```
import re # A library that handles Regular Expressions (pattern matching and
replacement for strings)
import os # A library that allows us to read every file in a directory
from pprint import pprint # Print dictionaries in an attractive way
import json # A library to deal with JSON save/load
import math # Mathematical functions



def tokenize(text) -> list:
      """
      Given a string, return a list of the lowercase tokens (words) in that string
      More about tokenization and data cleaning for Data Science
      https://towardsdatascience.com/5-simple-ways-to-tokenize-text-in-python-
92c6804edfc4
      https://towardsdatascience.com/how-to-efficiently-remove-punctuations-from-a-
string-899ad4a059fb
      """
      # Split this text on any whitespace, then for each section
      # * remove any non alphabetic character ([^A-Za-z])
      # * convert to lowercase
      # * strip away any extra whitespace
      tokens = [re.sub(r'[^A-Za-z]+', '', s).lower().strip() for s in
text.split()]
      return tokens

def add_text_to_frequency_dict(freq_dict, text):
      """
      Here is a text that contains some words:
      e.g. 'It's one of the greatest films!!!!'
      Tokenize that to get a list of 'normalized' words (without punctuation and
lowercased)
       -> ['its', 'one', 'of', 'the', 'greatest', 'films']
      Is this a positive or negative review? We don't know or care in this function
      All we need to do is add each word in this text to the freq_dict
      For each word in the tokens
      * if it is in the dictionary already,
            add one to the current value for that word in the dictionary
      * if it isn't, set the value to 1

      Parameters:
            freq_dict(dict): A dictionary of word frequencies
                  (how often we have seen a word in a text)
            text(str): Some text for use to add to the frequency dict
      Returns:
            None
      """

      # Task 0: Add some text to a dictionary
      # Tokenize the text to obtain a list of normalized words
      tokens = tokenize(text)

    # Loop through each token and add it to the frequency_dict
      for token in tokens:
            if token in freq_dict:
            # If the token exists, increment its frequency by 1
                  freq_dict[token] += 1
            else:
            # If the token doesn't exist, add it to the dictionary with a frequency
```

```
of 1
                  freq_dict[token] = 1

      # This function should return None, as it does not need to return anything
explicitly
      return None

      # Note that this function should return None
      # (that is, if functions don't return a value expicitly,
      #  they return None when they run out of function to execute)



def get_total_words(freq_dict):
      """
      Calclulate the total count of all words in this dictionary
      (*not* the unique number of words. IE, "cat cat cat" -> 3 words)
      Note that you can use some_dict.values() to get a list of all values in a
dictionary

      Parameters:
            freq_dict(dict): a dictionary of frequencies of words, like {"cat":10}
      Returns:
            the total number of word instances in this frequency dict (not unique
words)
      """

      # Task 1: Count up the total number of words in this frequency dictionary
      # You can use a for loop,
      # ...or use the sum(a list of numbers) built-in function,
      # but then you have to get a list of the numerical values
      #  (https://www.programiz.com/python-programming/methods/dictionary/values)
      x= sum(freq_dict.values())
      return x


# ---------------------------------------------------------------------
# ----------------------- WOW, A CLASS! -------------------------
# Read more about those here:
# https://www.programiz.com/python-programming/class
# Mostly you need to know that a class is a way of creating *instances*
# of the class, each with their own data
# For that reason, most methods (functions in a class) take *self*
# as a parameter, so that you can access the data in that instance
#  as "self.some_data" or "self.my_cool_method()"
# You don't need to pass "self" to a method, the instance will magically get
#  added to the parameters of the method when you call it on an instance

class BayesClassifier:
      def __init__(self):

            # This is how we tell if a review is positive or negative
            # ...it starts with one of these prefixes
            self.pos_file_prefix = "movies-5"
            self.neg_file_prefix = "movies-1"
            self.pos_freqs = {}
            self.neg_freqs = {}

      def __str__(self):
```

```
            """
            Note that this is the total number of different words,
            not the total number of words in the reviews
            """

            pos_count = len(self.pos_freqs.keys())
            neg_count = len(self.neg_freqs.keys())
            return f"BayesClassifier ({pos_count} unique words in pos_freqs,
{neg_count} in neg_freqs)"


      def train(self, directory_name):
            """
            "Train" this classifier by reading every .txt file in "directory_name"

            Note that you will need to check if filenames contain substrings
                  e.g. (".txt", neg_file_refix)
            There are lots of ways to do this with Python, but the easiest is
                  "some_string.startswith(some_prefix)"
                  "some_string.endswith(some_suffix)"

            For each file name in this directory (if it ends in ".txt")
                  * make a filepath using os.path.join to join the directory and
filename into one filepath
                        This handles the issue with / or \ going the opposite way
on Windows/Mac
                        (https://www.golinuxcloud.com/python-os-path-join-method/)
                  * Open the filepath (like usual)
                  * Use "read" to read in the text as a list of strings
                        (remember readlines? We used it back in A1.
                         "read" is like that but doesn't split it up into lines
                         so it returns a string rather than a list of strings)
                  * if this review is *positive*, (i.e., the filename begins with
pos_file_prefix)
                        then we will add it to the frequency data in pos_freqs
                        otherwise if it is negative (i.e., it begins with
neg_file_prefix)
                        we will add it to the frequency data in neg_freqs
                        (If it doesn't start with either, ignore it)


            Parameters:
                  directory_name(str)
            Returns:
                  None
            """

            # Task 2: Open a directory and "train" (store word frequency)
            #   on all the files' contents
            file_names = os.listdir(directory_name)
            print(f"Training on {len(file_names)} files")

            # Uncomment if you want to see the names you are training on
            # print(file_names)

            #forst check for .txt
            for i in file_names:
                  if i.endswith(".txt"):
                        #get the file path and open it
```

```
                        filepath = os.path.join(directory_name, i)

                        open_file = open(filepath, "r", encoding="utf8")

                        #coverting to a string to read whethewr its +Ve or not
                        read_file = open_file.read()


                        if i.startswith(self.pos_file_prefix):

                              add_text_to_frequency_dict(self.pos_freqs, read_file)

                        #if its -ve add to apropriate file

                        elif i.startswith(self.neg_file_prefix):
                              add_text_to_frequency_dict(self.neg_freqs, read_file)

                        open_file.close()
            return None




      def get_neg_frequency(self, word):
            """
            Parameters:
                  word(str): some word that may or may not be in the negative
dictionary
            Returns:
                  int: 0 if the word is not in the dictionary, or the correct
number if it is
            """

            # Task 3: return negative word frequency
            if word in self.neg_freqs:

                        #    # If the word exists in the dictionary, return its
frequency count

                  return self.neg_freqs[word]
            #
            return 0

      def get_pos_frequency(self, word):
            """
            Parameters:
                  word(str): some word that may or may not be in the positive
dictionary
            Returns:
                  int: 0 if the word is not in the dictionary, or the correct
number if it is
            """

            # Task 4: return positive word frequency

            if word in self.pos_freqs:
```

```
            ## If the word exists in the dictionary, return its frequency count
                return self.pos_freqs[word]
          return 0


    def save(self, filename):
          """
          Save the positive and negative dictionaries to a JSON file
           so we can load them later

          Parameters:
                filename(str): The name of the JSON file to store
                      this predictor's dictionaries
          Returns:
                None
          """

          # We want to store both pos_freqs and neg_freqs, so create an object
          # {"pos": self.pos_freqs, "neg": self.neg_freqs}
          # Open a file for writing, and use the
          #     json.dump(some_data, some_file, indent=4, sort_keys=True)
          #     ("dump" is a terrible name, I know)
          #  method to write the object to the file
          # Relevant review:
          # https://www.programiz.com/python-programming/file-operation
          # https://www.programiz.com/python-programming/json
          # Note that when saving a file that doesn't exist yet, we need to have
          # "open(filename, 'w')" <- the 'w' tells python to *write* a file

          # Task 5

          cheekum = {"pos": self.pos_freqs, "neg": self.neg_freqs}
          json_file = open(filename, 'w')
          json.dump(cheekum, json_file, indent=4, sort_keys=True)
          json_file.close()

          return None

    def load(self, filename):
          """
          Load this json object (json.load) and set the pos_freqs and neg_freqs
dictionaries
          to the ones loaded from the saved copy
          Parameters:
                filename(str): The name of the JSON file to store
                      this predictor's dictionaries
          Returns:
                None
          """

          # Task 6: load self.neg_freqs and self.pos_freqs from JSON

          json_file = open(filename)
          my_data = json.load(json_file)
          json_file.close()

          if "pos" in my_data and "neg" in my_data:
                self.pos_freqs = my_data["pos"]
                self.neg_freqs = my_data["neg"]
```

```
            return None

      def reset(self):
            """
            UTILITY FUNCTION
            Forget all the positive and negative frequencies
            'reset' my memory, useful for testing save/load
            """
            print("-- reset this classifier, forget all the words --")
            self.pos_freqs = {}
            self.neg_freqs = {}


      #-----------------------------------------
      # Prediction code


      def get_likelihood_of_word(self, word, is_positive):
            """
            Get the likelihood of a word, given some distribution of words
             (a dictionary of frequencies)

            This will be our P(word|positive_review) or P(word|negative_review)
             ("probability of this word given this dictionary")
             when we do our Bayes theorem

            In this function we will also do "add-one smoothing",
             a mathematical hack where we always add 1 to the frequency,
             so that no word, even "xhjcskhv", has a likelihood of 0, e.g.:
             If the word has a count of 57, it will be 58
             If the word has a count of 0, it will be 1

            Parameters:
                  word: a word that may or may not  be in our dictionary,
                  is_positive: True if we are using the frequencies in in the
positive dictionary,
                  False if we are comparing against the negative dcitionary
            Returns:
                  float: a likelihood between not-quite-zero and 1,
                        representing how likely that word is to be
                        randomly drawn from this frequency
            """

            # Task 7: calculate and return the likelihood that a word would be
            # randomly drawn from the positive or negative dictionary
            # (be sure to do plus-one smoothing!)


            # check if the review is positive
            if is_positive:
                  freq_dict = self.pos_freqs
            else:
                  freq_dict = self.neg_freqs

            # initialize a count variable
            count = freq_dict.get(word, 0) + 1
            total = get_total_words(freq_dict)
```

```
            probability = count / total # calculate the probability of the word
being positive or negative based on its frequency count and the total number of
words in the corresponding dictionary

            return probability # return the probability value




      def get_log_likelihood_of_text(self, text, is_positive):
            """
            How likely is this text (a bunch of words), for this dictionary?
            To get the probability of many independent events happening
             -> rolling '6' on a dice 5 times in a row
            normally we would multiply the probabilities
             -> (1/6 * 1/6 * 1/6 * 1/6 * 1/6) = 0.00012860082

            But our results for get_likelihood_of_word were already *very small*
            If we multiplied these small numbers, it would quickly become
            a number that is *too small for Python to store*!

            So instead we can do a Math trick where instead of multiplying
            the probabilities,
            we *add* the logarithms of the probabilities instead.

            This keeps the numbers from getting super-tiny, but still allows
            us to compare two probabilities and see which is larger

            Parameters:
                  text: Some text that may have many words
            Returns:
                  float: the sum of the log likelihoods of all the words
            """

            # Task 8: tokenize this text. For each token,
            #     compute the likelihood of this word being pulled from
            #           the dictionary (use previous method)
            #     add up all of the logs (use math.log(some_number))

            probab = sum([math.log(self.get_likelihood_of_word(token, is_positive))
for token in tokenize(text)])
            return probab




      def get_prediction(self, text):
            """
            Predict the likelihood of this text coming from a positive
            or negative review.

            We will assume equal numbers of positive and negative reviews.
            Normally Bayesian stats requires a "prior", if, for example
            only 1 out of every 100 reviews was positive, we would want to adjust
            our predictions accordingly. But we won't do that here.
```

```
            Parameters:
                  text(str): A string that may contain many words
            Returns:
                  str: "positive" if the log_likelihood is equal or higher for the
positive review
                        "negative" if the log_likelihood is higher for the
negative review
            """

            # Task 9: return a "positive" or "negative" prediction for this text

            log_positive = self.get_log_likelihood_of_text(text, True)
            log_negative = self.get_log_likelihood_of_text(text, False)
            if log_positive - log_negative > 0:
                  return "positive"
            elif log_negative - log_positive > 0:
                  return "negative"
            else:
                  return "unknown"



if __name__ == "__main__":

      # Note: for better formatting, I am using some of the nice formatting
      # tricks available with f-strings https://saralgyaan.com/posts/f-string-in-
python-usage-guide/
      #  f"{some_number:.2f} {some_text:20}"
      # which rounds the number to 2 decimal places
      # and pads the text until it is 20 characters

      #-----------------------------------------------------------------------
      # Test task 0
      # Make an empty dictionary, and add some test to it
      test_dict = {}
      add_text_to_frequency_dict(test_dict, "cats, cats, CATS!!!")
      add_text_to_frequency_dict(test_dict, "wow, I love cats")
      add_text_to_frequency_dict(test_dict, "CATS(2019) is the best movie, love it
forever")

      # Prints the test_dict in a nice way
      pprint(test_dict)

      assert test_dict["wow"] == 1, "'wow' should be in this dictionary now"
      assert test_dict["cats"] == 5, "Make sure you are adding new words to the
dictioary, not just setting it to 1"

      assert add_text_to_frequency_dict(test_dict, "some text") == None, "add_text
should return None"

      # #-----------------------------------------------------------------------
      # # Test task 1
      test_dict = {}
      add_text_to_frequency_dict(test_dict, "cats, cats, CATS!!!")
      add_text_to_frequency_dict(test_dict, "wow, I love cats")
      add_text_to_frequency_dict(test_dict, "CATS(2019) is the best movie, love it
forever")
      count = get_total_words(test_dict)
```

```
      print(f"The number of words in this frequency dict is {count}")
      assert count == 15, "Make sure you are counting the total words, not unique
words"

      # CREATE AN INSTANCE OF A CLASS!
      # First time we have seen that, notice that later we create a SECOND instance
(large_classifier)
      small_classifier = BayesClassifier()

      # #---------------------------------------------------------------------
      # # Test task 2

      # # Train on a small set of reviews
      small_classifier.train("movie_reviews_small")

      neg_count = get_total_words(small_classifier.neg_freqs)
      pos_count = get_total_words(small_classifier.pos_freqs)
      print(f"Total negative words {neg_count}, total positive words {pos_count}")

      # # #---------------------------------------------------------------------
      # # Test task 3, 4

      example_words = ["greatest", "awesome", "worst", "zoodles"]
      for word in example_words:
            print(f"'{word}' occurs in negative reviews:
{small_classifier.get_neg_frequency(word)}")
            print(f"'{word}' occurs in positive reviews:
{small_classifier.get_pos_frequency(word)}")

      assert small_classifier.get_pos_frequency("worst") == 1
      assert small_classifier.get_neg_frequency("awesome") == 2
      assert small_classifier.get_pos_frequency("awesome") == 5, "make sure that
you are adding to correct (positive or negative) dictionaries"


      # #---------------------------------------------------------------------
      # # Test task 5

      print(small_classifier)
      small_classifier.save("small_freqs.json")

      # To test this, see if "small_freqs.json" is now in your folder
      assert os.path.exists("small_freqs.json"), "I don't detect that this file
exists!"

      # #---------------------------------------------------------------------
      # # Test task 6

      # # Reset the classifier to forget all the words
      small_classifier.reset()
      print(small_classifier)
      small_classifier.load("small_freqs.json")
      print(small_classifier)
      assert get_total_words(small_classifier.neg_freqs) == 403, "There should be
403 words in the negative dictionary after loading"
      assert get_total_words(small_classifier.pos_freqs) == 558, "There should be
558 words in the positive dictionary after loading"
```

```
      # #---------------------------------------------------------------------
      # # Test task 7

      # # Test out the likelihood and log likelihood
      # # Notice that even rare words don't have super tiny numbers if you use logs
      # # And that missing words ("zoodles") aren't 0
      for word in example_words:
            pos_likelihood = small_classifier.get_likelihood_of_word(word, True)
            neg_likelihood = small_classifier.get_likelihood_of_word(word, False)
            print(f"likelihood of '{word:14}': pos {pos_likelihood:.8f}, neg
{neg_likelihood:.8f}")
            pos_log_likelihood = math.log(pos_likelihood)
            neg_log_likelihood = math.log(neg_likelihood)
            print(f"log likelihood of '{word:10}': pos {pos_log_likelihood:.8f},
neg {neg_log_likelihood:.8f}")


      greatest_pos = small_classifier.get_likelihood_of_word("greatest", True)
      greatest_neg = small_classifier.get_likelihood_of_word("greatest", False)
      zoodle_pos = small_classifier.get_likelihood_of_word("zoodles", True)
      zoodle_neg = small_classifier.get_likelihood_of_word("zoodles", False)
      print("Greatest likelihood:", greatest_pos, greatest_neg)
      print("Zoodle likelihood:", zoodle_pos, zoodle_neg)
      # Test if these numbers are close (Python's floats usually not exact matches)
      assert math.isclose(greatest_pos, 0.00537634, rel_tol=1e-3, abs_tol=0.0)
      assert math.isclose(greatest_neg, 0.00248139, rel_tol=1e-3, abs_tol=0.0)
      assert math.isclose(zoodle_pos, 0.00179211, rel_tol=1e-3, abs_tol=0.0)
      assert math.isclose(zoodle_neg, 0.00248139, rel_tol=1e-3, abs_tol=0.0)

      #---------------------------------------------------------------------
      # Test task 8

      # A function for printing lots of preditions for a list of texts
      def test_log_likelihood(classifier, texts_to_test):
            for text in texts_to_test:
                  likelihood_pos = classifier.get_log_likelihood_of_text(text,
True)
                  likelihood_neg = classifier.get_log_likelihood_of_text(text,
False)

                  print(f"{text:30} pos:{likelihood_pos:.7f}, neg:
{likelihood_neg:.7f}")


      # Here are some good and bad reviews for us to test
      # What is the likelihood picking random words from
      # the negative dictionary would result in the review?
      # What is the likelihood picking random words from
      # the positive dictionary would result in the review?

      bad_review = "Worst, bad, awful!"
      good_review = "brilliant, a true classic"
      ok_review = "its ok if you like it"


      test_log_likelihood(small_classifier, [bad_review, good_review, ok_review])

      bad_review_likelihood_neg =
small_classifier.get_log_likelihood_of_text(bad_review, False)
```

```
      bad_review_likelihood_pos =
small_classifier.get_log_likelihood_of_text(bad_review, True)
      print(f"The small classifier says the likelihood that '{bad_review}'  is a
positive review is {bad_review_likelihood_pos}")
      print(f"The small classifier says the likelihood that '{bad_review}'  is a
negative review is {bad_review_likelihood_neg}")
      assert math.isclose(bad_review_likelihood_pos, -18.2799297, rel_tol=1e-3,
abs_tol=0.0)
      assert math.isclose(bad_review_likelihood_neg, -14.8187559, rel_tol=1e-3,
abs_tol=0.0)

      large_classifier = BayesClassifier()

      # # You only need to run this section once, because you can always
      # #  load your json file after saving it.
      # # Notice: you can do the work once, and then you can use it
      # #  many times without rerunning slow code
      print("Training on the *large* dataset, this may take a while")
      large_classifier.train("movie_reviews")
      print("Done training")
      large_classifier.save("large_freqs.json")

      large_classifier.load("large_freqs.json")
      test_log_likelihood(large_classifier, [bad_review, good_review, ok_review])

      bad_review_likelihood_neg =
large_classifier.get_log_likelihood_of_text(bad_review, False)
      bad_review_likelihood_pos =
large_classifier.get_log_likelihood_of_text(bad_review, True)
      print(f"The large classifier says the likelihood that '{bad_review}'  is a
positive review is {bad_review_likelihood_pos}")
      print(f"The large classifier says the likelihood that '{bad_review}'  is a
negative review is {bad_review_likelihood_neg}")
      assert math.isclose(bad_review_likelihood_pos, -27.754, rel_tol=1e-3,
abs_tol=0.0)
      assert math.isclose(bad_review_likelihood_neg, -18.521, rel_tol=1e-3,
abs_tol=0.0)

      # #-----------------------------------------------------------------
      # # Test task 9

      def test_predictions(classifier, texts_to_test):
            for text in texts_to_test:
                  predition = classifier.get_prediction(text)
                  print(f"{text:30} {predition}")

      test_predictions(large_classifier, [bad_review, good_review, ok_review])

      assert large_classifier.get_prediction(bad_review) == "negative", "The bad
review should return 'negative'"
      assert large_classifier.get_prediction(good_review) == "positive", "The good
review should return 'positive'"

      # # Lets see how well this system actually classifies randomly selected
reviews
      # # On average, can it determine a good or a bad review?
      # # (the above tests were carefully cherrypicked to get good results
      # #   bayesian bag-of-word models are actually not very effective!)
      # # NOTE: THERE IS NOTHING TO DO HERE, BUT DON'T YOU WANT TO SEE THE ENDING??
```

```
      def test_classification(classifier):
            pos_test_reviews = open("all_pos.txt", "r").readlines()
            neg_test_reviews = open("all_neg.txt", "r").readlines()
            total_test_reviews = len(pos_test_reviews) + len(neg_test_reviews)

            false_positives = []
            false_negatives = []
            correctly_labeled = 0

            for review in pos_test_reviews:
                  prediction = classifier.get_prediction(review)
                  if prediction == "positive":
                        correctly_labeled += 1
                  else:
                        false_negatives.append(review)
            for review in neg_test_reviews:
                  prediction = classifier.get_prediction(review)
                  if prediction == "negative":
                        correctly_labeled += 1
                  else:
                        false_positives.append(review)

            print(f"{correctly_labeled}/{total_test_reviews}
{100*correctly_labeled/total_test_reviews:2f}% correct")

      #     # Uncomment to see the misclassified ones:
            print("False negatives: " + " | ".join(false_negatives))
            print("False positives: " + " | ".join(false_positives))

      print("\nSmall classifier performance")
      test_classification(small_classifier)
      print("\nLarge classifier performance")
      test_classification(large_classifier)


# Skip the test code if we are running this as a *module* (ie, when we are grading
it)
# # Otherwise, run your test code!
# if __name__ == "__main__":
#     run_test_code()
```