HW3_A20439949_Soutonglang

October 25, 2023

1 CS 585 - Homework 3

Tania Soutonglang A20439949

[1]:
```
import pandas as pd
import numpy as np
import math

import nltk
nltk.download('punkt')
from  nltk import bigrams as nltk_bigrams

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\tsout\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

1.1 Problem 1 - Reading the Data

Read in file "train.tsv" from the Stanford Sentiment Treebank (SST) as shared in the GLUE task.

[2]:
```
df_sst = pd.read_csv("./SST-2/train.tsv",delimiter="\t")
df_sst.head(3)
```

[2]:

	sentence	label
0	hide new secretions from the parental units	0
1	contains no wit , only labored gags	0
2	that loves its characters and communicates som…	1

Next, split your dataset into train, test, and validation datasets with these sizes (Note that 100 is a small size for test and validation datasets; it was selected to simplify this homework): - Validation: 100 rows - Test: 100 rows - Training: All remaining rows

[3]:
```
train_df, test_df = train_test_split(df_sst, test_size = 100/len(df_sst),␣
 ↪random_state = 42)
train_df, val_df = train_test_split(train_df, test_size = 100/len(train_df),␣
 ↪random_state = 42)
```

```
train_df = train_df.reset_index(drop = True)
test_df = test_df.reset_index(drop = True)
val_df = val_df.reset_index(drop = True)

print("Training size: ", len(train_df))
print("Testing size: ", len(test_df))
print("Validation size: ", len(val_df))
```

```
Training size:  67149
Testing size:  100
Validation size:  100
```

Review the column "label" which indicates positive=1 or negative=0 sentiment. What is the prior probability of each class on your training set? Show results in your notebook.

[4]:
```
prior_pos = sum(train_df.label == 1) / len(train_df)
print("Positive labels: ", round(prior_pos, 2))

prior_neg = sum(train_df.label == 0) / len(train_df)
print("Negative labels: ", round(prior_neg, 2))
```

```
Positive labels:  0.56
Negative labels:  0.44
```

1.2 Problem 2 - Tokenizing Data

Write a function that takes a sentence as input, represented as a string, and converts it to a tokenized sequence padded by start and end symbols. For example, "hello class" would be converted to: ['<s>', 'hello', 'class', '</s>']

[5]:
```
def tokenize(sentence):
    tokens = []

    tokens = nltk.word_tokenize(sentence)      # tokenize sentence
    tokens = ['<s>'] + tokens + ['</s>']       # add padding

    return tokens
```

Apply your function to all sentences in your training set. Show the tokenization of the first sentence of your training set in your notebook output.

[6]:
```
train_df_tokenized = train_df['sentence'].apply(tokenize)

print(train_df_tokenized[0])
```

```
['<s>', 'stinks', 'so', 'badly', 'of', 'hard-sell', 'image-mongering', 'you',
'</s>']
```

What is the vocabulary size of your training set? Include your start and end symbol in your vocabulary. Show your result in your notebook.

[7]:
```
# list all words in the document
temp = []
for words in train_df_tokenized:
    temp.extend(words)

# keep only the unique words
vocab = np.unique(temp)

print("Vocabulary size: ", len(vocab))
print("Start symbol: ", vocab[0])
print("End symbol: ", vocab[len(vocab) - 1])
```

```
Vocabulary size:  14805
Start symbol:  !
End symbol:  élan
```

1.3 Problem 3 - Bigram Counts

Write a function that takes an array of tokenized sequences as input (i.e., a list of lists) and counts bigram frequencies in that dataset.

Your function should return a two-level dictionary (dictionary of dictionaries) or similar data structure, where the value at index [wi][wj] gives the frequency count of bigram (wi, wj).

For example, this expression would give the counts of the bigram "academy award": `bigram_counts["academy"]["award"]`

[8]:
```
def get_frequency(tokens):
    # create a list of bigrams from the tokens
    bigrams = []
    for sentence in tokens:
        bigrams += list(nltk_bigrams(sentence))

    # find the frequency distribution
    bigramsFreq = nltk.FreqDist(bigrams)

    # create the dictionary
    freqDict = { }

    # iterate through list of bigrams
    for w in range(len(bigrams)):
        # check if key already exists so it doesn't reinit a dict
        if bigrams[w][0] in freqDict.keys():
            freqDict[bigrams[w][0]][bigrams[w][1]] =␣
 ↪bigramsFreq[(bigrams[w][0], bigrams[w][1])]
        else: # make a new dict with a new key
            freqDict[bigrams[w][0]] = {}
            freqDict[bigrams[w][0]][bigrams[w][1]] =␣
 ↪bigramsFreq[(bigrams[w][0], bigrams[w][1])]
```

```
    # return the two-level dictionary
    return freqDict
```

Apply your function to the output of problem 2. You should build one counter that represents all sentences in the training dataset.

[9]:
```
bigram_counts = get_frequency(train_df_tokenized)
```

Use this result to show how many times a sentence starts with “the”. That is, how often do you see the bigram (“<s>”,“the”) in your training set? Show results in your notebook.

[10]:
```
bigram_counts["<s>"]["the"]
```

[10]:
```
4459
```

1.4 Problem 4 - Smoothing

Write a function that implements formula [6.13] in that E-NLP textbook (page 129, 6.2 Smoothing and discounting). That is, write a function that applies smoothing and returns a (negative) log-probability of a word given the previous word in the sequence. It is suggested that you use these parameters: - The current word, wm - The previous word, wm-1 - bigram counts (output of Problem 3) - alpha, a smoothing parameter - vocabulary size

[11]:
```
def smooth(wm, wm_prev, bigram_counts, alpha, vocab_size):
    # get count of wm
    # check if it is a key
    wm_count = 0
    if wm_prev in bigram_counts:
        if wm not in bigram_counts[wm_prev]:
            wm_count = 0
        else:
            wm_count = bigram_counts[wm_prev][wm]


    # get count of wm-1
    # check if it is a key
    wm_prev_count = 0
    if bigram_counts[wm_prev]:
        for w in bigram_counts[wm_prev]:
            wm_prev_count += bigram_counts[wm_prev][w]

    # count(wm-1, wm) + alpha
    num = wm_count + alpha

    # count(wm-1) + alpha * vocabulary_size
    denom = wm_prev_count + alpha * vocab_size

    # calc smoothed probability
```

```
    smoothed = num / denom

    # calc log-prob
    log_prob = math.log(smoothed)
    return log_prob
```

Using this function to show the log probability that the word "academy" will be followed by the word "award". Try this with alpha=0.001 and alpha=0.5 (you should see very different results!). Show your results in your notebook.

[12]:
```
a1 = smooth("award", "academy", bigram_counts, alpha=0.001,␣
 ↪vocab_size=len(vocab))
print('(academy, award)\tlog probability: ', a1)
```

```
(academy, award)        log probability:  -1.024899084055268
```

[13]:
```
a2 = smooth("award", "academy", bigram_counts, alpha=0.5, vocab_size=len(vocab))
print('(academy, award)\tlog probability: ', a2)
```

```
(academy, award)        log probability:  -6.172373816771201
```

1.5 Problem 5 - Sentence Log-Probability

Write a function that returns the log-probability of a sentence which is expected to be a negative number. To do this, assume that the probability of a word in a sequence only depends on the previous word. It is suggested that you use these parameters: - A sentence represented as a single python string - bigram counts (output of Problem 3) - alpha, a smoothing parameter - vocabulary size

[14]:
```
def sentence_log_prob(sentence, bigram_counts, alpha, vocab_size):
    # normalize sentence
    tokens = tokenize(sentence)

    # create a list of bigrams from the tokens
    bigrams = []
    bigrams += list(nltk_bigrams(tokens))

    log_prob = 0
    # find bigram in bigram_counts
    for w in range(len(bigrams)):
        wm_prev = bigrams[w][0]
        wm = bigrams[w][1]

        # check if it is a key
        if wm_prev in bigram_counts:
            if wm not in bigram_counts[wm_prev]:
                log_prob += 0
            else:
```

```
                log_prob += smooth(wm, wm_prev, bigram_counts, alpha,␣
 ↪vocab_size)

    return log_prob
```

Use your function to compute the log probability of these two sentences (Note that the 2nd is not natural English, so it should have a lower (more negative) result that the first): - “this was a really great movie but it was a little too long.” - “long too little a was it but movie great really a was this.”

[15]:
```
s1 = sentence_log_prob("this was a really great movie but it was a little too␣
 ↪long.", bigram_counts, alpha=1.0, vocab_size=len(vocab))
print("\"this was a really great movie but it was a little too long.\" \tlog␣
 ↪probability: ", s1)
```

```
"this was a really great movie but it was a little too long."   log probability:
-80.43567758724953
```

[16]:
```
s2 = sentence_log_prob("long too little a was it but movie great really a was␣
 ↪this.", bigram_counts, alpha=1.0, vocab_size=len(vocab))
print("\"long too little a was it but movie great really a was this.\" \tlog␣
 ↪probability: ", s2)
```

```
"long too little a was it but movie great really a was this."   log probability:
-36.96219434585985
```

1.6 Problem 6 - Tuning Alpha

Next, use your validation set to select a good value for “alpha”.

Apply the function you wrote in Problem 5 to your validation dataset using 3 different values of “alpha”, such as (0.001, 0.01, 0.1). For each value, show the log-likelihood estimate of the validation set. That is, in your notebook show the sum of the log probabilities of all sentences.

[17]:
```
val_prob_a1 = 0
for sentence in val_df['sentence']:
    val_prob_a1 += sentence_log_prob(sentence, bigram_counts, alpha=0.001,␣
 ↪vocab_size=len(vocab))

print("alpha = 0.001\tlog probability: ", val_prob_a1)
```

```
alpha = 0.001   log probability:  -3734.9420028956974
```

[18]:
```
val_prob_a2 = 0
for sentence in val_df['sentence']:
    val_prob_a2 += sentence_log_prob(sentence, bigram_counts, alpha=0.01,␣
 ↪vocab_size=len(vocab))

print("alpha = 0.01\tlog probability: ", val_prob_a2)
```

```
alpha = 0.01    log probability:  -4218.16081868473
```

[19]:
```
val_prob_a3 = 0
for sentence in val_df['sentence']:
    val_prob_a3 += sentence_log_prob(sentence, bigram_counts, alpha=0.1,␣
 ↪vocab_size=len(vocab))

print("alpha = 0.1\tlog probability: ", val_prob_a3)
```

```
alpha = 0.1     log probability:  -5154.761527845872
```

Which alpha gives you the best result? To indicate your selection to the grader, save your selected value to a variable named "selected_alpha".

[20]:
```
selected_alpha = 0.001

print("Having an alpha of {} gave me the best log-likelihood estimate of the␣
 ↪validation set.".format(selected_alpha))
```

```
Having an alpha of 0.001 gave me the best log-likelihood estimate of the
validation set.
```

1.7 Problem 7 - Applying Language Models

In this problem, you will classify your test set of 100 sentences by sentiment, by applying your work from previous problems and modeling the language of both positive and negative sentiment. To do this, you can follow these steps: - Separate your training dataset into positive and negative sentences, and compute vocabulary size and bigram counts for both datasets. - For each of the 100 sentences in your test set: - Compute both a "positive sentiment score" and a "negative sentiment score" using (1) the function you wrote in Problem 5, (2) Bayes rule, and (3) class priors as computed in Problem 1. - Compare these scores to assign a predicted sentiment label to the sentence. - What is the class distribution of your predicted label? That is, how often did your method predict positive sentiment, correctly or incorrectly? How often did it predict negative sentiment? Show results in your notebook. - Compare your predicted label to the true sentiment label. What is the accuracy of this experiment? That is, how often did the true and predicted label match on the test set? Show results in your notebook.

For this problem, you do not need to re-tune alpha for your positive and negative datasets (although it may be a good idea to do so), you can re-use the value selected in Problem 6.

[21]:
```
# separate the training dataset into positive and negative sentences

train_pos = []
train_neg = []

for i in range(len(train_df)):
    if train_df["label"][i] == 1:
        train_pos.append(train_df["sentence"][i])
    elif train_df["label"][i] == 0:
        train_neg.append(train_df["sentence"][i])
```

```
train_pos = pd.Series(train_pos)
train_neg = pd.Series(train_neg)
```

[22]:
```
# compute the vocabulary size and bigram counts of positive dataset

# tokenize
train_pos_tokenized = train_pos.apply(tokenize)

# get vocabulary
temp = []
for words in train_pos_tokenized:
    temp.extend(words)
vocab_pos = np.unique(temp)

# get bigram counts
bigram_counts_pos = get_frequency(train_pos_tokenized)
```

[23]:
```
# compute the vocabulary size and bigram counts of negative dataset

# tokenize
train_neg_tokenized = train_neg.apply(tokenize)

# get vocabulary
temp = []
for words in train_neg_tokenized:
    temp.extend(words)
vocab_neg = np.unique(temp)

# get bigram counts
bigram_counts_neg = get_frequency(train_neg_tokenized)
```

[24]:
```
print("Positive vocabulary size: ", len(vocab_pos))
print("Negative vocabulary size: ", len(vocab_neg))
```

```
Positive vocabulary size:  11527
Negative vocabulary size:  11242
```

[25]:
```
def sentiment_analysis(sentence, vocab_pos, vocab_neg):
    # calculate positive log-likelihood
    log_prob_pos = 0
    log_prob_pos += sentence_log_prob(sentence, bigram_counts_pos,␣
 ↪alpha=selected_alpha, vocab_size=len(vocab_pos))

    # negative log-likelihood
    log_prob_neg = 0
    log_prob_neg += sentence_log_prob(sentence, bigram_counts_neg,␣
 ↪alpha=selected_alpha, vocab_size=len(vocab_neg))
```

```
    # apply Bayes rule
    posterior_pos = prior_pos * log_prob_pos
    posterior_neg = prior_neg * log_prob_neg

    # normalize
    posterior_total = posterior_pos + posterior_neg
    if posterior_total > 0:
        posterior_pos /= posterior_total
        posterior_neg /= posterior_total

    if posterior_pos < posterior_neg:
        return 1
    else:
        return 0
```

[26]:
```
test_pred = []
for sentence in test_df["sentence"]:
    test_pred.append(sentiment_analysis(sentence, vocab_pos, vocab_neg))
test_pred = pd.Series(test_pred)
```

[27]:
```
correct_pos = 0
correct_neg = 0
incorrect_pos = 0
incorrect_neg = 0

for i in range(100):
    if test_df["label"][i] == 1 and test_pred[i] == 1:
        correct_pos += 1
    elif test_df["label"][i] == 0 and test_pred[i] == 1:
        incorrect_pos += 1
    elif test_df["label"][i] == 0 and test_pred[i] == 0:
        correct_neg += 1
    elif test_df["label"][i] == 1 and test_pred[i] == 0:
        incorrect_neg += 1

print("My method counted {} positive sentiments correctly and {} incorrectly.".
 ↪format(correct_pos, incorrect_pos))
print("My method counted {} negative sentiments correctly and {} incorrectly.".
 ↪format(correct_neg, incorrect_neg))
```

```
My method counted 52 positive sentiments correctly and 19 incorrectly.
My method counted 25 negative sentiments correctly and 4 incorrectly.
```

[28]:
```
acc = accuracy_score(test_df["label"], test_pred)
print("Accuracy: ", acc)
```

```
Accuracy:  0.77
```

1.8 Problem 8 - Markov Assumption

- Where in this homework did you apply the Markov assumption?

The Markov assumption was applied in problem 4 where we predicted the likelihood of the word coming after a certain word.

- Imagine you applied the 2nd-order Markov assumption, using trigrams. Do you think your accuracy results would increase or decrease? Why? Or, if you are not sure, give a benefit or drawback of using trigrams for this experiment. (Note: You do not need to rerun this experiment with trigrams to answer this question.)

I'm not sure how trigrams would affect the accuracy because while trigrams would provide more context and improve accuracy for some cases, it would cause a larger dimensionality and won't be able to generalize the data because it would be more specific.

[ ]: