Report of the Log-Linear Model Implement

1700013003, Zejun Wang, EECS, Department of Artificial Intelligence

Project Information(Please don't break the file structure)

The File Structure(Very Important)

```
/train.py  # my training entrance
/test.py  # my testing entrance
/ReadMe.pdf  # the document that you're reading
/run.sh  # the bash shell script to run the project
/data  # the directory to store my processed data
        /clean_train.csv  # the processed training text
        /clean_test.csv  # the processed test text
        /clean_valid.csv  # the processed valid text
        /features.pkl  # the features which are chosen previouly
        /__init__.py
        /dataLoader.py  # used to load the data and extract features for given text and label
        /previous_scripts.7z  # the python scripts and .ipynb notebook for preprocessing
        /raw_data.zip  # contains the raw data and the cleaning xcript
/model  # the directory containing the parameters
        /classifier.pkl  # contains the parameters
/source  # the directory containing the python script that define my model
        /__init__.py
        /model.py  # definition of the model
        /metrics.py  # definition of the ACC counter and the F1 counter (written by my self
/from start  # the codes if you want to run from the raw data
        /dataLoader.py
        /formalizer.py
        /metrics.py
        /model.py
        /run.sh
        /test.py
        /train.py
```

There're **two way** to examine this project.

Start Right in the Root Directory(Recommended)

If you want to examine the test result of the model, please run like this using the already extracted features and the cleaned data because the valid data is extracted from the training data randomly, besides, at each beginning of the epoch, the training data is shuffled as well.

If there's already a file of parameters provided in the model directory and you still want to start over, please run like these

```
chmod +x run.sh
./run.sh train retrain
```

The script will make a backup of the previous parameters. But if you retrain again, the backup will be overwritten.

But if the model directory is empty, run like this

```
chmod +x run.sh
./run.sh train
```

If you want to run a test, run like this

```
chmod +x run.sh
./run.sh test
```

Start Over from the Raw Data

Just copy the files in the "from start" directory to where the raw data files store. Then run like this:

```
chmod +x run.sh
./run.sh
```

The script will start from cleaning the data, drawing features to training and testing. But pay attention, it cannot be run separately to check these models one by one.

The packages which are required to run the scripts

- python3 (version 3.7 is recommended)
- Numpy
- Pandas
- sys
- os
- tqdm
- re
- cPickle

How to Run the Shell Script

The Implementation of the Log-Linear Model

Preprocess

1. I used a regular expression to remove each head of the posts and the punctuators which acted as delimiters, then did a naïve word segmentation ---- separating the numbers and the letters for example, changing '340m/s' to '340 m/s'.
2. I removed the stop words using a chart collected from the Internet and then counted the frequency of each word. Then I replaced the words with their frequency less than 5 with a special token <oov>. This is a step of an implementation of [1], which introduces an improved MI feature extraction method, which will be mentioned as follows.
3. I use **Unigram** with **Mutual Information** to build my features. To measure the importance of word w_i to the news category c_j,we just need to compute the following probability:

$$log\frac{P(w_i|c_j)}{P(w_i)}$$

$$P(w_i|c_j) = \frac{count(w_i, c_j)}{count(c_j)}$$

$$P(w_i) = \frac{count(w_i)}{|D|}$$

   where $count(w_i, c_j)$ is the number of documents in class c_j that has word w_i appeared, $count(c_j)$ is the number of documents in class c_j, $count(w_i)$ is the number of documents in the whole dataset that has word w_i appeared and $|D|$ is the number of the documents in the dataset. But **pay attention**, these measurements were done only on the **training set**, which means the **"dataset"** mentioned previously is the **training set** in common sense. I chose **3000** best words for each category as its features which means the whole vector label has **60000** dimensions.

Feature Construction

The feature vector is a binary vector of 60000 dimensions with a pattern of

$$x[i] = \begin{cases} 1 & w_i \in p \wedge c_p = j \\ 0 & else \end{cases}$$

w_i is the word in position i which is a feature of category j. p is the current processing paragraph.

In theory, we need to build 20 vectors for each paragraph because we don't know their category when interfering but in practice, we can simply add them together because the corresponding locations of other vectors are all 0. It's a very important trick to save memory and time.

Model Construction

To mention before, all the vectors in the following descriptions are all column vectors.

Because I have a feature vector of 60000 dimensions, the corresponding weight vector should have 60000 dimensions as well. Now I give it an annotation w and from now on, the words are represented with word t in short of $token$. I use h for features. For category i, I compute its probability with the following equations:

$$p(i) = \frac{exp(w^T h_i)}{\sum_j exp(w^T h_j)}$$

In practice, we don't need to compute 20 dot product. Using the tip mentioned before, we just need to compute them once. If we place the features in order, for example, the elements of indexes from 0~2999 are of category 0 and index 3000~5999 are of category 1 and so on, we just need to apply the dot product once and then reshape the vector into a 20*3000 matrix to perform an element addition after applying a matrix exponential operation. This is very important.

The Implementation of the optimization process

Gradient Ascent

I use Log-MLE to learn the parameters. Formally, I try to maximize

$$Y = \sum_i logP(y_i) = \sum_i w^T h_{y_i}^{(i)} - \sum_i \sum_j exp(w^T h_j^{(i)})$$

where y_i is the ground-truth for article x_i and the superscript of the features are related to the corresponding article. It's hard to get an analytical expression of the best w, so the gradient ascent strategy is applied:

$$\nabla_w Y = \sum_i (h_{y_i}^{(i)} - \sum_j exp(w^T h_j^{(i)}) h_j^{(i)})$$

$$w' = w + \eta * \nabla_w Y$$

where w' is the updated parameters and η is the learning rate.

There're many tricks to compute on batches combining the matrix transformation and the previously mentioned tricks to cut down the memory and time expense which are hard to explain here in detail. But I need to mention one thing here is that no explicit loop appears in my code. Just common Numpy basic matrix operations such as element-wise addition and element-multiplication can cover this task. About the function that I use in my model definition, I'll mention them in the appendix.

'SGA' and 'mini-batch SGA'

I've got no idea about the mathematical theory behind the popular SGD algorithm. I just know that applying SGD can have a better generalization result. Lacking computational power, I guessed that I could apply the 'mini-batch SGA' algorithm to the log-linear model optimization. Luckily, it worked well. It gave me a fast convergence. Only a little time and memory were needed.

The training tricks

I use 10% of the training set as the validation set. The samples in the validation set were chosen randomly. The training epoch were determined by the intermediate result, which involves the early stopping and the endurance strategy. Besides, a learning rate schedule is used. When a batch training is complete, a test on the validation test is performed. If the test result is worse than the best result, I'll cut the learning rate in half and go into an endurance mode until the best result is updated. If the best result haven't updated for more than 5 epochs, the training process will be cut off.

At the beginning of each training epoch, I shuffled the training set.

Only the parameter with the best validation result is saved. The initial learning rate is tuned to be 0.1. The initial parameters were drawn from a normal distribution.

The Training Result and the Testing Result

	Accuracy	Macro-F1
Train	99.27%	0.9927
Validation	81.52%	0.8103
Test	72.61%	0.7190

Appendix: The Numpy Functions that I Use

1. np.random.normal, np.ones, np.zeros
2. np.multiply, np.sum, np.expand_dims
3. np.putmask
4. np.array
5. np.stack

References:

[1]: ZHU Hao-dong，CHEN Ning，LI Hong-chan. Optimized mutual information feature selection method. Computer Engineering and Applications，2010，46（26）：122-124.