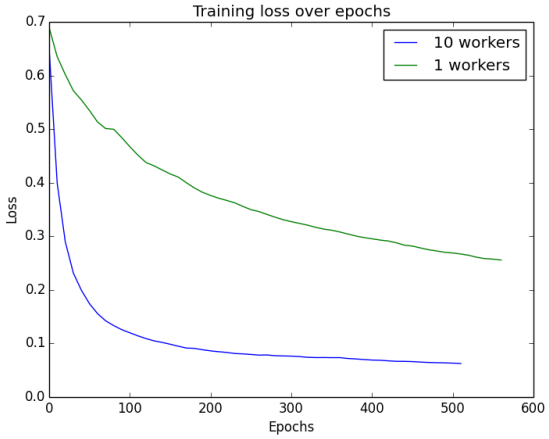

Figure 2. Training loss for single classifier over epochs for Synchronous SGD

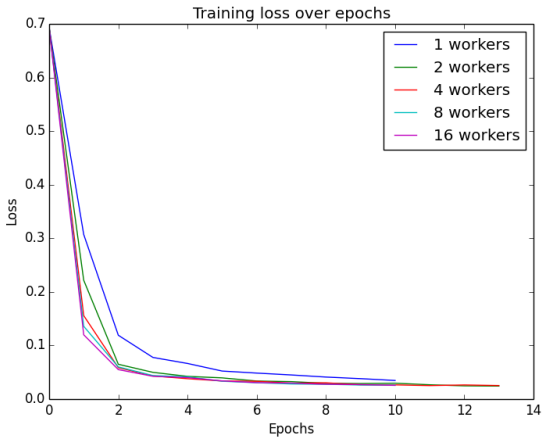

Figure 3. Training loss for single classifier over epochs for Asynchronous SGD

Testing took around **2 minutes** (over test data)

7.2. Distributed implementation

Synchronous SGD took very long to converge for a single class, on the order of **25 minutes**. This is probably because of the high cost of communication after each iteration. Hence proper testing was only carried out for first 5 classes.

Asynchronous SGD converged fast in **less than 3 minutes** for 4-10 number of workers (ie, time to train 1 classifier = 3 minutes. 50 classifiers took 150 minutes). Accuracy obtained here are almost the same as that obtained in the local case.

Learning rate	*Train Acc*	*Test Acc*
Constant	71.46/92.96	**72.21/88.81**
Increasing	-	70.38/88.49
Decreasing	-	71.43/88.63

Table 1. Prediction accuracies in local SGD. Format is *Exact-accuracy/At-least-one-accuracy*

	Train Acc	*Test Acc*
Sync (10 workers)	99.0*	96.0*
Async (best)	**72.2/91.6**	**72.1/88.7**

Table 2. Prediction accuracies in distributed SGD. Format is *Exact-accuracy/At-least-one-accuracy*. *Only evaluated on first 5 classes

8. Results

Training a single classifier is fastest using asynchronous SGD among the 3 implementations tested.

However, a point to be noted is that 50 different classifiers need to be trained in this problem. In this setting, training each classifier in an iterative manner while executing these 50 processes in parallel gives us the fastest overall training time.

References

Snowball Stemming algorithm. http://snowballstem.org/. Accessed: 2018-09-7.

Ray: A distributed system for ai. https://bair.berkeley.edu/blog/2018/01/09/ray/. Accessed: 2018-10-7.