CSE 5243 Lab 2 – Classification of Reuters Article Text

Afnan Rehman, Matthew Stock, & Kumar Dhital

Section 26469

November 6, 2018

Feature Vector Design:

Our feature vectors were based off a bag of words model and a hybrid bag of words model which weights different words based on their characteristics. We felt these two vector types would provide a solid baseline classification system. We also thought it would be interesting to compare the two and see how accurate the different models would be in a real-life scenario.

The first vector is a standard bag of words model, where each word that appeared in the body of a document was counted and added to the feature vector. This provided a baseline for our second vector which used custom weights for each word. The second vector prioritizes words which are in the TITLE tag of the different records, by placing any word in the title into the bag of words multiple times. This essentially weights the word by making it more important in relation to a specific topic or place. The second vector also gives increased importance to words which match the topic or place, for example, in the training data there will be many Canada words in the vector if the PLACE is also Canada.

We decided to create two vectors of each type, one for PLACES and one for TOPICS. The classes were sorted by individual items such as a specific place or topic, even if there were two or more grouped together for an article. The way we went about reconciling this is by providing a count for the word to each vector if two or more vectors intersected on an article.

For example, if an article contained a PLACES key for both ‘usa’ and ‘uk’, the body text would be counted towards both the vector for ‘usa’ and the vector for ‘uk’. This way we did not lose any information, and the door is left open in case you wanted to consider ‘usa’+’uk’ as a separate class altogether.

We used Python for the entire process due to its wealth of tools and libraries available including nltk, scikit-learn, pandas, beautifulsoup, and others. It also proved a most robust tool when dealing with messy input, making data cleaning easier and was the tool with the most familiarity to all the group members.

Selected Classifier:

For this particular case, we chose a standard Naïve Bayes classifier using the Python NLTK language processing library to tokenize words. This approach allowed us to construct different vectors of words with different parameters while still using the same classifier for both vectors.

The naïve_bayes.py file shows the process of using a bag-of-words vector to classify different bags as pertaining to a specific place or topic. Both vectors are classified by the naïve_bayes.py file and the output is the probability that the selected PLACE or TOPIC is each of the possible PLACES or TOPICS.

Training and Validation:

For this step, we used cross-validation as a way to segment and then validate the training data. We decided to split the data based on the file it was in, making the process of segmenting easy. Once we completed this, we averaged the results and compared. Essentially, the corpus was broken into k = 21 subsets with k-1 used for training and 1 used for validation.

For example, one training set would be reut2-001.sgm – reut2-021.sgm, and reut2-000.sgm would be held out for validation. This could be repeated for every file in the given set and then accuracy results could be averaged. Using this cross-validation technique we are able to use most of the data for training the Naïve Bayes classifier, while retaining an entire file for testing, so that the classifier still can be accurately tested.

Results:

The results were expectedly not incredibly accurate but telling in several ways. We defined accuracy as the number of correct class predictions divided by the number of incorrect class predictions.

For our PLACES we were able to test validation data against training data and get consistent matches, as well as infinitesimally small numbers for probabilities against other countries which are counted as essentially 0. Sorting these numbers in Excel (as we have them output to CSV) helped us understand which countries were more distinct from each other and which ended up matching very closely. For example, in our tests, we found Canada and the US to be extremely similar when we used our bag of words vector, causing the classifier to almost rank them equal in terms of probability against the test data. Overall, we achieved bout 55-64% accuracy in most cases. The modified bag of words model slightly improved accuracy, increasing the overall results from 55-64% to 58-68%. Below is a sample output of something you might see, and in the appendix, you can see a fuller table of results.

Sample output for Naïve Bayes Classifier:

Id	‘canada’	‘uk’	‘pakistan’
1 (UK)	1.77e-1285	1	1.47e-984
2 (USA)	1	5.46e-1903	4.6e-532
3 (Canada)	1	6.19e-1526	6.3e-150

Moving on to the TOPICS, we found lower accuracies of around 25-30%. The modified bag of words method had a larger impact on the TOPICS category, improving accuracies to 35-40%. This is most likely due to the fact that the importance of title words helped classify the texts more precisely, as documents are very likely to be titled with words relevant to the topic of discussion. We believe that the shorter bags led to a lower accuracy as topics were more sparely populated and the number of possible topics to classify were much greater than the number of places.

Distribution of work:

Afnan Rehman – Implemented Naïve Bayes classifier (with cited work as reference), created bag of words vector, created final report document, code testing and validation

Kumar Dhital – Implemented word counter

Matthew Stock – Created the modified bag of words vector with weighted importance. Added modified vector results to the report and adjusted the Naïve Bayes classifier so that it works with both vector types.

Appendix:

Examples of Feature Vectors:

Bag of Words Vector (PLACE/TOPIC specific):

‘argentina’	‘Argentine grain board figures show crop …’
‘bangladesh’	‘Bangladesh police mounted a cross-country…’
‘usa’	‘Showers continued throughout the week…’

OR

‘wheat’	‘The US Agriculture Department said …’
‘earn’	‘Newmont Mining Corp's board has …’
‘retail’	‘The volume of UK Retail …’

Modified Bag of Words Vector (PLACE/TOPIC specific):

‘argentina’	‘Argentine grain board figures show crop …’
‘bangladesh’*	‘Bangladesh Bangladesh Bangladesh police…’
‘usa’	‘Showers continued throughout the week…’

OR

‘wheat’*	‘The US Agriculture Agriculture Department…’
‘earn’	‘Newmont Mining Corp's board has …’
‘retail’	‘The volume of UK Retail …’

*Note that certain words are repeated based on their importance.

Source code for countTags.py – Creates unweighted vector

```
# -*- coding: utf-8 -*-
"""
@author: Afnan, Cited Sources
Note: some of this is carried over from lab 1
"""
from bs4 import BeautifulSoup
from typing import Tuple
import csv
# BEGIN PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1
class TrieNode(object):
    """
    Trie node implementation.
    """

    def __init__(self, char: str):
        self.char = char
        self.children = []
        # Is it the last character of the word.
        self.word_finished = False
        # How many times this character appeared in the addition process
        self.counter = 1


def add(root, word: str):
    """
    Adding a word in the trie structure
    """
    node = root
    for char in word:
        found_in_child = False
        # Search for the character in the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found it, increase the counter by 1 to keep track that another
                # word has it as well
                child.counter += 1
                # And point the node to the child that contains this char
                node = child
                found_in_child = True
                break
        # We did not find it so add a new chlid
        if not found_in_child:
            new_node = TrieNode(char)
            node.children.append(new_node)
            # And then point node to the new child
            node = new_node
    # Everything finished. Mark it as the end of a word.
    node.word_finished = True


def find_prefix(root, prefix: str) -> Tuple[bool, int]:
    """
    Check and return
      1. If the prefix exsists in any of the words we added so far
      2. If yes then how may words actually have the prefix
    """
    node = root
    # If the root node has no children, then return False.
    # Because it means we are trying to search in an empty trie
    if not root.children:
        return False, 0
    for char in prefix:
        char_not_found = True
        # Search through all the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found the char existing in the child.
                char_not_found = False
                # Assign node as the child containing the char and break
                node = child
                break
        # Return False anyway when we did not find a char.
        if char_not_found:
            return False, 0
    # Well, we are here means we have found the prefix. Return true to indicate that
    # And also the counter of the last node. This indicates how many words have this
    # prefix
    return True, node.counter

# END PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1

# This method works like str.split, but splits for as many times as a delimiter shows up in the
doc
# It is also original work based on prior knowledge of how string splits work in Python.
def multi_splitter(input_string, delimiter):
    out_strings = []
    new_sub = str(input_string).split(delimiter)
    for str_element in new_sub:
        sub = str_element.split("</D>")
        out_strings.append(sub[0])
    return out_strings


def get_text(place, sources, places_bag_vector, t_type):
    # This portion involving reading the body text in from the file mostly done by Kumar
    print (place)
    total_text = ""
    for source in sources:
        with open(source) as f:
            data = f.read()
            soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to
structure of these files
            reuters_tags = soup.find_all('reuters')
            for reuter_tag in reuters_tags: # get information stored within each reuters tag
                if t_type == 'topics':
                    p_tag = reuter_tag.topics
                else:
                    p_tag = reuter_tag.places
                d_tags = p_tag.find_all('d') # find all places/topics mentioned
                for d_tag in d_tags:
                    for child in d_tag.children: # find relevant tags to current call and add
text to a master string
                        if(place == child):
                            try:
                                total_text += reuter_tag.body.get_text()
                            except:
                                total_text += ""

    # This subsequent section is devoted to removing a few bits of rather unwieldy extra
characters in our
    # output string. We wanted to retain as many words as possible, so more tedious methods of
extraction,
    # such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.
    array = total_text.split()
    new_array = []
    for word in array: # each word gets examined and picked apart if it contains the offending
characters
        new_word = ""
        if '\n' in word: # removing line breaks, wherever they may occur
            subword = word.split('\n')
            for part in subword:
                if '\n' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '.' in word: # removing punctuation
            subword = word.split('.')
            for part in subword:
                if '.' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if ',' in word: # removing punctuation
            subword = word.split(',')
            for part in subword:
                if ',' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '"' in word: # removing punctuation
            subword = word.split('"')
            for part in subword:
                if '"' not in part:
                    new_word += part
                    word = new_word
        word += " "
        new_array.append(word)

    cleaned_text = ""
    for newword in new_array:# now removing some final pesky words as well as any numbers we
don't want in our analysis
        if "reuter" not in newword.lower() and "\x03" not in newword and '"' not in newword and
newword.isdigit() == False:
            cleaned_text += newword
    # Optionally, add the finished bag of words to a output file
    cleaned_text.rstrip()
    file= open(place+'.txt', "a")
    try:
        file.write(cleaned_text)
    except:
        file.write("")
    file.close();

    # Create vector and return to calling function
    places_bag_vector[place] = cleaned_text
    # output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today
that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}
    return places_bag_vector

if __name__ == "__main__":
    sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \
               "files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \
               "files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \
               "files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \
               "files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \
               "files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \
               "files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \
               "files/reut2-021.sgm"]
    total_blank_places = 0
    total_blank_topics = 0
    total_countries = []
    total_topics = []
    root = TrieNode('*')


    # Here, my algorithm for splitting the elements of the TOPICS and PLACES fields is my
original work
    for source in sources:
        with open(source) as f: # Open the file and read line by line to a list array
            array = []
            for line in f:
                array.append(line)
        # Since PLACES were contained within one line of code according to the data I saw, I
assumed
        # that any line with the PLACES tag would contain all of the location info for that
article
        places = []
        for index in array: # Look at lines containing the "PLACES" tag and read those into a
separate list
```

```
            if "<PLACES>" in index:
                places.append(index)
        # Once I got the line, I split the string on the multiple "<D>" tags to extract the
location
        # information within
        new_places = []
        for place in places:
            new_places.extend(multi_splitter(place, "<D>")) # Using the helpful method above, I
split on one or more <D> tags
        new_places = [x for x in new_places if x not in ('', '/', '\n', 'PLACES', '/PLACES')]# I
then removed instances of tag information or blank information from the overall list

        # One trick I learned in coding Python for work is that by casting a list as a set,
        # you can remove duplicates in one line of code since sets do not contain duplicates
        distinct_countries = set(new_places)
        total_countries.extend(distinct_countries)

        # Next I moved onto TOPICS, using many of the same methods
        # that I used for PLACES to count and extract the information
        topics = []
        for index in array:
            if "<TOPICS>" in index:
                topics.append(index)

        # Once again I used the same string split method to extract the contents of each field
        tops = []
        for topic in topics:
            tops.extend(multi_splitter(topic, "<D>"))
        tops = [x for x in tops if x not in ('', '/', '\n', 'TOPICS', '/TOPICS')]

        # Counted distinct topics using the same cast to set
        distinct_topics = set(tops)
        # You may notice the issue with simply extending the list of total topics
        # There may end up being duplicates between documents that are not addressed
        # I address this issue in the final step: printing the statistics after all loops are
finished
        total_topics.extend(distinct_topics)

    # Here, we create all output vectors already sorted into training and test groups based on
cross-validation where k = 21
    # These files are then fed into the classifier program
    for i in range(21):
        training_sources = sources[:i] + sources[i+1:]
        test_sources = []
        test_sources.append(sources[i])
        # Here we begin to make our bag of words vectors
        # First we make the training groups

        # TEST SET FOR SPEED
        total_countries = ['afghanistan', 'uk', 'france',
'canada','turkey','usa','japan','pakistan']
        total_topics = ['acq', 'alum', 'lumber', 'jobs', 'interest', 'income','trade', 'wheat']
        # TEST


        bag_vector = {}
        for country in sorted(set(total_countries)):
            if "<PLACES>" not in country:
                get_text(country, training_sources, bag_vector, 'places')
        with open('place_bag_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["country", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        bag_vector = {}
        for topic in sorted(set(total_topics)):
            if "<TOPICS>" not in topic:
                get_text(topic, training_sources, bag_vector, 'topics')
        with open('topic_bag_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["topic", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        # These two will be the test groups
        bag_vector = {}
        for country in sorted(set(total_countries)):
            if "<PLACES>" not in country:
                get_text(country, test_sources, bag_vector, 'places')
        with open('place_bag_test' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["id", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        bag_vector = {}
        for topic in sorted(set(total_topics)):
            if "<TOPICS>" not in topic:
                get_text(topic, test_sources, bag_vector, 'topics')
        with open('topic_bag_test' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["id", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])
```

Source code for find_trie_prefix.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 12 01:23:38 2018

@author: Afnan
"""

# BEGIN PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1

from typing import Tuple

class TrieNode(object):
    """
    Trie node implementation.
    """

    def __init__(self, char: str):
        self.char = char
        self.children = []
        # Is it the last character of the word.
        self.word_finished = False
        # How many times this character appeared in the addition process
        self.counter = 1


def add(root, word: str):
    """
    Adding a word in the trie structure
    """
    node = root
    for char in word:
        found_in_child = False
        # Search for the character in the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found it, increase the counter by 1 to keep track that another
                # word has it as well
                child.counter += 1
                # And point the node to the child that contains this char
                node = child
                found_in_child = True
                break
        # We did not find it so add a new chlid
        if not found_in_child:
            new_node = TrieNode(char)
            node.children.append(new_node)
            # And then point node to the new child
            node = new_node
    # Everything finished. Mark it as the end of a word.
    node.word_finished = True


def find_prefix(root, prefix: str) -> Tuple[bool, int]:
    """
    Check and return
      1. If the prefix exsists in any of the words we added so far
      2. If yes then how may words actually have the prefix
    """
    node = root
    # If the root node has no children, then return False.
    # Because it means we are trying to search in an empty trie
    if not root.children:
        return False, 0
    for char in prefix:
        char_not_found = True
        # Search through all the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found the char existing in the child.
                char_not_found = False
                # Assign node as the child containing the char and break
                node = child
                break
        # Return False anyway when we did not find a char.
        if char_not_found:
            return False, 0
    # Well, we are here means we have found the prefix. Return true to indicate that
    # And also the counter of the last node. This indicates how many words have this
    # prefix
    return True, node.counter

# END PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1

if __name__ == "__main__":
    root = TrieNode('*')

    # Here my algorithm for splitting off and counting the body words is my own
    # Here the sources were hard coded. This can be easily changed to accept user input or to
    # search through a list of files given a directory if needed
    distinct_words = []
    sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \
               "files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \
               "files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \
               "files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \
               "files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \
               "files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \
               "files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \
               "files/reut2-021.sgm"]
    for source in sources:
        print("Parsing " + source[-13:])
        with open(source) as f: # Open the file and read line by line into array
            array = []
            for line in f:
                array.append(line)
        words = []
        body_on = False
        # Since there was no separator like the "<D>" used for TOPICS and PLACES,
        # extracting the body text of the article only needed a standard strng split along one
delimiter.
        for index in array:
            if "<BODY>" in index:
                index = index.split("<BODY>",1)[1]
                body_on = True # flag is used to track when to start and stop adding lines to the
body text
                words.append(index)
            if "</BODY>" in index:
                index = index.split("</BODY>",1)[0]
                body_on = False
            if body_on == True:
                words.append(index)
        distinct_words.extend(words)
        # Words are then added to the prefix trie using the add function above in a loop
    distinct_words = set(distinct_words) # list casted as set to get rid of duplicates
    print("Adding words to prefix trie...")
    for word in distinct_words:
        for word in distinct_words:
        try:
            int(word) # here I try to filter out integers that would not be useful in the future.
        except:
            add(root,word)
    print("Done!")
    # counts of words can be found and printed using the find_prefix methods as shown below
    print(find_prefix(root, 'limited'))
    print(find_prefix(root, 'the'))
```

Source Code for getBodyTextBag.py

```
# -*- coding: utf-8 -*-
"""
@author: Kumar, Afnan
"""

from bs4 import BeautifulSoup # Make sure BeautifulSoup is installed on your device before
running it

'''
This is the function takes PLACES as argument and takes body text from file for each body where
PLACES happens.
It output whole body text to a file based on its <place>.txt name where place is the name of the
country.
This method example could be placed in another script such as countTags.py to create vectors for
PLACES and TOPICS separaately.
It counts only one country at a time, so if 'usa' is in an article along with 'uk', then the body
of that article would
fall into both the USA key of the dictionary vector as well as the UK key of the vector.
'''
def get_text(place, sources, places_bag_vector):
    # This portion involving reading the body text in from the file mostly done by Kumar
    total_text = ""
    for source in sources:
        with open(source) as f:
            data = f.read()
            soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to
structure of these files
            reuters_tags = soup.find_all('reuters')
            for reuter_tag in reuters_tags: # get information stored within each reuters tag
                places_tag = reuter_tag.places
                d_tags = places_tag.find_all('d') # find all places/topics mentioned
                for d_tag in d_tags:
                    for child in d_tag.children: # find relevant tags to current call and add
text to a master string
                        if(place == child):
                            total_text += reuter_tag.body.get_text()

    # This subsequent section is devoted to removing a few bits of rather unwieldy extra
characters in our
    # output string. We wanted to retain as many words as possible, so more tedious methods of
extraction,
    # such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.
    array = total_text.split()
    new_array = []
    for word in array: # each word gets examined and picked apart if it contains the offending
characters
        new_word = ""
        if '\n' in word: # removing line breaks, wherever they may occur
            subword = word.split('\n')
            for part in subword:
                if '\n' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '.' in word: # removing punctuation
            subword = word.split('.')
            for part in subword:
                if '.' not in part:
                    new_word += part
                    word = new_word
        word += " "
        new_array.append(word)

    cleaned_text = ""
    for newword in new_array: # now removing some final pesky words as well as any numbers we
don't want in our analysis
        if "reuter" not in newword.lower() and "\x03" not in newword and newword.isdigit() ==
False:
            cleaned_text += newword
    # Optionally, add the finished bag of words to a output file
    file= open(place+'.txt', "a")
    try:
        file.write(cleaned_text)
    except:
        file.write("")
    file.close();
    # Create vector and return to calling function
    places_bag_vector[place] = cleaned_text
    return places_bag_vector
    # output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today
that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}

if __name__ == "__main__":
    sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \
               "files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \
               "files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \
               "files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \
               "files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \
               "files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \
               "files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \
               "files/reut2-021.sgm"]
    vector = {}
    vector = get_text("nepal", sources, vector) # call method
    print(vector)
```

Source code for naïve_bayes.py

```
# -*- coding: utf-8 -*-
"""
@author: https://www.kaggle.com/antmarakis/word-count-and-naive-bayes/notebook, Afnan

Most code on this document is sourced from https://www.kaggle.com/antmarakis/word-count-and-
naive-bayes/notebook
some adjustments were made by Afnan for use case
"""

import pandas as pd
import csv
from collections import Counter, defaultdict
from nltk.tokenize import word_tokenize
import decimal
from decimal import Decimal
csv.field_size_limit(100000000) # Expand to handle large field size of vector
decimal.getcontext().prec = 1000

def create_dist(text):
    c = Counter(text)

    least_common = c.most_common()[-1][1]
    total = sum(c.values())

    for k, v in c.items():
        c[k] = v/total

    return defaultdict(lambda: min(c.values()), c)

def precise_product(numbers):
    result = 1
    for x in numbers:
        result *= Decimal(x)
    return result

def NaiveBayes(dist):
    """A simple naive bayes classifier that takes as input a dictionary of
    Counter distributions and can then be used to find the probability
    of a given item belonging to each class.
    The input dictionary is in the following form:
        ClassName: Counter"""
    attr_dist = {c_name: count_prob for c_name, count_prob in dist.items()}

    def predict(example):
        """Predict the probabilities for each class."""
        def class_prob(target, e):
            attr = attr_dist[target]
            return precise_product([attr[a] for a in e])

        pred = {t: class_prob(t, example) for t in dist.keys()}

        total = sum(pred.values())
        for k, v in pred.items():
            pred[k] = v / total

        return pred

    return predict


def recognize(sentence, nBS):
    try:
        return nBS(word_tokenize(sentence.lower()))
    except:
        return nBS("")

def predictions(test_x, nBS):
    d = []
    for index, row in test_x.iterrows():
        i, t = row['id'], row['text']
        p = recognize(t, nBS)
        d.append({'id': i, 'canada': float(p['canada']), 'uk': float(p['uk'])})

    return pd.DataFrame(data=d)

for x in range(21):

    train_x = pd.read_csv("place_bag_train" + str(x) + ".csv", sep=',', encoding = "ISO-8859-1",
engine='python')
    test_x = pd.read_csv("place_bag_test" + str(x) + ".csv", sep=',', encoding = "ISO-8859-1",
engine='python')


    canada, uk = "", ""

    for i, row in train_x.iterrows():
        a, t = row['country'], row['text']
        if a == 'canada':
            canada += " " + t.lower()
        elif a == 'uk':
            uk += " " + t.lower()

    canada = word_tokenize(canada)
    uk = word_tokenize(uk)

    c_canada, c_uk = create_dist(canada),  create_dist(uk)

    dist = {'canada': c_canada, 'uk': c_uk}
    nBS = NaiveBayes(dist)


    submission = predictions(test_x, nBS)
    submission.to_csv('results_place' + str(x) + '.csv', index=False)
```

Source Code for CountTagsWeighted.py – Creating the “weighted” vector

```
# -*- coding: utf-8 -*-
"""
@author: Matt, Afnan, Cited Sources
Note: some of this is carried over from lab 1
"""
from bs4 import BeautifulSoup
from typing import Tuple
import csv
import re
# BEGIN PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1
class TrieNode(object):
    """
    Trie node implementation.
    """

    def __init__(self, char: str):
        self.char = char
        self.children = []
        # Is it the last character of the word.
        self.word_finished = False
        # How many times this character appeared in the addition process
        self.counter = 1


def add(root, word: str):
    """
    Adding a word in the trie structure
    """
    node = root
    for char in word:
        found_in_child = False
        # Search for the character in the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found it, increase the counter by 1 to keep track that another
                # word has it as well
                child.counter += 1
                # And point the node to the child that contains this char
                node = child
                found_in_child = True
                break
        # We did not find it so add a new chlid
        if not found_in_child:
            new_node = TrieNode(char)
            node.children.append(new_node)
            # And then point node to the new child
            node = new_node
    # Everything finished. Mark it as the end of a word.
    node.word_finished = True


def find_prefix(root, prefix: str) -> Tuple[bool, int]:
    """
    Check and return
      1. If the prefix exsists in any of the words we added so far
      2. If yes then how may words actually have the prefix
    """
    node = root
    # If the root node has no children, then return False.
    # Because it means we are trying to search in an empty trie
    if not root.children:
        return False, 0
    for char in prefix:
        char_not_found = True
        # Search through all the children of the present `node`
        for child in node.children:
            if child.char == char:
                # We found the char existing in the child.
                char_not_found = False
                # Assign node as the child containing the char and break
                node = child
                break
        # Return False anyway when we did not find a char.
        if char_not_found:
            return False, 0
    # Well, we are here means we have found the prefix. Return true to indicate that
    # And also the counter of the last node. This indicates how many words have this
    # prefix
    return True, node.counter

# END PART FROM https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-
less-than-100-lines-of-code-a877ea23c1a1

# This method works like str.split, but splits for as many times as a delimiter shows up in the
doc
# It is also original work based on prior knowledge of how string splits work in Python.
def multi_splitter(input_string, delimiter):
    out_strings = []
    new_sub = str(input_string).split(delimiter)
    for str_element in new_sub:
        sub = str_element.split("</D>")
        out_strings.append(sub[0])
    return out_strings


def get_text(place, sources, places_bag_vector, t_type):
    # This portion involving reading the body text in from the file mostly done by Kumar
    print (place)
    total_text = ""
    for source in sources:
        with open(source) as f:
            data = f.read()
            soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to
structure of these files
            reuters_tags = soup.find_all('reuters')
            for reuter_tag in reuters_tags: # get information stored within each reuters tag
                if t_type == 'topics':
                    p_tag = reuter_tag.topics
                else:
                    p_tag = reuter_tag.places
                d_tags = p_tag.find_all('d') # find all places/topics mentioned
                for d_tag in d_tags:
                    for child in d_tag.children: # find relevant tags to current call and add
text to a master string
                        if(place == child):
                            try:
                                total_text += reuter_tag.body.get_text()
                            except:
                                total_text += ""

    # This subsequent section is devoted to removing a few bits of rather unwieldy extra
characters in our
    # output string. We wanted to retain as many words as possible, so more tedious methods of
extraction,
    # such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.
    array = total_text.split()
    new_array = []
    for word in array: # each word gets examined and picked apart if it contains the offending
characters
```

```
        new_word = ""
        if '\n' in word: # removing line breaks, wherever they may occur
            subword = word.split('\n')
            for part in subword:
                if '\n' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '.' in word: # removing punctuation
            subword = word.split('.')
            for part in subword:
                if '.' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if ',' in word: # removing punctuation
            subword = word.split(',')
            for part in subword:
                if ',' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '"' in word: # removing punctuation
            subword = word.split('"')
            for part in subword:
                if '"' not in part:
                    new_word += part
                    word = new_word
        word += " "
        new_array.append(word)

    cleaned_text = ""
    for newword in new_array:# now removing some final pesky words as well as any numbers we
don't want in our analysis
        if "reuter" not in newword.lower() and "\x03" not in newword and '"' not in newword and
newword.isdigit() == False:
            cleaned_text += newword

    # Create vector and return to calling function
    places_bag_vector[place] = cleaned_text
    # output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today
that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}
    return places_bag_vector



def weighted_get_text(header_importance, place, sources, weighted_bag_vector, t_type):

    # This portion involving reading the body text in from the file mostly done by Kumar
    print (place)
    total_text = ""
    for source in sources:
        with open(source) as f:
            data = f.read()
            soup = BeautifulSoup(data, 'html.parser') # parse using HTML parser, close to
structure of these files
            reuters_tags = soup.find_all('reuters')
            for reuter_tag in reuters_tags: # get information stored within each reuters tag
                if t_type == 'topics':
                    p_tag = reuter_tag.topics
                else:
                    p_tag = reuter_tag.places

                d_tags = p_tag.find_all('d') # find all places/topics mentioned
                for d_tag in d_tags:
                    for child in d_tag.children: # find relevant tags to current call and add
text to a master string
                        if(place == child):

                            try:
                                total_text += reuter_tag.body.get_text()
                            except:
                                total_text += ""

                            title_tags = reuter_tag.find_all('title')
                            if (len(title_tags) > 0):
                                for title_tag in title_tags:
                                    for i in range(header_importance):
                                        total_text += title_tag.get_text()
                                        total_text += ' '

    # This subsequent section is devoted to removing a few bits of rather unwieldy extra
characters in our
    # output string. We wanted to retain as many words as possible, so more tedious methods of
extraction,
    # such as removing '\n' from the MIDDLE of the word was required. This part written by Afnan.
    array = total_text.split()
    new_array = []
    for word in array: # each word gets examined and picked apart if it contains the offending
characters
        new_word = ""
        if '\n' in word: # removing line breaks, wherever they may occur
            subword = word.split('\n')
            for part in subword:
                if '\n' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '.' in word: # removing punctuation
            subword = word.split('.')
            for part in subword:
                if '.' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if ',' in word: # removing punctuation
            subword = word.split(',')
            for part in subword:
                if ',' not in part:
                    new_word += part
                    word = new_word
        new_word = ""
        if '"' in word: # removing punctuation
            subword = word.split('"')
            for part in subword:
                if '"' not in part:
                    new_word += part
                    word = new_word
                    new_word = ""
        if '<' in word: # removing punctuation
            subword = word.split('<')
            for part in subword:
                if '<' not in part:
                    new_word += part
                    word = new_word
                    new_word = ""
        if '>' in word: # removing punctuation
            subword = word.split('>')
            for part in subword:
                if '>' not in part:
                    new_word += part
                    word = new_word
            word += " "
            new_array.append(word)

    cleaned_text = ""
    for newword in new_array:# now removing some final pesky words as well as any numbers we
don't want in our analysis
        if "reuter" not in newword.lower() and "\x03" not in newword and '"' not in newword and
newword.isdigit() == False:
            if (re.search(place, newword, re.IGNORECASE)):
                for i in range(header_importance * 2):
                    cleaned_text += newword
            else:
                cleaned_text += newword

    cleaned_text.rstrip()

    wordList = []
    words = re.findall(r'\b[a-zA-Z]+\b', cleaned_text)
    for word in words:
        wordList.append(word.lower())

    wordPairs = []
    for word in set(wordList):
        wordPairs.append([word, wordList.count(word)])

    cleaned_text = ""
    for wordPair in wordPairs:
        if wordPair[1] > 1:
            for i in range(wordPair[1]):
                cleaned_text += wordPair[0] + ' '

    # Create vector and return to calling function
    weighted_bag_vector[place] = cleaned_text
    # output looks like: {'afghanistan' : 'Pakistan complained to the United Nations today
that...', 'algeria' : 'Liquefied natural gas imports from Algeria...', ....}
    return weighted_bag_vector


if __name__ == "__main__":
    sources = ["files/reut2-000.sgm", "files/reut2-001.sgm", "files/reut2-002.sgm", \
               "files/reut2-003.sgm", "files/reut2-004.sgm", "files/reut2-005.sgm", \
               "files/reut2-006.sgm", "files/reut2-007.sgm", "files/reut2-008.sgm", \
               "files/reut2-009.sgm", "files/reut2-010.sgm", "files/reut2-011.sgm", \
               "files/reut2-012.sgm", "files/reut2-013.sgm", "files/reut2-014.sgm", \
               "files/reut2-015.sgm", "files/reut2-016.sgm", "files/reut2-017.sgm", \
               "files/reut2-018.sgm", "files/reut2-019.sgm", "files/reut2-020.sgm", \
               "files/reut2-021.sgm"]

    total_blank_places = 0
    total_blank_topics = 0
    total_countries = []
    total_topics = []
    root = TrieNode('*')


    # Here, my algorithm for splitting the elements of the TOPICS and PLACES fields is my
original work
    for source in sources:
        with open(source) as f: # Open the file and read line by line to a list array
            array = []
            for line in f:
                array.append(line)
        # Since PLACES were contained within one line of code according to the data I saw, I
assumed
```

```
        # that any line with the PLACES tag would contain all of the location info for that
article
        places = []
        for index in array: # Look at lines containing the "PLACES" tag and read those into a
separate list
            if "<PLACES>" in index:
                places.append(index)
        # Once I got the line, I split the string on the multiple "<D>" tags to extract the
location
        # information within
        new_places = []
        for place in places:
            new_places.extend(multi_splitter(place, "<D>")) # Using the helpful method above, I
split on one or more <D> tags
        new_places = [x for x in new_places if x not in ('', '/', '\n', 'PLACES', '/PLACES')]# I
then removed instances of tag information or blank information from the overall list

        # One trick I learned in coding Python for work is that by casting a list as a set,
        # you can remove duplicates in one line of code since sets do not contain duplicates
        distinct_countries = set(new_places)
        total_countries.extend(distinct_countries)

        # Next I moved onto TOPICS, using many of the same methods
        # that I used for PLACES to count and extract the information
        topics = []
        for index in array:
            if "<TOPICS>" in index:
                topics.append(index)

        # Once again I used the same string split method to extract the contents of each field
        tops = []
        for topic in topics:
            tops.extend(multi_splitter(topic, "<D>"))
        tops = [x for x in tops if x not in ('', '/', '\n', 'TOPICS', '/TOPICS')]

        # Counted distinct topics using the same cast to set
        distinct_topics = set(tops)
        # You may notice the issue with simply extending the list of total topics
        # There may end up being duplicates between documents that are not addressed
        # I address this issue in the final step: printing the statistics after all loops are
finished
        total_topics.extend(distinct_topics)

    # Here, we create all output vectors already sorted into training and test groups based on
cross-validation where k = 21
    # These files are then fed into the classifier program
    for i in range(3):
        training_sources = sources[:i] + sources[i+1:]
        test_sources = []
        test_sources.append(sources[i])
        # Here we begin to make our bag of words vectors
        # First we make the training groups

        # TEST SET FOR SPEED
        total_countries = ['afghanistan', 'uk', 'france',
'canada','turkey','usa','japan','pakistan']
        total_topics = ['acq', 'alum', 'lumber', 'jobs', 'interest', 'income','trade', 'wheat']
        # TEST


        #
        #   Vector 1: Generic Bag of Words method. Uses get_text to create bag of words.
        #
        bag_vector = {}
        for country in sorted(set(total_countries)):
            if "<PLACES>" not in country:
                get_text(country, training_sources, bag_vector, 'places')
        with open('place_bag_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["country", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        bag_vector = {}
        for topic in sorted(set(total_topics)):
            if "<TOPICS>" not in topic:
                get_text(topic, training_sources, bag_vector, 'topics')
        with open('topic_bag_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["topic", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        # Testing bag of words data.
        bag_vector = {}
        for country in sorted(set(total_countries)):
            if "<PLACES>" not in country:
                get_text(country, test_sources, bag_vector, 'places')
        with open('place_bag_test' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["id", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        bag_vector = {}
        for topic in sorted(set(total_topics)):
            if "<TOPICS>" not in topic:
                get_text(topic, test_sources, bag_vector, 'topics')
        with open('topic_bag_test' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["id", "text"])
            for key, value in bag_vector.items():
                writer.writerow([key, value])

        #
        #   Vector 2: Weighted Bag of Words method. Uses weighted_get_text to create bag of
words. Title words are five times as important,
        #   the name of the place or topic is ten times as important, and words which only appear
once are removed from the bag of words.
        #   Words are added multiple times based on their importance.
        #
        weighted_bag_vector = {}
        for country in sorted(set(total_countries)):
            if "<PLACES>" not in country:
                weighted_get_text(5, country, training_sources, weighted_bag_vector, 'places')
        with open('place_weighted_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["country", "text"])
            for key, value in weighted_bag_vector.items():
                writer.writerow([key, value])

        weighted_bag_vector = {}
        for topic in sorted(set(total_topics)):
            if "<TOPICS>" not in topic:
                weighted_get_text(5, topic, training_sources, weighted_bag_vector, 'topics')
        with open('topic_weighted_train' + str(i) + '.csv', 'w') as csv_file:
            writer = csv.writer(csv_file)
            writer.writerow(["topic", "text"])
            for key, value in weighted_bag_vector.items():
                writer.writerow([key, value])

        # Testing modified bag of words data.
```

```
weighted_bag_vector = {}
for country in sorted(set(total_countries)):
    if "<PLACES>" not in country:
        weighted_get_text(5, country, test_sources, weighted_bag_vector, 'places')
with open('place_weighted_test' + str(i) + '.csv', 'w') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(["id", "text"])
    for key, value in weighted_bag_vector.items():
        writer.writerow([key, value])

weighted_bag_vector = {}
for topic in sorted(set(total_topics)):
    if "<TOPICS>" not in topic:
        weighted_get_text(5, topic, test_sources, weighted_bag_vector, 'topics')
with open('topic_weighted_test' + str(i) + '.csv', 'w') as csv_file:
    writer = csv.writer(csv_file)
    writer.writerow(["id", "text"])
    for key, value in weighted_bag_vector.items():
        writer.writerow([key, value])
```

Source Code for naïve_bayes_weighted.py

```
# -*- coding: utf-8 -*-
"""
@author: https://www.kaggle.com/antmarakis/word-count-and-naive-bayes/notebook, Afnan
Most code on this document is sourced from https://www.kaggle.com/antmarakis/word-count-and-
naive-bayes/notebook
some adjustments were made by Afnan and Matt for use case
"""

import pandas as pd
import csv
from collections import Counter, defaultdict
from nltk.tokenize import word_tokenize
import decimal
from decimal import Decimal
csv.field_size_limit(100000000) # Expand to handle large field size of vector
decimal.getcontext().prec = 1000

def create_dist(text):
    c = Counter(text)

    least_common = c.most_common()[-1][1]
    total = sum(c.values())

    for k, v in c.items():
        c[k] = v/total

    return defaultdict(lambda: min(c.values()), c)

def precise_product(numbers):
    result = 1
    for x in numbers:
        result *= Decimal(x)
    return result

def NaiveBayes(dist):
    """A simple naive bayes classifier that takes as input a dictionary of
    Counter distributions and can then be used to find the probability
    of a given item belonging to each class.
    The input dictionary is in the following form:
        ClassName: Counter"""
    attr_dist = {c_name: count_prob for c_name, count_prob in dist.items()}

    def predict(example):
        """Predict the probabilities for each class."""
        def class_prob(target, e):
            attr = attr_dist[target]
            return precise_product([attr[a] for a in e])

        pred = {t: class_prob(t, example) for t in dist.keys()}

        total = sum(pred.values())
        for k, v in pred.items():
            pred[k] = v / total

        return pred

    return predict


def recognize(sentence, nBS):
    try:
        return nBS(word_tokenize(sentence.lower()))
    except:
        return nBS("")
```

```
def predictions(test_x, nBS):
    d = []
    for index, row in test_x.iterrows():
        i, t = row['id'], row['text']
        p = recognize(t, nBS)
        d.append({'id': i, 'canada': float(p['canada']), 'uk': float(p['uk'])})

    return pd.DataFrame(data=d)
# This part was modified to loop as many times as the value of k for validation and
classification
for x in range(21):
    # Modified by Afnan to accept CSV as encoded by vector creation script
    train_x = pd.read_csv("place_weighted_train" + str(x) + ".csv", sep=',', encoding = "ISO-
8859-1", engine='python')
    test_x = pd.read_csv("place_weighted_test" + str(x) + ".csv", sep=',', encoding = "ISO-8859-
1", engine='python')


    canada, uk = "", "" # Can specify which and how many classes you want to test

    for i, row in train_x.iterrows():
        a, t = row['country'], row['text']
        if a == 'canada':
            canada += " " + t.lower()
        elif a == 'uk':
            uk += " " + t.lower()

    canada = word_tokenize(canada)
    uk = word_tokenize(uk)

    c_canada, c_uk = create_dist(canada),  create_dist(uk)

    dist = {'canada': c_canada, 'uk': c_uk}
    nBS = NaiveBayes(dist)


    submission = predictions(test_x, nBS)
    submission.to_csv('results_place_weighted' + str(x) + '.csv', index=False)
```

Sample output for Naïve Bayes Classifier:

id	canada	uk	id	canada	uk
kuwait	1.733e-320	1	ussr	0.014535	0.985465
canada	1	0	syria	0.00708	0.99292
egypt	1	4.69E-23	mozambique	0.00191	0.99809
israel	1	1.67E-32	sudan	0.000551	0.999449
philippines	1	3.71E-31	australia	0.000214	0.999786
south-africa	1	2.12E-50	ghana	7.33E-06	0.999993
uganda	1	4.28E-25	morocco	1.85E-06	0.999998
usa	1	0	sri-lanka	6.69E-07	0.999999
venezuela	1	9.74E-18	norway	3.61E-08	1
ecuador	1	1.14E-15	lebanon	1.04E-09	1
bangladesh	1	1.15E-14	argentina	2.99E-10	1
zimbabwe	1	4.51E-14	greece	5.25E-12	1
yemen-demo-republic	1	1.39E-11	oman	1.31E-14	1
bolivia	1	2.98E-11	mexico	3.85E-17	1
taiwan	1	4.97E-10	cuba	3.07E-17	1
turkey	1	1.81E-09	pakistan	7.86E-18	1
zambia	0.999999	5.12E-07	south-korea	5.11E-18	1
peru	0.999999	5.56E-07	china	5.40E-19	1
fiji	0.999999	1.47E-06	indonesia	7.64E-23	1
finland	0.999998	2.04E-06	ivory-coast	1.93E-24	1
brunei	0.999995	4.76E-06	new-zealand	4.44E-26	1
djibouti	0.999994	6.43E-06	sweden	4.19E-29	1
ethiopia	0.999994	6.43E-06	india	3.65E-29	1
lesotho	0.999994	6.43E-06	czechoslovakia	7.87E-30	1
mauritius	0.999994	6.43E-06	denmark	2.17E-31	1
rwanda	0.999994	6.43E-06	dominican-republic	2.66E-32	1
somalia	0.999994	6.43E-06	nigeria	2.77E-33	1
swaziland	0.999994	6.43E-06	poland	1.45E-35	1
tanzania	0.999977	2.35E-05	kenya	9.47E-36	1
uruguay	0.999909	9.08E-05	libya	1.21E-37	1
bahamas	0.999776	0.000224	el-salvador	1.30E-41	1
malawi	0.999319	0.000681	costa-rica	1.10E-45	1
chile	0.999032	0.000968	spain	3.24E-49	1
yugoslavia	0.992157	0.007843	thailand	1.20E-49	1
niger	0.96917	0.03083	algeria	1.26E-50	1
jamaica	0.239754	0.760246	portugal	2.65E-53	1
zaire	0.228678	0.771322	nicaragua	9.61E-54	1

suriname	0.11708	0.88292	italy	9.02E-58	1
hungary	0.041451	0.958549	papua-new-guinea	1.18E-59	1
brazil	2.57E-71	1	cyprus	1.00E-64	1
switzerland	1.09E-72	1			
singapore	6.05E-76	1			
austria	3.39E-76	1			
uae	3.44E-82	1			
qatar	1.30E-88	1			
malaysia	8.36E-91	1			
colombia	7.54E-117	1			
hong-kong	6.20E-136	1			
bahrain	7.64E-140	1			
belgium	1.81E-171	1			
saudi-arabia	7.59E-178	1			
netherlands	3.52E-227	1			
france	4.13E-228	1			
luxembourg	7.06E-230	1			
iraq	1.39E-271	1			
iran	0	1			
japan	0	1			
uk	0	1			
west-germany	0	1			

Source Code Citations:

Trie Source Code: https://towardsdatascience.com/implementing-a-trie-data-structure-in-python-in-less-than-100-lines-of-code-a877ea23c1a1

Naive Bayes Helper Code: https://www.kaggle.com/antmarakis/word-count-and-naive-bayes/notebook