CS 585 - HW 1

• For Problems 1 through 7, please upload to Blackboard: o One Jupyter notebook (.ipynb file) with cell output, showing your work for both datasets. Name this file "HW3_[CWID]_[LastName].ipynb". o A PDF copy of the exact same notebook (same code and same output). o You will lose points if you do not submit both of these files. Please do not zip or compress files before you upload. • For Problem 8: Enter your written answer in Blackboard with your HW submission.

PROBLEM 1 – Reading the data (5 pts) • Read in file "train.tsv" from the Stanford Sentiment Treebank (SST) as shared in the GLUE task. (See section "DATA" above.) • 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): o Validation: 100 rows o Test: 100 rows o Training: All remaining rows. • 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.

```
# !pip install pandas
import pandas as pd

# Read the CSV file with a header row
df = pd.read_csv("/content/train.tsv", delimiter='\t')

# To split the dataset into train, test, and validation sets
validation_set = df[:100]
test_set = df[100:200]
training_set = df[200:]

# Calculate the prior probabilities:
value_counts  = training_set['label'].value_counts()
# Calculate the total count of instances
total_count = value_counts.sum()

# Calculate the prior probabilities
prior_probabilities = value_counts / total_count

# Display the prior probabilities
# print(prior_probabilities)
positive_prior_probability = prior_probabilities[1]
negative_prior_probability = prior_probabilities[0]
print(f"positive_prior_probability: {positive_prior_probability:.6f}")
print(f"negative_prior_probability: {negative_prior_probability:.6f}")
```

```
positive_prior_probability: 0.557968
negative_prior_probability: 0.442032
```

#PROBLEM 2 – Tokenizing data (10 pts) • 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: o ['<_s>', 'hello', 'class', '</_s>'] • 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. • What is the vocabulary size of your training set? Include your start and end symbol in your vocabulary. Show your result in your notebook.

```
# !pip install nltk
import nltk
nltk.download('punkt')  # Download the 'punkt' tokenizer model
from nltk.tokenize import word_tokenize
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
# Function to tokenize and add start/end symbols
def tokenize_and_add_symbols(sentence):
    tokens = nltk.word_tokenize(sentence)
    tokens = ['<_s>'] + tokens + ['</_s>']
    return tokens

# # Apply the function to all sentences in the training set
# training_set['tokenized_sentence'] =
training_set['sentence'].apply(tokenize_and_add_symbols)
# Apply the function to all sentences in the training set and create a
new column
training_set = training_set.copy()
training_set['tokenized_sentence'] =
training_set['sentence'].apply(lambda x: tokenize_and_add_symbols(x))

# Tokenization of the first sentence in the training set
print("Tokenization of the first sentence:")
print(training_set['tokenized_sentence'].iloc[0])

# Calculate the vocabulary size
vocabulary = set()
for tokens in training_set['tokenized_sentence']:
    vocabulary.update(tokens)

# Vocabulary size, including start and end symbols
vocabulary_size = len(vocabulary)
print("Vocabulary Size (including start and end symbols):",
vocabulary_size)
```

```
Tokenization of the first sentence:
['<_s>', 'told', 'in', 'scattered', 'fashion', '</_s>']
Vocabulary Size (including start and end symbols): 14801
```

#PROBLEM 3 – Bigram counts (10 pts) • 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"] • Apply your function to the output of problem 2. You should build one counter that represents all sentences in the training dataset. • 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. PROGRAMMING HINTS: • You can use the function nltk.bigrams to convert a sequence to bigrams, but you are not required to do so. • In python, you can use function "dict.get(key, default)" to return the value "default" if "key" is not present in a dictionary.

```
import nltk

# Function to count bigram frequencies in a list of tokenized
sequences
def count_bigrams(tokenized_sequences):
    bigram_counts = {}

    for tokens in tokenized_sequences:
        # Convert the tokens into bigrams
        bigrams = list(nltk.bigrams(tokens))

        # Count the bigram frequencies
        for bigram in bigrams:
            wi, wj = bigram
            if wi not in bigram_counts:
                bigram_counts[wi] = {}
            bigram_counts[wi][wj] = bigram_counts[wi].setdefault(wj,
0) + 1

    return bigram_counts

# Apply the function to the tokenized sentences in the training set
tokenized_sentences = training_set['tokenized_sentence'].tolist()
bigram_counts = count_bigrams(tokenized_sentences)

# Count how often a sentence starts with "the"
the_count = bigram_counts.get('<_s>', {}).get('the', 0)

# Display the count of "<_s>", "the"
print(f'Count of "<_s>", "the": {the_count}')
```

```
Count of "<_s>", "the": 4450
```

#PROBLEM 4 – Smoothing (20 pts) • 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: o The current word, wm o The previous word, wm-1 o bigram counts (output of Problem 3) o alpha, a smoothing parameter o vocabulary size • 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. PROGRAMMING ALTERNATIVE: If you are familiar

with python classes, you may build a LanguageModel class that is initialized with the above parameters and implements formula [6.13] as a member function.

```
import math

def smoothing_log_probability(wm, wm_1, bigram_counts, alpha,
size_of_vocabulary):
    if wm_1 in bigram_counts and wm in bigram_counts[wm_1]:
        count_wm_1_wm = bigram_counts[wm_1][wm]
    else:
        count_wm_1_wm = 0

    prob = (count_wm_1_wm + alpha) / (sum(bigram_counts.get(wm_1,
{}).values()) + (alpha * size_of_vocabulary))
    return  math.log(prob, 2)


alpha_1 = 0.001
alpha_2 = 0.5

log_prob_1 = smoothing_log_probability("award", "academy",
bigram_counts, alpha_1, vocabulary_size)
log_prob_2 = smoothing_log_probability("award", "academy",
bigram_counts, alpha_2, vocabulary_size)

print(f"Negative log probability (alpha=0.001): {log_prob_1}")
print(f"Negative log probability (alpha=0.5): {log_prob_2}")
```

```
Negative log probability (alpha=0.001): -1.4784787789440406
Negative log probability (alpha=0.5): -8.904464674335902
```

#PROBLEM 5 – Sentence log-probability (10 pts) • 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: o A sentence represented as a single python string o bigram counts (output of Problem 3) o alpha, a smoothing parameter o vocabulary size • 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): o "this was a really great movie but it was a little too long." o "long too little a was it but movie great really a was this."

```
# Function to calculate the log-probability of a sentence using
bigrams and smoothing
def sentence_log_probability(sentence, bigram_counts, alpha,
vocabulary_size):
    # Tokenize the sentence into words
    words = nltk.word_tokenize(sentence)

    # Initialize the log probability
    log_probability = 0.0
```

```
    # Iterate through the words to calculate the log probability
    for i in range(1, len(words)):
        wm_minus_1 = words[i - 1]
        wm = words[i]

        # Use the calculate_negative_log_prob function
        log_prob = smoothing_log_probability(wm, wm_minus_1,
bigram_counts, alpha, vocabulary_size)

        # Add the log probability to the overall log probability
        log_probability += log_prob

    return log_probability

# Example sentences
sentence1 = "this was a really great movie but it was a little too
long."
sentence2 = "long too little a was it but movie great really a was
this."

# Example usage with alpha=0.001 and vocabulary size
alpha = 0.001
vocabulary_size = 10000  # Replace with your vocabulary size

log_prob1 = sentence_log_probability(sentence1, bigram_counts, alpha,
vocabulary_size)
log_prob2 = sentence_log_probability(sentence2, bigram_counts, alpha,
vocabulary_size)

print(f'Log Probability (alpha={alpha}) - Sentence 1: {log_prob1}')
print(f'Log Probability (alpha={alpha}) - Sentence 2: {log_prob2}')
```

```
Log Probability (alpha=0.001) - Sentence 1: -91.40757002585701
Log Probability (alpha=0.001) - Sentence 2: -218.02674910987346
```

#PROBLEM 6 – Tuning Alpha (10pts) 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. • Which alpha gives you the best result? To indicate your selection to the grader, save your selected value to a variable named "selected_alpha".

```
# Define the list of alpha values to test
alpha_values = [0.001, 0.01, 0.1]

# Initialize variables to keep track of the best alpha and its
corresponding log likelihood
best_alpha = None
best_log_likelihood = float("-inf")
```

```
for alpha in alpha_values:
    total_log_likelihood = 0.0

    # Calculate log likelihood for each sentence in the validation set
    for sentence in validation_set['sentence']:
        # log_prob = sentence_log_probability(sentence, bigram_counts,
alpha, vocabulary_size)
        total_log_likelihood += sentence_log_probability(sentence,
bigram_counts, alpha, vocabulary_size)

    print(f"Log likelihood for alpha={alpha}: {total_log_likelihood}")

    # Check if this alpha is the best so far
    if total_log_likelihood > best_log_likelihood:
        best_log_likelihood = total_log_likelihood
        best_alpha = alpha

# Display the best alpha and its corresponding log likelihood
print(f"Best alpha: {best_alpha}")
print(f"Log likelihood with the best alpha: {best_log_likelihood}")

# Save the selected alpha to a variable named "selected_alpha"
selected_alpha = best_alpha
```

```
Log likelihood for alpha=0.001: -4981.168778451172
Log likelihood for alpha=0.01: -5508.845270385804
Log likelihood for alpha=0.1: -6644.975804447448
Best alpha: 0.001
Log likelihood with the best alpha: -4981.168778451172
```

#PROBLEM 7 – Applying Language Models (20 pts) 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: o 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. o 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.

```
# Step 1: Separate Training Dataset
positive_training = training_set[training_set['label'] == 1]
negative_training = training_set[training_set['label'] == 0]
```

```
# Calculate vocabulary size and bigram counts for both datasets
positive_bigram_counts =
count_bigrams(positive_training['tokenized_sentence'])
negative_bigram_counts =
count_bigrams(negative_training['tokenized_sentence'])

# Step 2: Classify Test Sentences
def classify_sentiment(sentence, positive_bigram_counts,
negative_bigram_counts, alpha_positive, alpha_negative,
vocabulary_size, prior_positive, prior_negative):
    positive_log_prob = sentence_log_probability(sentence,
positive_bigram_counts, alpha_positive, vocabulary_size) +
math.log(prior_positive)
    negative_log_prob = sentence_log_probability(sentence,
negative_bigram_counts, alpha_negative, vocabulary_size) +
math.log(prior_negative)
    return "positive" if positive_log_prob > negative_log_prob else
"negative"

# Classify test sentences
selected_alpha_positive = 0.001
selected_alpha_negative = 0.001

test_set = test_set.copy()
test_set['predicted_sentiment'] = test_set['sentence'].apply(lambda x:
classify_sentiment(x, positive_bigram_counts, negative_bigram_counts,
selected_alpha_positive, selected_alpha_negative, vocabulary_size,
positive_prior_probability, negative_prior_probability))

# Step 3: Evaluate Predictions
correct_predictions = (test_set['label'] ==
(test_set['predicted_sentiment'] == 'positive')).sum()
accuracy = correct_predictions / len(test_set)

# Calculate class distribution
predicted_positive = (test_set['predicted_sentiment'] ==
'positive').sum()
predicted_negative = (test_set['predicted_sentiment'] ==
'negative').sum()

print(f"Class Distribution - Predicted Positive:
{predicted_positive}")
print(f"Class Distribution - Predicted Negative:
{predicted_negative}")
print(f"Accuracy: {accuracy*100:.2f}%")
```

```
Class Distribution - Predicted Positive: 53
Class Distribution - Predicted Negative: 47
Accuracy: 92.00%
```