|  |  |
| --- | --- |
| ZZEN9444  Assessment 3:  Language Processing |  |
|  |  |
| 11 October, 2021  PAUL JOHN CRONIN  Z5330951 | |  |

# 1. Overview

This assessment asks the students to analyse a series of reviews and to extract a Sentiment rating (0 or 1) and a business Categorization (0, 1, 2, 3, 4) for each. The lectures presented sophisticated neural network architectures to be able to handle whole reviews, but early on I decided to use a tried-and-true architecture, that being a “bag of ngrams”. In hindsight, this was not a wise decision.

My first inclination was to create two separate networks, each independently tuned for each specific requirement, because:

1. Ngrams that may contribute to Categorization may not also contribute to Sentiment,
2. Knowing the Categorization first is probably helpful for computing the corresponding Sentiment of the review; for example, “it really burned” might have a positive sentiment for an Automotive, but the same word group might have a negative sentiment when it comes to Health & Medical.
3. The stopWords may be significantly different for Categorization than for Sentiment analysis.
4. Overall, the two processes are inherently different: business categorization is a discrete classification, but sentiment is a continuous process from negative to positive. These lend themselves to different – business Categorization to SoftMax, and Sentiment to a Sigmoid.

However, the a3main.py code, that can’t be modified, doesn’t easily allow for two separate nets – for example there is only an allowance for one set of stopWords. Hence, it was decided that a single network would be built, that allowed for a single choice of n-grams and stopWords, but which tried to alleviate the issues listed above, and still allow the Categorization to affect the Sentiment calculation (it turned out by allowing only certain n-grams, that I didn’t need stopWords at all).

# 2. Mistakes made & lessons learned

I wish to acknowledge early in this report some misunderstandings and mistakes that were made during this assessment, and what I have learnt from them.

1. My choice of computational platform: I didn’t fully appreciate how many computational resources would be required for this assessment. I am running a 2013 MacBook Pro, which has a CUDA-capable GPU. I spent a few days unsuccessfully attempting to enable the GPU, which was a rabbit hole I should not have gone down. Instead, I should have purchased as much online GPU time as necessary. This is a mistake I will not make again.

2. During the lectures, a number of architectures were presented that would have been a wiser choice, such as Long Short Term Networks and Gated Recurrent Unit networks. However, because I mistakenly thought the assignment didn’t require that much sophistication, I decided to go with the tried-and-true, brute force, n-gram architecture.

3. My plan was to develop the architecture in Matlab, a language I have much more experienced with, then port to Python. I had a good deal of early success in this process, I was able to choose the optimal n-grams, but I didn’t start using GloVe word embedding till after the porting process. This was a mistake, as GLoVe does not handle n-grams well, only single word embeddings. I should have made the switch to an LSTM network when I realized this, however, I continued to try to make the n-gram architecture work with GLoVe.

These mistakes; the choice of neural-net architecture, the use of an elderly CPU, and the slowness of torch cosine\_similarity caused incredible slow processing. It had been my intention to use 2,000 n-grams inputs, but in the end, only used 150 n-grams, which required a 15 hour run-time.

These have been hard lessons to be learned, but I have learnt them.

# 3. Pre-processing

One of the reasons I chose the n-gram architecture was because I believed that it would make stopWords unnecessary. Instead of defining all the words I would initially block with stopWords, I would choose the optimal set of n-grams, and reject all other words that didn’t fall within some similarity to that optimal set (this is why the stopWords block of code is empty).

I wished to be judicious in choosing the n-grams that would be used as inputs to the neural net, words which would contribute significantly to both the Categorization and Ratings. It was decided that 1-, 2- and 3-grams would be used for this network. For this pre-processing step, I used the same Tokenisation code as for the eventual neural network: all non-alphanumeric characters were stripped from the review, and then every possible 1-, 2- and 3-gram was generated for every review. This resulted in a total of 2,260,393 n-grams.

This is far too many for a neural net to process. To downselect to the optimal, minimal number of n-grams, a two-step process was undertaken:

1. First, only n-grams that occurred at least 30 times in the entire corpus were selected. This reduced the possible number of ngrams from 2,260,393 to 19,915 (see Appendix A.1).

2. Next, a process very similar to a Singular Value Decomposition was used to arrange the possible 19,915 n-grams in order of which best classified the review and sentiment correctly (see Appendix A.2 & 3).

This last process is also very similar to a neural net linear model (with no activation functions) was implemented which chose the 2000 n-grams that optimally chose the best ngrams – resulting in 75+% accuracy in determining the Categorization and 90+% accuracy in determining the Sentiment – see Figure below. I felt that this would be a good start, which a neural net would only improve upon.

Chart, line chart, scatter chart

Description automatically generated

Figure 1 This graph shows the linear model accuracy for both Categorisation and Sentiment, as a function of the number of ngrams chosen. Please note that this involved choosing the optimally best n-grams in order. To validate this, two additional analyses were done, where the n-grams were chosen at random, and where the optimally worst n-grams were chosen. The results of those analysis are in Appendix A.3.

# 4. Net architecture, algorithms, and enhancements

## The architecture that was planned

The architecture plan was to use approximately 2000 n-gram inputs, followed by two hidden, fully connected layers (with 285 and 40 hidden nodes respectively, and tanh() activation functions). Following the second hidden layer would be fully connected layer giving the Categorization result (with SoftMax). The results of the Categorization and a copy of the last hidden layer (with 40 hidden nodes) would be used to compute the Sentiment analysis (with a sigmoid).

The original 2000 n-grams was chosen from the pre-processing analysis, which should generate approximately 90% accuracy, while the number of initial nodes was found by the heuristic of a common factor of 7x reduction with each step.

2000 n-gram inputs

CATEGORIZATION

1st hidden layer 285 nodes

2nd hidden layer 40 nodes

SENTIMENT

Copy of 2nd hidden layer

## The architecture that was implemented

Unfortunately, due to processor slowness, the planned architecture was not able to be implemented. Instead, a significantly simplified architecture was implemented. The number of n-gram input nodes was reduced from 2,000 to 150, the number of hidden layers was reduced from 2 to 1, and there was no feed-forward from the Categorization to the Sentiment computation. But even this significantly reduced architecture still required more than 14 hours to run.

CATEGORIZATION

150 n-gram inputs

Hidden layer 30 with nodes

SENTIMENT

I would also have liked to use dropout a great deal more, as I feel that would have eliminated many of the unnecessary (and slow) redundant connectivity of the network.

## The algorithm

The algorithm implemented was quite different from the algorithm planned. One of the early mistakes that I made was that GloVe would handle n-grams as well as it handles words, but that is not correct. I had hoped that I could train the algorithm uses those up to 3-word phrases, and GloVe would match “phrase vectors”, but this is not the case. The remnants of this approach are still visible in student.tokenization.

As I discussed in Section 2, I should have switched algorithms at this point, but I pushed forward using n-grams. The algorithm as implemented is as follows:

1. 10,000 optimal n-grams are stored, in order of optimality, within student.py. During \_\_init\_\_, each of the words in these optimal n-grams are transformed into GLoVe vectors and stored.
2. Each review, in the form of a sequence of GloVe vectors is received, and broken into every single 1-, 2- and 3-gram GloVe vectors and stored.
3. An input\_matrix is created and stored which will become the input to the neural network. Each column of this matrix corresponds to a particular optimal n-gram slot, and each row corresponds to every review in the batch.
4. Each of the stored input n-gram GloVe vectors is compared to each of the optimal n-gram GloVe vectors. If any of the input are within a threshold distance, the corresponding data point in the matrix is incremented.
5. Once all the n-grams for all of the reviews in the batch have been digitized, this input\_matrix is fed to the neural network (which contains a fully connected hidden layer with approximately 30 hidden nodes and tanh() activation function). This single hidden layer then feeds both a Sentiment and Categorization output, with SoftMax and Sigmoid classification respectively.
6. The outputs from these classifications is sent to BCELoss and CrossEntropyLoss for the Rating and Classification elements respectively.

Given what I know now, I would not choose this architecture or algorithmic approach again, rather I would choose an LTSM approach. The n-gram approach worked well when it comes to words but suffered problems trying to get GloVe word vectors to work.

# 5. Cost functions and optimizer

The activation function used was Tanh() at the hidden nodes. I wished to maintain both the positive and negative sense, especially when it came to sentiment. Of course, Sigmoid was an alternative, but that would have lost the negative element – all forward multiplications would have been zeroed.

The loss functions at the classification nodes were:

1. Sigmoid function for Sentiment (0 being a negative review, 1 being a positive review). This function was chosen as it cleanly selects between only two possible classification.
2. SoftMax for Classification, as this cleanly selects and amplifies for a multi-element classification.

Other loss functions considered were Squared Error and Weight Decay.

The cost functions used to minimize over the entire problem were:

1. BCELoss() for the Ratings,
2. CrossEntropyLoss() for the Categorization,

And the total loss was simply the sum of these two. I should have increased the weight of Categorization, as it is marked higher than Ratings.

The optimizer selected was ADAM, as I had previous good experience with it, and rather poor experience with SGD.

Other parameters considered:

* Epochs: I mostly explored the space with 5 Epochs, simply because of the slowness of the device. I feel if I had trained with a larger number of Epochs, greater than 10, I would seen greater convergence eventually – I simply did not have the computing power to wait.
* The Learning Rate was initially set at 0.01, then 0.05, and finally at 0.30, trying to push to convergence faster. This was definitely a double edged sword, as a higher learning rate did not noticeably improve the outcomes.
* Batch size: I found little difference in using batch size between 16, 32 and 64, so in general I used 16 – it gave me results a little faster than size 32.
* I stayed with the train/val split of 80:20, and did not explore the use of dropout – although that could have probably helped significantly.

# 6. Selection of GloVe dimensions

I have learnt a great deal about GloVe – the power and limitations. For my needs, which were only to compare to optimized n-grams, the 50-dim GloVe vectors were more than enough. I even tried reducing the dimensionality to improve the speed of the algorithm.

My early mistake, misunderstanding that GloVe could handle phrases and not just individual words, was costly, and a limitation of GloVe that I had not been aware of.

I also explored the different ways to measure distance between vectors. While I much prefer tensor.cosine\_similarity to find the “angle” between vectors, I found it about 40% slower than tensor.norm, and so switched to the faster measurement.

# 7. Validation set

From lectures, we learned the value of the validation set, including such techniques as stopping early and using weight decay to avoid overfitting.

Much of the work to avoid over-fitting had been done during the pre-processing work: the optimal n-grams were carefully selected after removing rare words (which would prevent us from fitting to them) and then finding which n-grams truly had the highest impact on the whole dataset. By examining Appendix

Those n-grams were ranked by that impact, and we could easily choose from the best. This became important as I had to reduce the number used in the process; there are 10,000 optimal and ranked n-grams included in student.py, but at times I only used 150 or even fewer. I feel that if I had the computation power to increase this to 2,000 or so, there would be dramatically improved results.

# Appendix A: Pre-processing code

## Appendix A.1: Selecting the optimal n-grams

import torch

from torchtext.legacy import data

from torchtext.legacy import vocab

import csv

from sklearn.feature\_extraction import DictVectorizer

from config import device

import student

# defines datatype textField

textField = data.Field(lower=True, include\_lengths=True, batch\_first=True,use\_vocab=True,

tokenize=student.tokenise,

# preprocessing=student.preprocessing,

# postprocessing=student.postprocessing,

stop\_words=student.stopWords)

# defines datatype labelField

labelField = data.Field(sequential=False, use\_vocab=False, is\_target=True)

# this loads all the data

dataset = data.TabularDataset('train.json', 'json',

{'reviewText': ('reviewText', textField),

'rating': ('rating', labelField),

'businessCategory': ('businessCategory', labelField)})

busCat = []

ratCat = []

for i in range(50000):

busCat.append(dataset[i].businessCategory)

ratCat.append(dataset[i].rating)

f = open('bus\_cat.txt', 'wb')

writer = csv.writer(f)

data\_line = bytearray(busCat)

f.write(data\_line)

f.close()

f = open('rat\_cat.txt', 'wb')

writer = csv.writer(f)

data\_line = bytearray(ratCat)

f.write(data\_line)

f.close()

unique\_words ={}

for i in range(50000):

for word in dataset[i].reviewText:

unique\_words[word] = unique\_words.get(word, 0) + 1

red\_unique\_words = {k:v for (k,v) in unique\_words.items() if v > 10}

unique\_zeroed = dict.fromkeys(sorted(red\_unique\_words, key=red\_unique\_words.get, reverse=True), 0)

#unique\_zeroed = dict.fromkeys(sorted(unique\_words, key=unique\_words.get, reverse=True), 0)

f = open('reviews\_ngramsXX.txt', 'wb')

writer = csv.writer(f)

#write the review data, in the form of ngrams, to file

for i in range(50000):

dict\_line = dict(unique\_zeroed) #creates an empty dictionary, with correct ngram keys, but zeros for values

for word in dataset[i].reviewText: #goes through each review, and if possible, adds each ngram

if word in unique\_zeroed:

dict\_line[word] +=1

data\_line = bytearray(list(dict\_line.values())) #creates line of data to be written to file

# print(list(dict\_line.values())[0:10])

# print(dict\_line.keys())

f.write(data\_line)

if (i % 1000 == 0):

print(i)

f.close()

print(len(unique\_words))

print(len(red\_unique\_words))

## Appendix A.2: Linear model to select optimal ngrams (Matlab)

% load each review as a set of ngrams

no\_words = 19915;

fileID = fopen('reviews\_ngrams.txt','r');

load\_data = uint8(fread(fileID,[no\_words,50000],'uint8'));

fclose(fileID)

raw\_data = load\_data';

clear load\_data

% loads the business category data and codes it

fileID = fopen('bus\_cat.txt','r');

B = fread(fileID,'uint8')';

fclose(fileID);

% loads the ratings category data and codes it

fileID = fopen('rat\_cat.txt','r');

R = fread(fileID,'uint8')';

fclose(fileID);

% forms the results matrix

bMatrix = zeros(50000,7);

for i = 1:50000

bMatrix(i,B(i)+1)=1;

bMatrix(i,R(i)+6)=1;

end

% normalizes the review and results data

normalized\_data = normalize(single(raw\_data));

bMatrix = normalize(bMatrix,2);

clear raw\_data

% computes the weights for each ngram

weights = lsqminnorm(normalized\_data,bMatrix);

% computes the most import ngrams in order

norm\_data = vecnorm(weights,2,2);

optimal\_ngams = sort(norm\_data,'descend');

## Appendix A.3: Randomly & intentionally wrongly chosen n-grams

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated