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

Assignment 2

Name: Twisha Naik, *NetID:* tn268 *Students discussed with:*

Problem 1: Log-Linear Models

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

1. The code for *basic features1 suffix3* which extracts features in "basic features1" plus suffixes of length up to 3 of window[-2].

```
def basic_features1_suffix3(window):
    # TODO: Implement
    word_length = len(window[-2])
    features_dict = {}

    # basic_features1
    features_dict['c-1=%s^w=%s' % (window[-2], window[-1])] = True

    # based on the length of the word, add the keys in the dictionary
    if word_length >=1:
        features_dict['c-1s1=%s^w=%s' % (window[-2][-1:], window[-1])] = True
    if word_length >=2:
        features_dict['c-1s2=%s^w=%s' % (window[-2][-2:], window[-1])] = True
    if word_length >=3:
        features_dict['c-1s3=%s^w=%s' % (window[-2][-3:], window[-1])] = True

    return features_dict
```

Thus, this function is able to extract bigram features and features with suffixes up to length 3 of precious word.

```
In [2]: window = ["test", "string"]

In [3]: basic_features1_suffix3(window)
Out[3]:
{'c-1=test^w=string': True,
 'c-1s1=t^w=string': True,
 'c-1s2=st^w=string': True,
 'c-1s3=est^w=string': True}
```

Figure 1: Feature extractor result

2. Using 10% of the training data and 10% of the validation data (default), the number of feature types reported are as follows:

   - basic features1 = 61144
   - basic features1 suffix3 = 171688
   - basic features2 = 233888

   NOTE: *basic features 1* considers the bigrams in the model whereas *basic features 1 suffix3* additionally considers the suffixes of length 3 of the first word in bigram. Thus the number of features are almost 3 times. *basic features 2* considers bigram, skip gram and trigram. This feature extractor takes the full words in a window size of three. Hence there is less chance of repetition, making the number of features too large.

3. Softmax Function
   Here, the maximum value of the vector is subtracted from each element of the vector. Using this way of calculating softmax, we prevent the exponential from exploding because now we are dealing with just negative powers of e.

```
def softmax(v):
    v_max = np.amax(v)
    v_updated = v - v_max
    v_exp = np.exp(v_updated)
    v_sum = np.sum(v_exp)
    softmax_vector = v_exp/v_sum

    return softmax_vector
```

The result of softmax function just depends on the relative difference between the given input values. Subtracting the same value from each element will not change the relative difference between the elements keeping the value of softmax value constant. Thus, the answer for softmax([1,2,3]) = softmax([2,3,4]) and will also remain the same if we subtract the max value from each element i.e. softmax([-2, -1, 0]) in this case.

4. Implementation of *compute probs* Using the cached values, this function computes the probabilty distribution over the vocabulary for the next word.

```
    def compute_probs(self, x):
        # TODO: Calculate NumPy score vector q_ s.t. q_[ind(y)] = w' phi(x, y).

        # Finding the possible values of y given x using the cache x2ys
        _ys = self.x2ys[tuple(x)]
        q_ = np.zeros(len(self.token_to_idx))

      # For each y, construct the window
        for _y in _ys:
            current_window = x.copy()
            current_window.append(_y)

            # Using fcache, find the non-zero feature indices for the current window
            inds = self.fcache[tuple(current_window)]

            weight = 0
            for _ind in inds:
                weight += self.w[_ind]

            q_[self.token_to_idx[_y]] = weight

        return softmax(q_)
```

5. Implementation of the gradient update in *do epoch*:
   Here, we will use gradient ascent to achieve convergence.

```
    # TODO: Implement the gradient update w = w - lr * grad_w J(w).
    # The update must be sparse. Do not work with the whole vector w. Use caches self.
    fcache and self.x2ys.

    window = x.copy()
    window.append(y)
    features_k = self.fcache[tuple(window)]

    for k in features_k:
        grad = 0
        _ys = self.x2ys[tuple(x)]
        for _y in _ys:
            current_window = x.copy()
            current_window.append(_y)
            inds = self.fcache[tuple(current_window)]

            for idx in inds:
                if idx == k:
                    grad += q[self.token_to_idx[_y]]
                else:
                    self.w[idx] -= self.lr * q[self.token_to_idx[_y]]

        self.w[k] += self.lr * (1 - grad)
```

6. The graph for different training and validation perplexity for learning rates = [0.1, 0.5, 1, 2, 4, 8]:

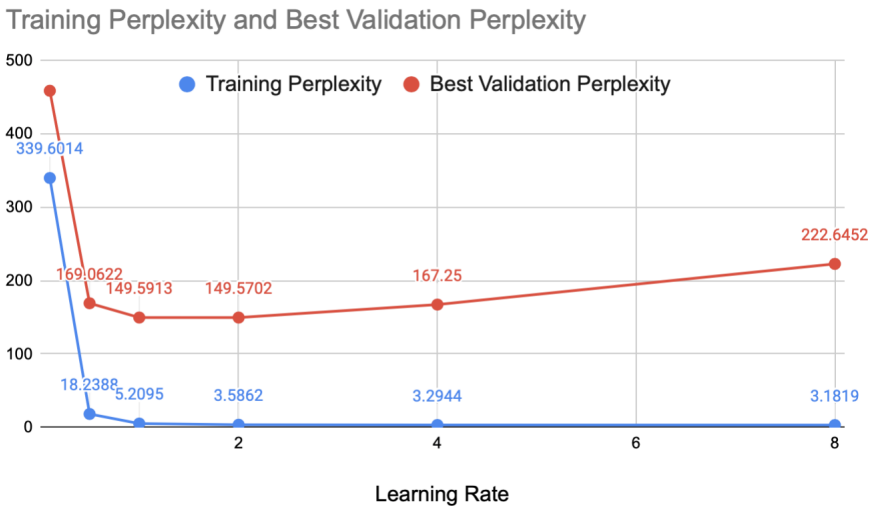

Figure 2: Perplexity w.r.t. varying alpha values for Laplace smoothing

7. For learning rate = 2, we get the least validation perplexity and the top-10 feature types that have been assigned the highest weight are:

Index	Feature	Weight
1030	c-1=prime^w=minister	16.2011
1474	c-1=according^w=to	15.7474
690	c-1=barack^w=obama	15.385
528	c-1=laurent^w=gbagbo	15.385
2213	c-1=end^w=of	15.272
4546	c-1=ahead^w=of	15.2557
3167	c-1=part^w=of	14.8111
2092	c-1=such^w=as	14.767
768	c-1=able^w=to	14.721
2054	c-1=members^w=of	14.622

Table 1: Feature indices, feature values and feature names

The weights of the features signify the importance of that features. Here, the features are the bigrams and the bigrams with highest weights are the frequent ones. If we look at the training data, these would be the features with the highest counts.

8. 10 random seeds

Random Seed	Validation Perplexity
7	148.3557
9	148.2889
25	148.5418
42	149.5702
121	148.6029
250	149.3393
500	147.9554
1001	158.6683
8716	158.936
9881	158.9063

Table 2: Results with Random seed

- Mean = 151.71648
- Standard Deviation = 4.936959565

9. **(Bonus)**

- **Feature extractor**:
  All the experiments till now were performed using the basic feature extractor. The new feature extractor coded has some additional features in comparison to the one used earlier. The trigram feature space was too large. The best validation perplexity for trigram features at learning rate 0.5 in 10 epochs was 306.245261 which is much higher as compared to other models.
  Hence, the ***basic features1 suffix3*** is used for best results.
- **Learning rate:**
  Keeping the seed fixed = 42, different learning rates were tried on 10% of the train and validation data.

Learning Rate	0.175	0.25	0.5	1	2	4
Validation Perplexity	82.2845	79.0478	79.500306	82.032705	113.575197	440.2613

Table 3: Perplexity w.r.t. learning rate

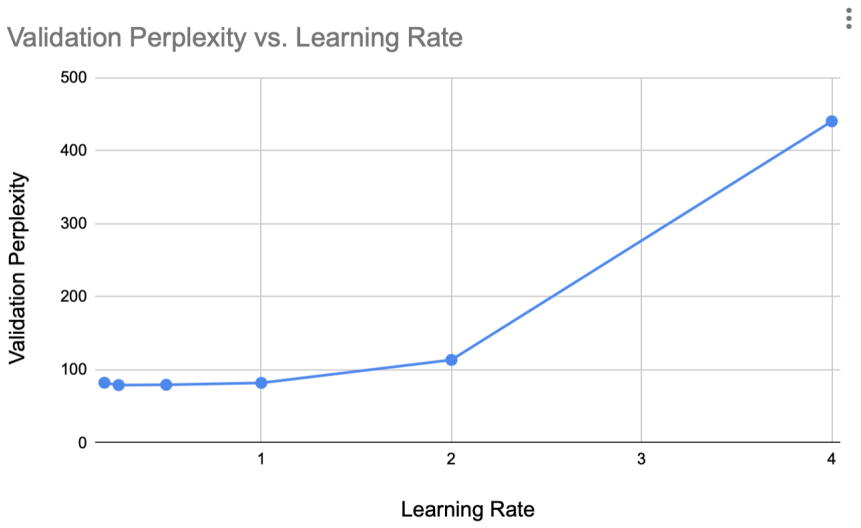

Figure 3: Perplexity w.r.t. learning rate

- **Seed:**
  The variation with respect to seed is quite random and does not follow any specific pattern. I tested a few seeds which gave good results in the previous model, 500 and 9, but that did not give good results here (42 worked the best).
- **Features with highest weights:**
  The features with highest weights in 10 epochs are as mentioned in the table below. As it can observed from the table 4, the top most features are the ones with last one letter in suffix and the highest weight in the model is lower than the previous feature space.

The results for 100% training dataset and validation dataset are mentioned in the table 5.

Command to replicate the best result: python3 assignment2.py –epochs 1000 –lr 0.25 –features basic1suffix3 –train_fraction 1 –val_fraction 1

(Collaborated with two of my classmates Keya Desai and Prakruti Joshi for divided computation)

Index	Feature	Weight
2140	c-1s1=s^w=of	7.3527
133	c-1s1=s^w=,	7.3414
119	c-1s1=s^w=.	7.2449
1007	c-1s1=s^w=and	7.1561
35	c-1s1=e^w=of	7.0663
4135	c-1s1=a^w=,	7.019
705	c-1s1=s^w=that	7.0179
13005	c-1s1=6^w=,	7.0157
394	c-1s1=s^w=in	7.0145
96	c-1s1=e^w=to	6.9599

Table 4: Features with highest weights

	Epoch				
lr	**1**	**2**	**3**	**4**	**5**
1	243.0453	252.5182	102.8083	**101.1708**	103.2816
0.25	122.7985	106.4633	**99.1144**		

Table 5: Results for 100% training data

Problem 2: Feedforward Neural Language Model ((2 + 2 + 2 + 2 + 2 + 2 = 12 points))

1. Initialization of self.FF with correct dimensions as follows:

   - input dimension = length of history elements to be considered * dimension of word embedding
   - hidden state dimension = hidden dimensions as defined
   - output dimension = vocabulary size
   - number of layers = number of layers defined

   Thus, the code for the same looks like:

```
    self.FF = FF(nhis*wdim, hdim, self.V, nlayers)

```

2. Implementation of (batch-version) forward in FFLM

```
    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.

        word_embedding = self.E(X).view(self.batch_size, -1)
        logits = self.FF.forward(word_embedding)

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

```

3. Exploration of the impact of the number of parameters in a linear model.
   When nlayers=0, we have no hidden layer and input is mapped to the output directly thus making it a linear model.

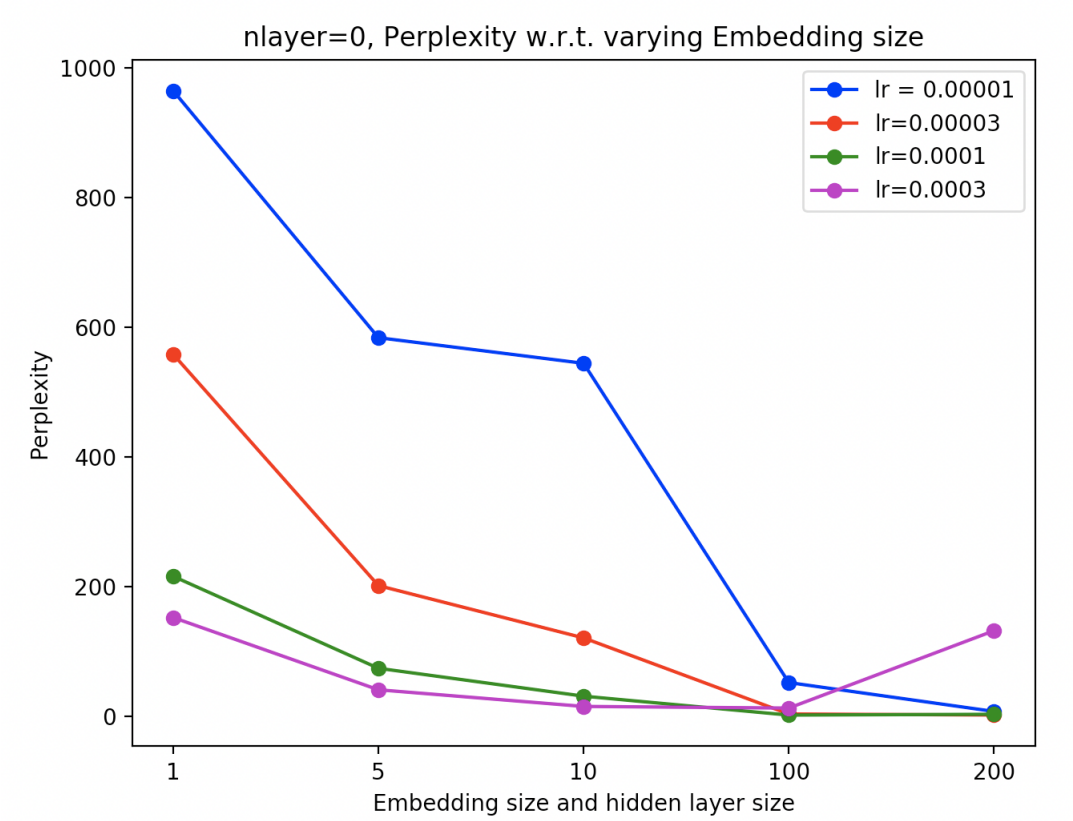

Figure 4: Perplexity w.r.t. varying Embedding size
The perplexity of the model increases largely when learning rate is 0.001 and hence it is plotted separately.

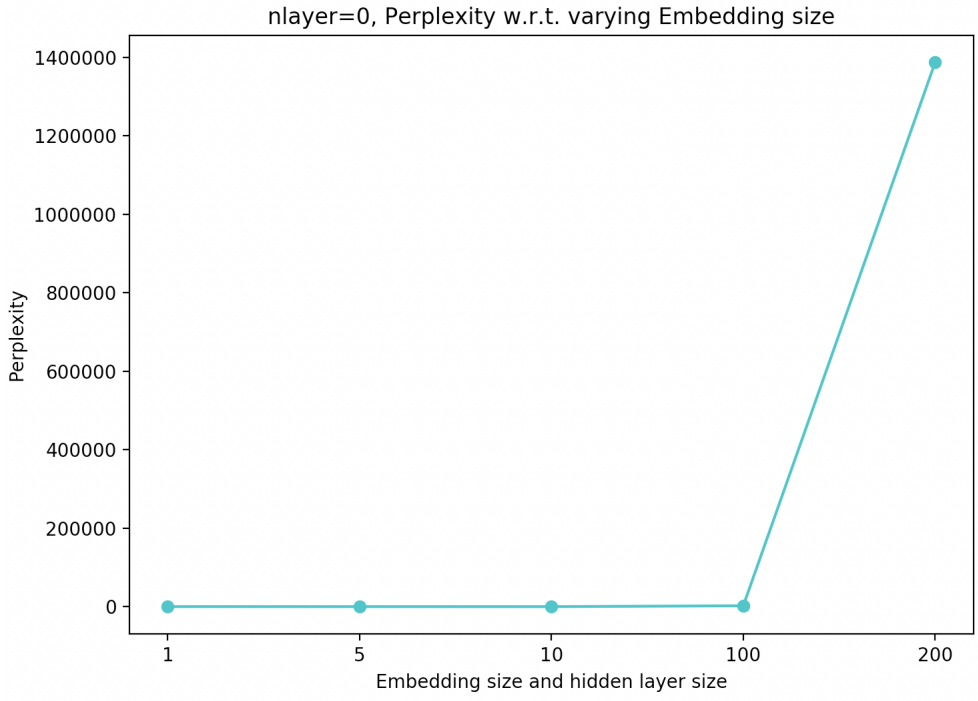

Figure 5: Perplexity w.r.t. varying Embedding size for learning rate = 0.001

4. Exploration of the impact of the number of parameters in a non-linear model.
When nlayers=1, we have one hidden layer with a non-linear activation function making the model non-linear.

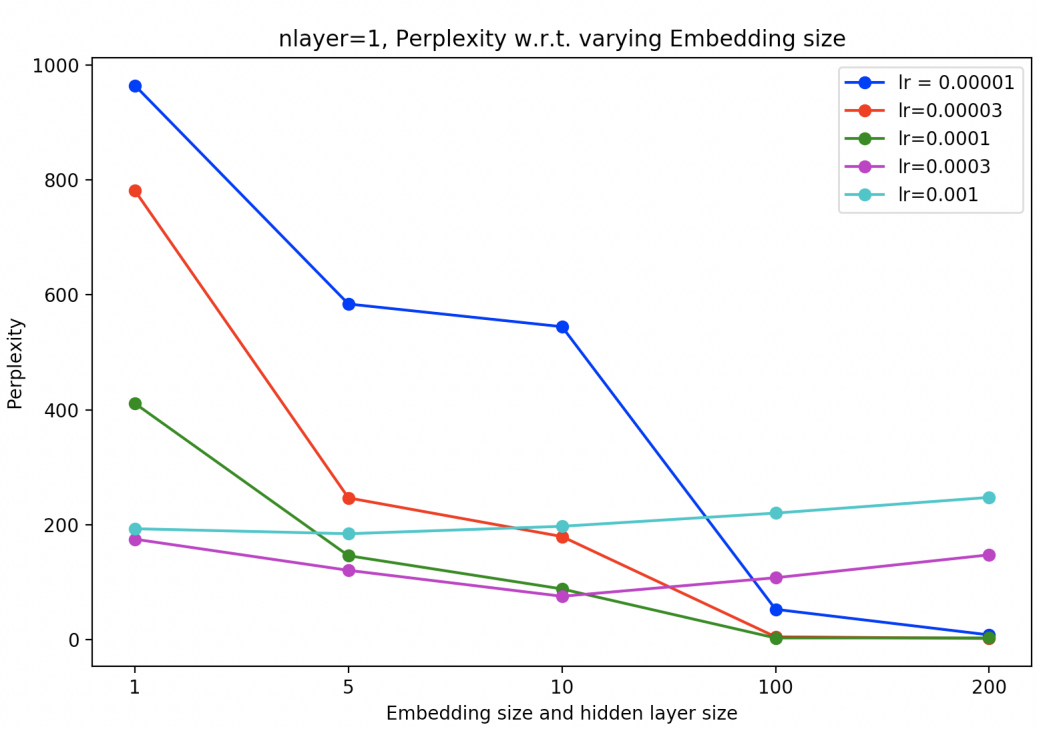

Figure 6: Perplexity w.r.t. varying alpha values for Laplace smoothing

5. Prove or disprove: "One hypothesis is that it takes more updates for bigger models to converge, but when they do they can achieve a smaller training loss."

Model type	Epochs	Loss	Optimised Perplexity
Linear	176	0.6049	1.765332
Non-Linear	123	1.0871	2.738181

Table 6: Best model results

As seen from the values, the linear model takes longer to converge and has better results as compared to the model with a hidden layer.

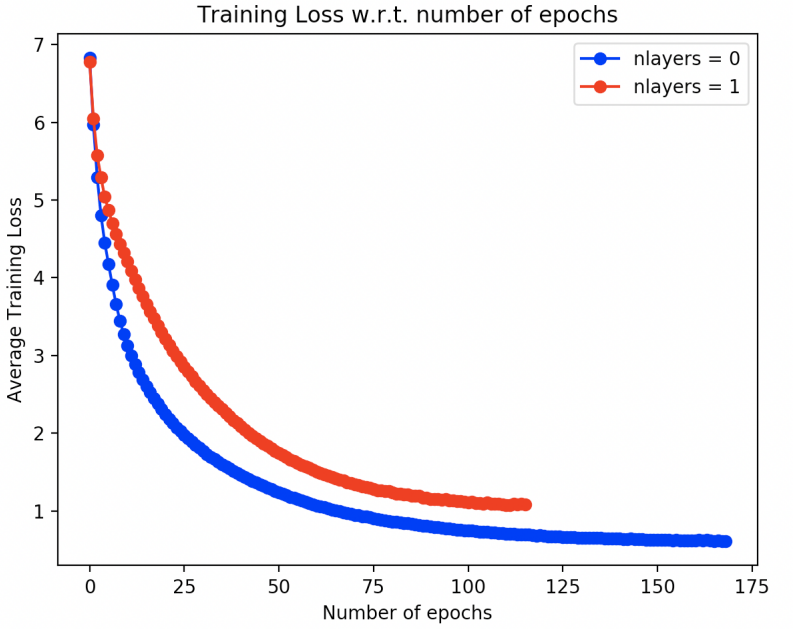

Figure 7: Perplexity w.r.t. varying alpha values for Laplace smoothing

These results clearly "REFUTES" the proposed hypothesis.

6. **Nearest neighbors analysis**
   Keeping the model specifications as follows:
   Learning rate = 0.00003
   Embedding size = 100

   Linear model (nlayers = 0)

```
regulatory: new close winning halter come any recovery example immediate legislation
squad: north chance stoke efforts want 27 following transportation guys led
finished: led authorities many central advanced miss seen begun taken these
out: more members part three really chile joint minister position agenda
carbon: cuba silver this title complex afp rogers china denotes now
met: best likely hope ghanaian events villa former authorities egypt protection
commentary: 1996 season lieberman santiago regulations tournament afp stronger owen sure
event: run draw milner tough seen too showed take reconciliation growing
tackle: like airport game 6-3 schumacher experience ghana : need tuesday
carlos: major cooperation talks winning between serious counterpart house increase being
```

   Non-linear model (nlayers = 1)

```
'm: session relief previously opened persuade strong work brought growing already
into: earned miss shape article released cheeseburgers involved through won for
medal: site ohno strategy draw iran agreed rooney out quarterfinals john
our: formal spanish national biggest financial from u.n. step available dinner
seeding: millions draw example sports michael faltered track market slip appeared
largest: consensus fourth efforts howard everything farmers oil hamelin resources approach
focus: tournament talks especially vacancies winner adviser war announcement months questions
enough: crisis conference 7-5 pollution semifinals juan 17 6-3 field (
our: formal spanish national biggest financial from u.n. step available dinner
island: proposal working medals included injuries campaign iran ca palestinian continue
```

   As seen from the results, the nearest words does not always make sense. There are few neighbors that make sense like "carbon" and "silver" both of which are natural elements. Other good example is "medal", "quarterfinals", "draw" which are all used in similar context.
   The similarity in the embeddings here does not give us the synonyms but gives us the contextual information (That too not very accurate.)
   The reason might be the lack of full context and the data itself. We are just providing the trigrams in the input and then learning the embeddings. The models such as skip-gram models take into consideration both the previous and next words to learn the full context and hence it generates better embeddings.