NLP Assignment 2

Matthew Dunn - mtd368

10/16/16

1 Dataset Preprocessing

1.1 Merging Reviews

The files for both the Training and Test datasets were merged together using a simple *for loop*. The *for loop* appended the text from each file to a list which was then used to create a Pandas Dataframe for each dataset separately. A column was then appended to indicate whether the review was *pos* or *neg*. This Dataframe was then written to disk as *.csv* file with a Test or Train label.

1.2 Tokenizing

The merged *.csv* files for both Test and Training data were loaded into memory as Pandas Dataframes. The necessary tokenization, vocab building, and additional preprocessing tasks were executed in a vectorized manner. By vectorizing the preprocessing work, the code is scalable to a much large dataset if necessary for future projects. Specific preprocessing steps were to create a vocabulary for 10k unigrams, 5k bigrams, and 2.5k trigrams using the *collections* library which allowed efficient selection of the most frequently occurring n_grams above the given thresholds. Then each respective vocabulary was cast as a set object for efficient comparison to the raw text in the movie reviews. Any word not in the vocabulary *set*, i.e., out-of-vocabularly, was replaced with $< oov >$ for each n_gram type. Lastly, each *neg* or *pos* label was converted into a [0,1] or [1,0] vector and then concatenated together to form the target variable.

1.3 Fit, Split and Pickle

Once the data was cleaned and out-of-vocabularly n_grams were mapped to $< oov >$, the Vocabulary Preprocessing library [1] from Tensorflow was used to assign indices to each n_gram. The vocabulary preprocessor was *fit* on the complete *x_train* dataset and then complete *x_train* and *y_train* datasets were split using the Sklearn Preprocessing package *train_test_split* to create a 0.1 Validation dataset and 0.9 training dataset. The same procedure for preprocessing

[1] `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/learn/python/learn/preprocessing/text.py`

was used for the Test dataset with the notable difference being that the existing *fit* built on the training dataset was used to *transform* the Test dataset, thus assigning the n_grams indices from the Training dataset to the Test dataset. The output for each was saved as a *pickle* file so the testing of different model architectures wasn't delayed by preprocessing the raw text data repetitively [2].

2 Architecture

After creating *pickle* files a simple MLP architecture was built. The initial model consisted of an *embedding layer* with an embedding matrix *embed*:

$$embed \in R^{v \times d}$$

where d is the embedding dimension and v is the vocab size. The dimension v varies according to whether bi_grams and tri_grams are included in the vocabulary in addition to uni_grams. This layer computes the mean for a single *x_train* sample by selecting the vectors from the embedding matrix that correspond to the *x_train* sample indices and computes the mean across dimension d. The resulting vector $[x \in R^d]^T$ is the output of the layer.

The next layer in the model is a *hidden layer* that takes as input x. The layer's dimensionality is set by the parameter h. The values tested for h ranged from 200 to 2000, but the best performance was observed with $h = 2000$. The *scores* for the layer are computed using the weight matrix $W \in R^{d \times h}$ and bias b. The output of the layer is computed by applying a point-wise non-linearity to the *scores* and outputs $h\_out(x)$.

$$h\_out(x) = relu(Wx + b)$$

To mitigate over-fitting, a *dropout layer* was added that removed 0.5 of $h\_out(x)$ entries during *training* to mitigate over-fitting. Various dropout rates were tested from 0.25 to 0.75, but it was found that 0.5 was the most effective.

Lastly, a *output layer* calculated the predicted values and scores which were then used to calculate the loss and accuracy of the model.

2.1 Multiple Hidden Layers

Initially, one *hidden layer* was used for uni_grams, bi_grams, and tri_grams, but to determine whether an additional *hidden layer* could improve the model's accuracy a second *hidden layer* and corresponding *dropout layer* were added. The result of this experiment was a model that didn't generalize well to the Test dataset due to over-fitting on the Training dataset.

[2] Using the *pickle* files provided a 15x speed up in loading the necessary data for training and testing.

3 Training Procedure

3.1 Optimizers

Several optimizers were tested, but by far the best performance was achieved using the Adam algorithm. The SGD and Adadelta optimizers were slow to converge and resulted in somewhat noisy performance on when evaluated on the Validation and Test datasets. As a result all errors reported below are based on using the Adam optimizer.

3.2 Learning Rate

The learning rate used for these experiments was set initially at 1e-2 (0.01). A larger initial learning rate of 1e-1 (0.1) was tested, but it was found to cause erratic optimization and never achieved and accuracy above apprx 0.5. To avoid the learning rate being too large towards the end of model training, the initial learning rate was decreased using an exponential decay rate as training proceeded. Lastly, the AdamOptimizer's beta parameters were experimented with to understand how the 1st and 2nd moment calculations impact the optimization, but results were inconclusive.

3.3 Train, Val, Test Error

The error rates on the Train, Test and Validation datasets follow with embedding dimension as d, hidden layer dimension as h, n_gram size as n, dropout rate as dp, max sentence length as mx, and L2 regularization as $l2$:

Params	Train Acc.	Val Acc.	Test Acc.
$d = 128$, $h = 2000$, $n = 1$, $dp = 0.5$, $mx = 300$, $l2 = 1$	apprx 0.9	apprx 0.86	0.862
$d = 128$, $h = 2000$, $n = 2$, $dp = 0.5$, $mx = 300$	apprx 0.9	apprx 0.87	0.856
$d = 128$, $h = 2000$, $n = 3$, $dp = 0.5$, $mx = 300$	apprx 0.9	apprx 0.88	0.871

4 Facebook FastText

The first attempt to use Facebook FastText was done using a ported version written in python, but performance was abysmal. Consequently, the binary was installed directly from the Github repo. Once running from the command line, the training and testing of the models was incredible quick. Training the model using FastText took apprx 7 seconds. Testing the model on the Test dataset was just a fast and the Precision and Recall scores both were at 0.858.