CS 533: Natural Language Processing **(Due: 02/18/20)**

Assignment 2

Name: Madhu Sivaraj*, NetID:* ms2407 *Students discussed with:* N/A

Problem 1: Log-Linear Models

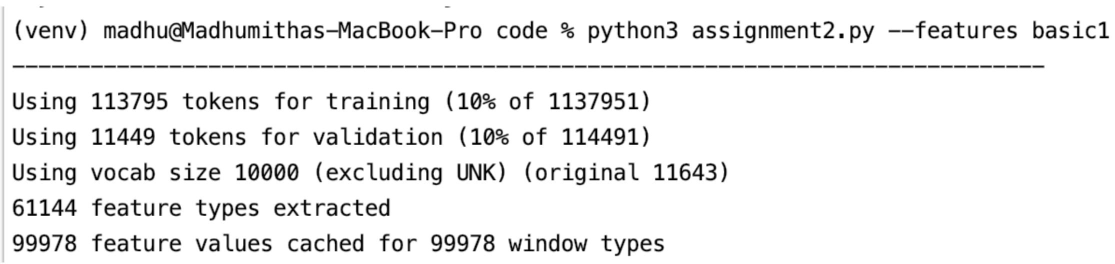
(2+2+2+2+4+2+2+2 (+10)=18 (+10)points)

Code is attached in the submission zip file.

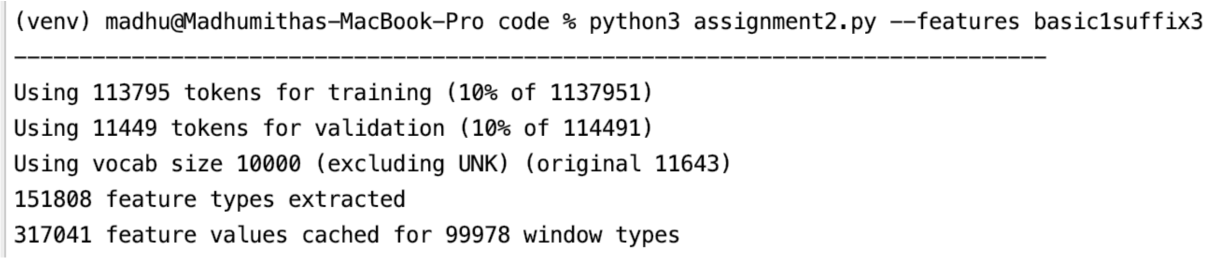
1. I implemented the basic_features1_suffix3 function which extracts features in basic_features1 plus suffixes of length up to 3 of window[-2]. Code can be found in util.py.

```
def basic_features1_suffix3(window):
    features=dict()
    for i+1 in range(3):
        if i+1<len(window[-2]):
            features[’c-1s’+str(-i+1)+’=%s^w=%s’%(window[-2][-i+1:],window[-1])]=True
    features.update(basic_features1(window))
    return features
```

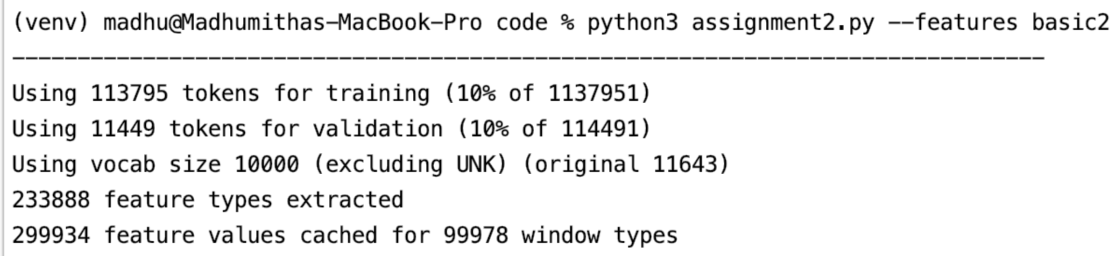
2. Using 10% of the training data and 10% of the validation data, I found the number of feature types extracted using basic features1, basic features1 suffix3, and basic features2.



```
(venv) madhu@Madhumithas-MacBook-Pro code % python3 assignment2.py --features basic1
--------------------------------------------------------------------------------
Using 113795 tokens for training (10% of 1137951)
Using 11449 tokens for validation (10% of 114491)
Using vocab size 10000 (excluding UNK) (original 11643)
61144 feature types extracted
99978 feature values cached for 99978 window types
```



```
(venv) madhu@Madhumithas-MacBook-Pro code % python3 assignment2.py --features basic1suffix3
--------------------------------------------------------------------------------
Using 113795 tokens for training (10% of 1137951)
Using 11449 tokens for validation (10% of 114491)
Using vocab size 10000 (excluding UNK) (original 11643)
151808 feature types extracted
317041 feature values cached for 99978 window types
```



```
(venv) madhu@Madhumithas-MacBook-Pro code % python3 assignment2.py --features basic2
--------------------------------------------------------------------------------
Using 113795 tokens for training (10% of 1137951)
Using 11449 tokens for validation (10% of 114491)
Using vocab size 10000 (excluding UNK) (original 11643)
233888 feature types extracted
299934 feature values cached for 99978 window types
```

3. I implemented the softmax function in util.py. The function gets a vector and returns the softmax transformation of the vector. This function prevents the exponential from exploding by setting $\mathrm{v}_{\mathrm{max}} = \mathrm{max}_{\mathrm{k}}$ v_{k}. See code below for my implementation of the softmax function in util.py.

```
def softmax(v):
   v_max = np.max(v)
   return np.exp(v-v_max) / np.sum(np.exp(v-v_max))
```

4. I implemented the compute_prob function in util.py. In this function, for every y that follows x, we get features for y, compute the probability distribution, get their associated weight values and sum them up and then pass them through softmax layer. I utilized two caches, self.fcache[window] and self.x2ys[tuple(x)] to speed up computation. See code below for my implementation in util.py.

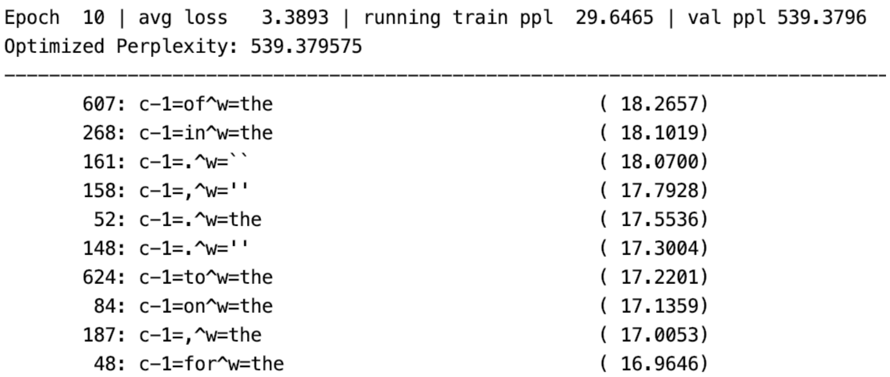
```
def compute_probs(self, x):
    q_=np.zeros(len(self.token_to_idx))
    for i in list(self.x2ys[tuple(x)].keys()):
        for index in self.fcache[tuple(x+[i])]:
           q_[self.token_to_idx[i]]+=self.w[index]
    return softmax(q_)
```

5. I implemented the gradient update in do_epoch in util.py. I had to iterate over all the features in x and whenever the current happens to have that feature subtract the q of the word in x and 1 from the weight. Otherwise, I just subtracted the q of the word in x from the y. See code below for my implementation in util.py.

```
def do_epoch(self, corpus, val_corpus):
    positions = list(range(WINDOW - 1, len(corpus)))
    random.shuffle(positions)
    total_loss = 0.0 # - sum_i ln q(y_i|x_i)

    for ex_num, position in enumerate(positions):
        x = corpus[position-WINDOW+1:position]
        y = corpus[position]
        q = self.compute_probs(x)
        sum=0
        for i in self.x2ys[tuple(x)]:
            sum+=q[self.token_to_idx[i]]
        for j in self.fcache[tuple(x)+tuple([y])]:
            self.w[j]-=np.multiply(self.lr,(sum-1))
        total_loss -= math.log(q[self.token_to_idx[y]])
       if (ex_num + 1) % self.check_interval == 0:
            print('%d/%d examples, avg loss %g' % (ex_num + 1, len(positions),
                ↪ total_loss / (ex_num + 1)))
     return total_loss / len(positions)
```

6. Through implementing the gradient update, I was able to reduce the training loss relatively fast for 10 percent of the data. I tried learning rates of 0.1, 0.5, 1, 2, 4, and 8 with basic_features1 and my findings provided me with the best validation perplexity for these. They can be found in the q1-6.txt file attached in my submissions.
Learning Rate: 0.1 - Avg loss: 6.7215 - Best Val Perplexity: 1195.8542
Learning Rate: 0.5 - Avg loss: 4.4768 - Best Val Perplexity: 589.0955
Learning Rate: 1 - Avg loss: 3.5263 - Best Val Perplexity: 515.221534
Learning Rate: 2 - Avg loss: 3.3893 - Best Val Perplexity: 539.379575
Learning Rate: 4 - Avg loss: 3.4403 - Best Val Perplexity: 557.570889
Learning Rate: 8 - Avg loss: 5.9984 - Best Val Perplexity: 1649.826260

7. For my best model, the top-10 feature types that have been assigned the highest weight are:



```
Epoch  10 | avg loss   3.3893 | running train ppl  29.6465 | val ppl 539.3796
Optimized Perplexity: 539.379575
--------------------------------------------------------------------------------
       607: c-1=of^w=the                          ( 18.2657)
       268: c-1=in^w=the                          ( 18.1019)
       161: c-1=.^w=``                            ( 18.0700)
       158: c-1=,^w=''                            ( 17.7928)
        52: c-1=.^w=the                           ( 17.5536)
       148: c-1=.^w=''                            ( 17.3004)
       624: c-1=to^w=the                          ( 17.2201)
        84: c-1=on^w=the                          ( 17.1359)
       187: c-1=,^w=the                           ( 17.0053)
        48: c-1=for^w=the                         ( 16.9646)
```

8. Standard deviation: 4.9281
   Mean: 143.6793

Problem 2: Feedforward Neural Language Model

(2 + 2 + 2 + 2 + 2 + 2 = 12 points)

Code is attached in the submission zip file.

1. I define self.FF with correct specifications, specifying the input dimension, hidden state dimension, output dimension, and number of layers. Code can be found in assignment2_nlm.py.

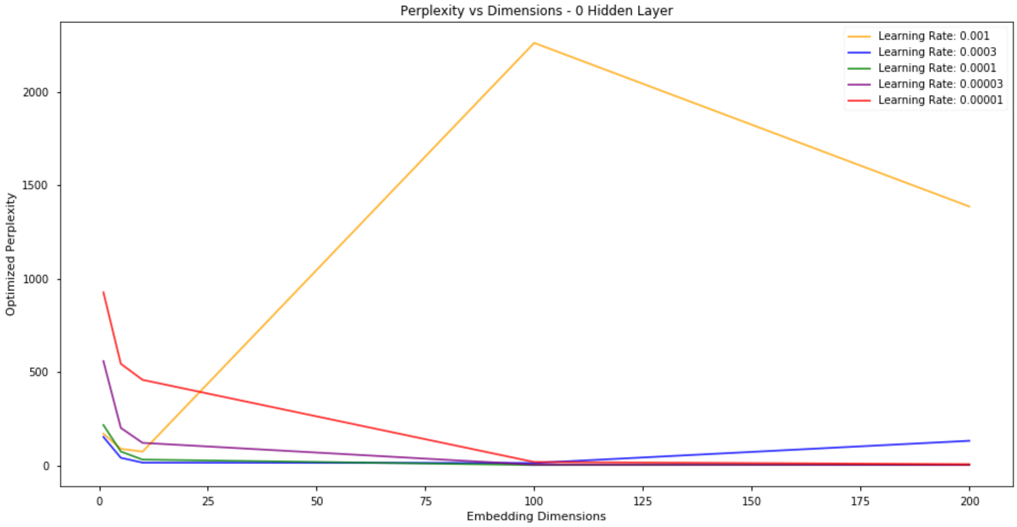
```
 self.FF = FF(np.multiply(wdim,nhis), wdim, self.V, nlayers)
```

2. I implemented (batch-version) forward in FFLM. Code can be found in assignment2_nlm.py.

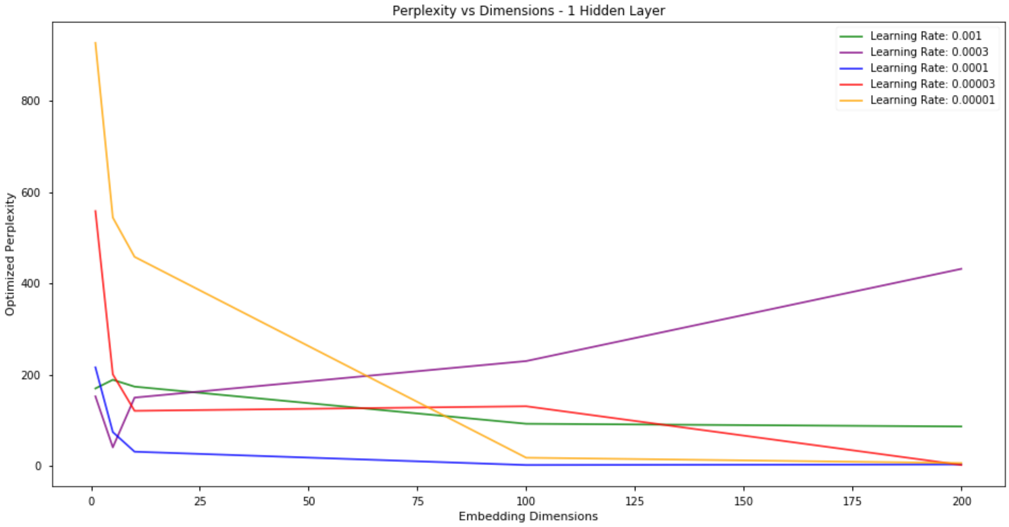
```
def forward(self, X, Y, mean=True): # X (B x nhis), Y (B)
    # TODO: calculate logits (B x V) s.t.
    # softmax(logits[i,:]) = distribution p(:|X[i]) under the model.

    # Q2.2
    embeds = self.E(X).view((-1,np.multiply(self.E(X).shape[-1], self.nhis)))
    logits = self.FF.forward(embeds)

    loss = self.mean_ce(logits, Y) if mean else self.sum_ce(logits, Y)
    return loss
```

   My forward function gets the embeddings for X and reshapes them using view, so that they are the correct shape for the feedforward layer. I find logits by passing the embeddings through my network.

3. In the following graph, I report the optimized perplexities at each dimension. I trained a linear layer using 3 previous words with batch size 16 up to 10 epochs. I varied wdim = hdim across 1, 5, 10, 100, 200 and for each choice optimized over learning rates of 0.00001, 0.00003, 0.0001, 0.0003, 0.001. Nlayers were set at 0.



4. In the following graph, I report the optimized perplexities at each dimension. I trained a linear layer using 3 previous words with batch size 16 up to 10 epochs. I varied wdim = hdim across 1, 5, 10, 100, 200 and for each choice optimized over learning rates of 0.00001, 0.00003, 0.0001, 0.0003, 0.001. Nlayers were set at 1.



5. One hypothesis is that it takes more updates for bigger models to converge, but when they do they can achieve a smaller training loss. By running the linear and nonlinear models to convergence and setting epochs to 1000 and wdim and hdim to 30, my findings reject this hypothesis.

   Here is a TLDR of my output files q2-5_n0.txt and q2-5_n1.txt.

   These are my optimized perplexities when nlayers=0 at different learning rates. I also included at which epoch the program terminated at.

   Optimized Perplexity: 1.623937 when LR = 0.00001 - ends at Epoch 698

Optimized Perplexity: 1.765332 when LR = 0.00003 - ends at Epoch 176
Optimized Perplexity: 2.250153 when LR = 0.0001 - ends at Epoch 46
Optimized Perplexity: 4.781387 when LR = 0.0003 - ends at Epoch 15
Optimized Perplexity: 69.353447 when LR = 0.001 - ends at Epoch 8

These are my optimized perplexities when nlayers=1 at different learning rates. I also included at which epoch the program terminated at.

Optimized Perplexity: 1.972964 when LR = 0.00001 - ends at Epoch 426
Optimized Perplexity: 1.802679 when LR = 0.00003 - ends at Epoch 123
Optimized Perplexity: 1.760574 when LR = 0.0001 - ends at Epoch 37
Optimized Perplexity: 1.749049 when LR = 0.0003 - ends at Epoch 14
Optimized Perplexity: 1.737731 when LR = 0.001 - ends at Epoch 10

From my findings, the optimized perplexity increases significantly as the learning rates increased when nlayers was equal to 0. However, when nlayers was set to 1, the optimized perplexity decreased (not as significantly) as the learning rates increased.

In addition, if you look at my output files q2-5_n0.txt, you can see that the training perplexity increases as the learning rates increase when nlayers is 0. Meanwhile, if you look at my output files q2-5_n1.txt, you can see that the training perplexity always decreases as the learning rates increase when nlayers is 1. Looking at the data, the bigger model of the two is when nlayers is 1, so thus it refutes the hypothesis.

6. When looking at the nearest neighbors of trained word embeddings in cosine similarity which are printed out by the program, I noticed that these neighbors make sense only sometimes.

   For instance, with the word, power, neighbors like disguise, prepared, votes, leadership, juan, continue, effort, need, bush and island appear. While some of these words make sense like leadership, votes, and effort, other words like island make no sense.

   Some of these unigrams may occur in similar context or co-occur with each other. However, there is also some randomness which may attribute to certain unigrams being associated with others.