CS 458 - Natural Language Processing

Assignment 01

Batool Ali Akbar - ba07612

1. (20 points) Implementing an N-Gram Language Model and Testing Perplexity
   (a) (10 points) Python Script

Loading and Preprocessing of Data

Solution:

```
def preprocessing(file_name):
  corpus = open(file_name, "r")
  corpus = corpus.read()
  corpus = corpus.lower()
  corpus = (re.findall(r'\w+', corpus))
  return corpus
```

The above code does a word-based tokenization of the corpus.

Generating N-grams and Calculating Probabilities

Solution:

```
def n_gram(corpus, n):
  ngram_dict = {}
  n_1gram_dict = {}

  for i in range(len(corpus) - n + 1):
    ngram = tuple(corpus[i:i+n])
    if ngram in ngram_dict:
      ngram_dict[ngram] += 1
    else:
      ngram_dict[ngram] = 1

    n_1gram = ngram[:-1]
    if n_1gram in n_1gram_dict:
      n_1gram_dict[n_1gram] += 1
    else:
      n_1gram_dict[n_1gram] = 1

  return (ngram_dict, n_1gram_dict)


def probabilities(ngram, n_1gram, vocab):
  p_dict = {}
  vocab_size = len(vocab)

  for word in ngram:
    n_gram_count = ngram[word]
    n_1 = word[:-1]
    n_1_count = n_1gram[n_1]
    prob = (n_gram_count+1)/(n_1_count+vocab_size)
    p_dict[word] = prob

  return p_dict
```

The above code creates two dictionaries; one to store the ngrams and their count and one to store the n-1grams or the prefixes and their count. These two dictionaries are then used by the probabilities function to calculate the probability of each ngram using add-1 smoothing.

Computing Perplexity

Solution:

```
def perplexity(nprob_dict, n_1dict, corpus_test, n, vocab_train):
  p = 1
  v = len(vocab_train)
  N = len(corpus_test) - n + 1

  for i in range(len(corpus_test)-n+1):
    ngram = tuple(corpus_test[i:i+n])
    if ngram in nprob_dict:
      p = p *  nprob_dict[ngram]
    else:
      n_1gram = ngram[:-1]
      if n_1gram in n_1dict:
        p = p * (1 / (n_1dict[n_1gram] + v))
      else:
        p = p * (1/v)

  p = 1/p
  p = p ** (1/N)
  return p

```

The above code calculates the perplexity of the given dataset by looking for the probability of each ngram of the dataset in the probabilities dictionary created using the training set. If the ngram is not found in the probabilities dictionary, the probability is recalculated using add-1 smoothing. All these probabilities are then multiplied, reciprocated and their root with respect to the total number of ngrams is taken.

(b) (20 points) Report

(a) Approach to implementing N-gram Model

Solution:

The ngram model works as follows:

1. the model loops over the entire corpus, making n sized groups and storing the count for each group
2. the count for the n-1 group i.e. the prefix is also stored in a separate dictionary
3. the probabilities for the ngram are calculated using the counts stored in the dictionary with add-1 smoothing and are stored in a new dictionary

(b) Perplexity calculation for different n values

Solution:

Perplexity calculation is done as follows:

1. iterate over the entire corpus and create the ngrams
2. for each ngram, check if it has been assigned a probability in the training set; if so, use the probability, else calculate the probability using add-1 smoothing
3. all these probabilities are then multiplied, reciprocated and their root with respect to the number of n grams is taken

The perplexity values are as follows:

n-gram	Perplexity
Unigram	30.45
Bigram	22.41
Trigram	26.75

(c) How n impacts the performance and perplexity

Solution:

The n values represent the number of words taken into notice while predicting the next word. The higher the value of n, the more context the model tries to capture. For bigger datasets, higher values of n would mean lower perplexity and better performance. However, in the current example, bigram performed the best. This happens because the data set size is quite small which means that the trigrams and all the other n grams where $n > 3$ of the test set might not be in the training set which increases the perplexity. As for the unigram, it does not consider any context, therefore, the perplexity is the highest i.e. performance is the lowest.

2. (40 points) Implementing a Naive Bayes Classifier for Sentiment Analysis

(a) (30 points) Python Script

Data Loading, Text Preprocessing, Tokenization, and Feature Extraction

Solution:

```
1
2 def extraction(path, n):
3     all_reviews = []
4     d = os.listdir(path)
5
6     for review_f in d[:n]:
7         review_path = os.path.join(path, review_f)
8         review = open(review_path, "r", encoding='utf-8')
9         review = review.read()
10        review = review.lower()
11        review = (re.sub(r'[^a-z\s]', "", review))
12        review = (re.findall(r'\w+', review))
13        all_reviews.append(review)
14
15    return all_reviews
16
17 def count(pos, neg):
18    c = {}
19
20    for p in pos:
21        for w in p:
22            if w in c:
23                c[w][0] += 1
24            else:
25                c[w] = [1, 0]
26
27    for n in neg:
28        for w in n:
29            if w in c:
30                c[w][1] += 1
31            else:
32                c[w] = [0, 1]
33
```

```
    return c

```

The above code first tokenizes each review into individual words and then appends it into a list. The count() function stores the frequency of each word in a positive and a negaative context in a dictionary where the key is the word and the value is a list with the first index representing the count in positive context and the second value representing the count for the negative context.

Implementation of Naive Bayes

Solution:

```

def nb(corpus, l, prior_n, prior_p):
    out = []

    for review in corpus:
        prob_pos = prior_p
        prob_neg = prior_n

        for word in review:
            if word in l:
                prob_pos *= (l[word][0])
                prob_neg *= (l[word][1])

        if prob_pos > prob_neg:
            out.append('pos')
        else:
            out.append('neg')

    return out

```

The above coded uses the formula for the Naive Bayes approach to calculate the class of the input. The corpus is the test dataset and l is the likelihood dictionary where the key is the word and the value is list where the first index represents the likelihood in a positive context and the second represents the likelihood in a negative context. The formula uses the prior probabilities and the likelihood to predict the class of each review.

Training and Testing on IMDB Dataset

Solution:

```

def likelihood(total_pos, total_neg, count_dict):
    l = {}
    v = len(count_dict)

    for i in count_dict:
        l_p = (count_dict[i][0] + 1) / (total_pos + v)
        l_n = (count_dict[i][1] + 1) / (total_neg + v)
        l[i] = [l_p, l_n]

    return l

```

The above code uses the count_dict created in count() to calculate the likelihood of each word in a positive and negative context with add-1 smoothing applied. This creates a dictionary where the key is the word and the value is a list of length 2 where the first index is the likelihood in the positive context and the second index is the likelihood in the negative context. This is the training part. The testing part is to use these likelihoods to predict the class of some test sample which is done in the function nb().

Evaluation Metrics

Solution:

```
def evaluate(p, n):
    tp = p.count("pos")
    tn = n.count("neg")
    fp = n.count("pos")
    fn = p.count("neg")

    a = (tp+tn) / (tp+tn+fp+fn)
    a = a * 100

    recall = tp / (tp+fn)

    precision = tp / (tp+fp)

    f1 = (2*precision*recall) / (precision+recall)

    return (a, recall, precision, f1, tp, tn, fp, fn)

```

The above code uses the predicted values to calculate a confusion matrix, recall, precision, accuracy and the f1 score which is used to evaluate the overall performance of the model.

(b) (10 points) Report

Performance

Solution:

46 (tp)	22 (fp)	0.67 (Precision)
54 (fn)	78 (tn)	
0.46 (Recall)		62.0% (Accuracy)

- True Positive = 46
- False Negative = 54
- True Negative = 78
- False Positive = 22
- Precision = $\frac{tp}{tp+fp} = 0.67$
- Recall = $\frac{tp}{tp+fn} = 0.46$
- Accuracy = $\frac{tp+tn}{tp+fp+tn+fn} = 0.62 = 62\%$
- F-1 Score = $\frac{2PR}{P+R} = 0.54$

Challenges and Experience

Solution:

The given dataset had many files. Since we are to work with only a specific number of files, the process of choosing the files highly impacts the model accuracy. For example, every chosen set of files may have a different set of vocabulary which might be biased towards a specific class. Similarly, the tokenization techniques also affect how the model performs. Moreover, the data size was quite big which made it hard to manually check the results.

Improving the Model

Solution:

The model can be improved by improving the tokenization and preprocessing of the data. Techniques like stemming and lemmatization can be used to improve the performance of the model. Similarly, n-grams can be used to capture context and sentiments. Furthermore, better smoothing techniques can be used to improve the performance.

3. (40 points) Implementing an Artificial Neural Network for Sentiment Classification
   (a) (30 points) Python Script

Preprocessing of Text Data

Solution:

```
1
2 def preprocessing(file_name):
3   vocab = {}
4   data = []
5
6   with open(file_name, mode='r', newline='', encoding='utf-8') as file:
7     corpus = csv.reader(file)
8     next(corpus)
9
10    for row in corpus:
11      text = row[0]
12      text = text.lower()
13      text = (re.sub(r'[^a-z0-9\s]', "", text))
14      words = text.split()
15      data.append((words, row[1]))
16      for w in words:
17        if w not in vocab:
18          vocab[w] = len(vocab)
19
20  return (vocab, data)
21
22 def encoding(vocab, data):
23  encoded = []
24  labels = []
25  v = len(vocab)
26
27  for sentence, label in data:
28    t = np.zeros(v)
29    for word in sentence:
30        if word in vocab:
31          t[vocab[word]] = 1
32    encoded.append(t)
33    labels.append(int(label))
34  labels = np.array(labels)
35  labels = labels.reshape(-1, 1)
36  return (np.array(encoded), labels)
37
```

The above code goes through the csv file row by row. For each row, it tokenizes its first index i.e. the sentence and creates a unique vocabulary out of each word. This vocabulary is stored in the form of a dictionary where the value represents the index. The code then reiterates over the data to generate one hot encoding for each sentence.

Implementation of ANN

Solution:

```
def forward(input, weight, bias):
    z = np.dot(input, weight) + bias
    a = 1 / (1 + np.exp(-z))
    return a

def backward(truth, encoding, ih, ho, w_ih, w_ho, b_h, b_o, l):
    output_error = ho * (1 - ho) * (truth - ho)
    hidden_error = ih * (1 - ih) * (output_error * w_ho.T)

    w_ho_updated = w_ho + (l * ih.T.dot(output_error))
    w_ih_updated = w_ih + (l * encoding.T.dot(hidden_error))

    b_o_updated = b_o + l * np.sum(output_error, axis=0, keepdims=True)
    b_h_updated = b_h + l * np.sum(hidden_error, axis=0, keepdims=True)

    return (w_ho_updated, w_ih_updated, b_h_updated, b_o_updated)

```

The ANN works by first running a forward pass. It applies the summation and the activation function and returns the final value. The backward pass calculates the error in each layer and calculates updated weights.

Training and Testing

Solution:

```
def initialize(input_size, hidden_size, output_size):
    w_ih = np.random.randn(input_size, hidden_size)
    b_h =  np.zeros((1, hidden_size))
    w_ho = np.random.randn(hidden_size, output_size)
    b_o = np.zeros((1, output_size))
    return (w_ih, b_h, w_ho, b_o)

def train(encoded, labels, epoch, learning_rate, w_ih, b_h, w_ho, b_o):
     for _ in range(epoch):
         ih = forward(encoded, w_ih, b_h)
         ho = forward(ih, w_ho, b_o)
         w_ho, w_ih, b_h, b_o = backward(labels, encoded, ih, ho, w_ih, w_ho,
     b_h, b_o, learning_rate)

     return w_ho, w_ih, b_h, b_o

def predict(encoding, w_ih, b_h, w_ho, b_o):
    output_ih = forward(encoding, w_ih, b_h)
    output_final = forward(output_ih, w_ho, b_o)
    return output_final

```

For training the model, the forward and the backward pass is run for a certain number of epoch where the output of one iteration is the input of the other. The final weights, returned after a selected number of epoch, are then used on the test set to predict its output.

Evaluation Metrics

Solution:

```
def evaluate(predicted, truth):
    tp = 0
    tn = 0
    fp = 0
    fn = 0

    for i in range (len(truth)):
        if truth[i] == 1 and predicted[i] >= 0.5:
           tp += 1
        elif truth[i] == 1 and predicted[i] < 0.5:
           fn += 1
        elif truth[i] == 0 and predicted[i] >= 0.5:
           fp += 1
        elif truth[i] == 0 and predicted[i] < 0.5:
           tn += 1

    a = (tp+tn) / (tp+tn+fp+fn)
    a = a * 100

    if (tp + fn) == 0:
       recall = 0
    else:
        recall = tp / (tp+fn)

    if (tp+fp) == 0:
       precision = 0
    else:
        precision = tp / (tp+fp)

    if (precision+recall) == 0:
      f1 = 0
    else:
      f1 = (2*precision*recall) / (precision+recall)

    return (a, recall, precision, f1, tp, tn, fp, fn)

```

The above code compares the truth and the predicted output to calculate the confusion matrix, recall, precision, accuracy and f-1 score which help evaluate the overall performance of the model.

(b) (10 points) Report

Results

Solution:

47 (tp)	29 (fp)	0.61 (Precision)
204 (fn)	220 (tn)	
0.18 (Recall)		53.4% (Accuracy)

- True Positive = 47
- False Negative = 204
- True Negative = 220
- False Positive = 29
- Precision = $\frac{tp}{tp+fp} = 0.61$
- Recall = $\frac{tp}{tp+fn} = 0.18$
- Accuracy = $\frac{tp+tn}{tp+fp+tn+fn} = 0.534 = 53.4\%$
- F-1 Score = $\frac{2PR}{P+R} = 0.28$

Performance

Solution:

The overall performance of the model is not too high. This can be seen by the accuracy which is 53.4%. The low value for recall suggests that only 18% of the actual positive cases were identified. The precision of value suggests that positive predictions are about a 61% correct.

Challenges

Solution:

The implementation of the back propagation was a little hard. Figuring out the dimensions for the matrix multiplication was quite challenging. I had to use $np.sum(output\_error, axis = 0, keepdims = True)$ to ensure that the matrix dimensions are maintained.

Improving the Model

Solution:

The model can be improved by changing the hyper parameters like the learning rates, the weight matrix and the epoch value. Furthermore, a change in the number of neurons of the hidden layer and the number of hidden layers can impact the performance. Lastly, a better activation function like ReLU can be used to improve the performance.