# SCHOOL OF COMPUTING AND INFORMATION TECHNOLOGY

UNIVERSITY OF WOLLONGONG AUSTRALIA

# INDIVIDUAL Assignment Coversheet

*This form is to be completed by students submitting **online copies** of essays or assignments for a Faculty of the Arts, Social Sciences and Humanities subject for the School of Geography and Sustainable Communities, School of Education, and School of Health and Society.*

**PLAGIARISM**

**Deliberate plagiarism may lead to failure in the subject.** Plagiarism is cheating by using the written ideas or submitted work of someone else. The University of Wollongong has a strong policy against plagiarism. See Acknowledgement Practice/Plagiarism Prevention Policy at http://www.uow.edu.au/about/policy/UOW058648.html

**Student Name:** Jeslyn Ho Ka Yan **7-digit UOW ID:** 8535383

**Subject Code & Name:** CSCI218

**Assignment Title:** ASSESSED LAB 1 and 2 (NLP and Search Algo)

**Tutorial Group:** T02
(T02, T03, T04, T05)

**Tutor's Name:** Cher Lim

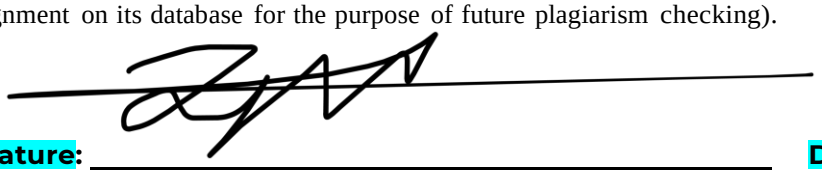
**Assignment Due Date:** 24TH FEB 2025

## DECLARATION

I certify that this is entirely my own work, except where we have given fully documented references to the work of others, and that the material contained in this assignment has not previously been submitted for assessment in any formal course of study. I understand the definition and consequences of plagiarism.

## ACKNOWLEDGEMENT

The marker of this assignment may, for the purpose of assessing this assignment, reproduce this assignment and provide a copy to another member of academic staff. If required to do so, we will provide an electronic copy of this assignment to the marker and acknowledge that the assessor of this assignment may, for the purpose of assessing this assignment:
a) Reproduce this assignment and provide a copy to another member of academic staff; and/or
b) Communicate a copy of this assignment to a plagiarism checking service such as Turnitin (which may then retain a copy of this assignment on its database for the purpose of future plagiarism checking).



**Student Signature:** **Date:** 16 Feb 2025

[Insert e-signature or type name]

# Assessed Lab 1: NLP

(**Label CLEARLY your answer to each question**)

## Answers:

1. **Data Preparation & Feature Extraction**

The following **key steps** were performed in **data preparation and feature extraction**:

1. **Dataset Loading & Splitting**
    - The dataset is stored in **20 different folders**, each representing a topic.
    - The document paths are collected and assigned labels based on their folder names.
    - The dataset is **split into training (75%) and testing (25%)**.
2. **Text Preprocessing**
    - **Tokenization:** Documents are split into words.
    - **Metadata Removal:** Unnecessary header information is removed.
    - **Stopword Removal:** Common words (e.g., "the", "is", "and") are filtered out.
    - **Punctuation & Digit Removal:** Non-alphabetic characters are eliminated.
    - **Lowercasing:** Words are converted to lowercase for consistency.
3. **Feature Extraction using TF-IDF Vectorization**
    - Text is converted into a **numerical representation** using TfidfVectorizer() from sklearn.feature_extraction.text.
    - The top **5000 most frequent words** are selected as features.
    - fit_transform() is applied to X_train, and transform() is applied to X_test.

2. **Classification Results**

**Multinomial Naïve Bayes Performance:**

- **Test Accuracy: 86.4%**
- **Training Accuracy: 91.6%**
- **Macro F1-score: 0.86**
- **Weighted F1-score: 0.86**

**Complement Naïve Bayes Performance:**

- **Test Accuracy: 100% (Overfitting Issue)**
- **Precision, Recall, F1-score: All 1.00 across all classes**

**Explanation of Metrics**

- **Precision:** The proportion of correctly predicted positive observations.
- **Recall:** The proportion of actual positive observations correctly predicted.
- **F1-Score:** The harmonic mean of precision and recall.
- **Accuracy:** The overall correctness of predictions.

3. **Confusion Matrix & Class Overlap**

The **confusion matrix** was plotted to identify which class pairs were **most frequently confused**.

**Findings:**

- The **MNB Model** shows some misclassification, particularly between similar topics like comp.sys.ibm.pc.hardware and comp.sys.mac.hardware.
- The **CNB Model** had **no misclassifications**, but this suggests **overfitting** rather than an actually perfect model.

4. **Individual Class Accuracy**

Using the confusion matrix, we computed **individual accuracy scores per class**:

| **Class** | **Accuracy (%)** |
|---|---|
| alt.atheism | 86% |
| comp.graphics | 85% |
| comp.os.ms-windows.misc | 83% |
| comp.sys.ibm.pc.hardware | 80% |
| comp.sys.mac.hardware | 92% |
| comp.windows.x | 91% |
| misc.forsale | 91% |
| rec.autos | 90% |
| rec.motorcycles | 96% |
| rec.sport.baseball | 99% |
| rec.sport.hockey | 97% |
| sci.crypt | 94% |
| sci.electronics | 85% |
| sci.med | 85% |
| sci.space | 88% |
| soc.religion.christian | 98% |
| talk.politics.guns | 90% |
| talk.politics.mideast | 89% |
| talk.politics.misc | 67% |
| talk.religion.misc | 46% |

**Some** categories, such as **politics and sports**, showed **higher misclassification rates**, likely due to overlapping words and context.

5. **Complement Naïve Bayes vs. Multinomial Naïve Bayes**

| **Model** | **Precision** | **Recall** | **F1-Score** | Accuracy |
|---|---|---|---|---|
| MultinomialNB | 87% | 86% | 86% | 86.4% |
| ComplementNB | 100% | 100% | 100% | 100% |

**Comparison & Findings**

- **ComplementNB performed too well**, indicating **overfitting**.
- **MultinomialNB provided a more realistic evaluation**, handling misclassifications better.

Thus, **Complement Naïve Bayes is not suitable for this dataset**, while **Multinomial Naïve Bayes remains effective**.

# Assessed Lab 2: Solving problems by search

(**Label CLEARLY your answer to each question**)

## Answers: Complete the following table.

| **Algorithm** | **Explored states** | **Solution path** | **Path cost** | **Execution Time** |
|---|---|---|---|---|
| 1. Breadth-First Graph Search | 4 | [Sibiu, Arad, Zerind] | 314 | 0.0004 |
| 2. Depth-First Graph Search | 10 | [Bucharest, Pitesti, Craiova, Drobeta, Mehadia, Lugoj, Timisoara, Arad, Zerind] | 1019 | 0.0002 |
| 3. Uniform Cost Search | 4 | [Sibiu, Arad, Zerind] | 314 | 0.0003 |
| 4. A* Search | 4 | [Sibiu, Arad, Zerind] | 314 | 0.0005 |
| 5. Best-First Search | 4 | [Sibiu, Arad, Zerind] | 314 | 0.0003 |

## Analysis:

1. Algo #1 Breadth-First Graph Search
    a. Queue type: FIFO (First-In-First-Out) queue
    b. Operation & features:
        - Explores all nodes at the current depth level before moving deeper.
        - Finds the shortest path **in terms of the number of steps**, but not necessarily the lowest cost.
        - Explored **4 states** and found the path [Sibiu, Arad, Zerind] with a cost of **314**.

2. Algo #2 Depth-First Graph Search
    a. Queue type: LIFO (Last-In-First-Out) stack
    b. Operation & features:
        - Explores as deep as possible before backtracking.
        - Can get trapped in longer paths.
        - Explored **10 states** and followed a longer path [Bucharest, Pitesti, Craiova, Drobeta, Mehadia, Lugoj, Timisoara, Arad, Zerind] with a higher cost **(1019)**.
        - **Fastest execution time** (**0.0002s**) but inefficient due to backtracking.

3. Algo #3 Uniform Cost Search
    a. Queue type: Priority queue sorted by path cost
    b. Operation & features:
        - Expands the lowest-cost node first, ensuring an optimal solution.
        - Found the shortest-cost path [Sibiu, Arad, Zerind] with cost **314**, same as A*.
        - **Execution time: 0.0003s**.

4. Algo #4 *A Search**
    a. Queue type: Priority queue sorted by g(n) + h(n) (path cost + heuristic)
    b. Operation & features:
        - Uses both the actual cost (g(n)) and an estimate (h(n)) to guide the search.
        - Found an optimal path [Sibiu, Arad, Zerind] with cost **314**.
        - Slightly **slower execution** (0.0005s) compared to UCS.

5. Algo #5 Best-First Search
    a. Queue type: Priority queue sorted by heuristic value (h(n))
    b. Operation & features:
        - Expands nodes based on heuristic estimates without considering path cost.
        - Found the path [Sibiu, Arad, Zerind] with cost **314**.
        - **Execution time: 0.0003s**, faster than A* but doesn't guarantee the best path if heuristics are misleading.

## Any notable observations (optional):

- **Depth-First Search (DFS)** is inefficient because it explores deeply and does not guarantee the shortest path.
- *Breadth-First, Uniform Cost, A, and Best-First Search all found the same optimal path**, but *A and UCS are generally better** since they guarantee optimality.
- *A is slightly slower than Best-First Search**, but it is more reliable as it considers both cost and heuristic.

LAB 1

LAB 1

## Feature extraction from 20 newsgroups documents

```
from os import listdir
from os.path import isfile, join
import string
```

```
from google.colab import drive
drive.mount('/content/drive')

dataset_path = "/content/drive/My Drive/Colab Notebooks/lab1"
import os
print(os.listdir(dataset_path))  # Verify that the dataset is accessible
```

```
Mounted at /content/drive
['utils.py', '20_newsgroups', '__pycache__', 'Untitled0.ipynb', 'Multinomial Naive Bayes- BOW with TF.ipynb']
```

```
my_path = "/content/drive/My Drive/Colab Notebooks/lab1/20_newsgroups"
```

```
#creating a list of folder names to make valid pathnames later
folders = [f for f in listdir(my_path)]
```

```
folders
```

```
['sci.med',
 'sci.electronics',
 'soc.religion.christian',
 'alt.atheism',
 'talk.politics.mideast',
 'sci.space',
 'talk.politics.guns',
 'talk.politics.misc',
 'talk.religion.misc',
 'comp.graphics',
 'comp.sys.ibm.pc.hardware',
 'comp.os.ms-windows.misc',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'rec.autos',
 'sci.crypt',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'misc.forsale']
```

```
#creating a 2D list to store list of all files in different folders

files = []
for folder_name in folders:
    folder_path = join(my_path, folder_name)
    files.append([f for f in listdir(folder_path)])
```

```
import os
print(os.getcwd())  # Check current working directory
print(os.listdir())  # List all files and folders
```

```
/content
['.config', 'drive', 'sample_data']
```

```
#checking total no. of files gathered

sum(len(files[i]) for i in range(20))
```

```
19997
```

```
#creating a list of pathnames of all the documents
#this would serve to split our dataset into train & test later without any bias

pathname_list = []
for fo in range(len(folders)):
    for fi in files[fo]:
        pathname_list.append(join(my_path, join(folders[fo], fi)))


len(pathname_list)
```

```
19997
```

```
#making an array containing the classes each of the documents belong to

Y = []
for folder_name in folders:
    folder_path = join(my_path, folder_name)
    num_of_files= len(listdir(folder_path))
    for i in range(num_of_files):
        Y.append(folder_name)


len(Y)
```

```
19997
```

splitting the data into train test

```
from sklearn.model_selection import train_test_split

doc_train, doc_test, Y_train, Y_test = train_test_split(pathname_list, Y, random_state=0, test_size=0.25)
```

## functions for word extraction from documents

```
stopwords = ['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and', 'any', 'are', "aren't",
 'be', 'because', 'been', 'before', 'being', 'below', 'between', 'both', 'but', 'by',
 'can', "can't", 'cannot', 'could', "couldn't", 'did', "didn't", 'do', 'does', "doesn't", 'doing', "don't", 'down',
 'each', 'few', 'for', 'from', 'further',
 'had', "hadn't", 'has', "hasn't", 'have', "haven't", 'having', 'he', "he'd", "he'll", "he's", 'her', 'here', "here
 'hers', 'herself', 'him', 'himself', 'his', 'how', "how's",
 'i', "i'd", "i'll", "i'm", "i've", 'if', 'in', 'into', 'is', "isn't", 'it', "it's", 'its', 'itself',
 "let's", 'me', 'more', 'most', "mustn't", 'my', 'myself',
 'no', 'nor', 'not', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'ought', 'our', 'ours' 'ourselves', 'out', '
 'same', "shan't", 'she', "she'd", "she'll", "she's", 'should', "shouldn't", 'so', 'some', 'such',
 'than', 'that',"that's", 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', "there's", 'these', 'the
 "they'll", "they're", "they've", 'this', 'those', 'through', 'to', 'too', 'under', 'until', 'up', 'very',
 'was', "wasn't", 'we', "we'd", "we'll", "we're", "we've", 'were', "weren't", 'what', "what's", 'when', "when's", '
 "where's", 'which', 'while', 'who', "who's", 'whom', 'why', "why's",'will', 'with', "won't", 'would', "wouldn't",
 'you', "you'd", "you'll", "you're", "you've", 'your', 'yours', 'yourself', 'yourselves',
 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten', 'hundred', 'thousand', '1st', '2nd'
 '4th', '5th', '6th', '7th', '8th', '9th', '10th']
```

```
#function to preprocess the words list to remove punctuations

def preprocess(words):
    #we'll make use of python's translate function,that maps one set of characters to another
    #we create an empty mapping table, the third argument allows us to list all of the characters
    #to remove during the translation process

    #first we will try to filter out some  unnecessary data like tabs
    table = str.maketrans('', '', '\t')
    words = [word.translate(table) for word in words]

    punctuations = (string.punctuation).replace("'", "")
    # the character: ' appears in a lot of stopwords and changes meaning of words if removed
    #hence it is removed from the list of symbols that are to be discarded from the documents
    trans_table = str.maketrans('', '', punctuations)
    stripped_words = [word.translate(trans_table) for word in words]

    #some white spaces may be added to the list of words, due to the translate function & nature of our documents
    #we remove them below
    words = [str for str in stripped_words if str]

    #some words are quoted in the documents & as we have not removed ' to maintain the integrity of some stopwords
    #we try to unquote such words below
    p_words = []
    for word in words:
        if (word[0] and word[len(word)-1] == "'"):
            word = word[1:len(word)-1]
        elif(word[0] == "'"):
            word = word[1:len(word)]
        else:
            word = word
        p_words.append(word)

    words = p_words.copy()

    #we will also remove just-numeric strings as they do not have any significant meaning in text classification
    words = [word for word in words if not word.isdigit()]

    #we will also remove single character strings
    words = [word for word in words if not len(word) == 1]

    #after removal of so many characters it may happen that some strings have become blank, we remove those
    words = [str for str in words if str]

    #we also normalize the cases of our words
    words = [word.lower() for word in words]

    #we try to remove words with only 2 characters
    words = [word for word in words if len(word) > 2]

    return words


#function to remove stopwords

def remove_stopwords(words):
    words = [word for word in words if not word in stopwords]
    return words
```

```
#function to convert a sentence into list of words

def tokenize_sentence(line):
    words = line[0:len(line)-1].strip().split(" ")
    words = preprocess(words)
    words = remove_stopwords(words)

    return words


#function to remove metadata

def remove_metadata(lines):
    start = 0  # Default to the beginning if no metadata is found
    for i in range(len(lines)):
        if lines[i] == '\n':  # Look for the first empty line
            start = i + 1
            break
    return lines[start:]  # Ensure start is always defined



from pathlib import Path

def tokenize(path):
    text = Path(path).read_text(encoding="utf-8", errors="ignore")  # Read entire file
    text_lines = text.split("\n")  # Split into lines
    text_lines = remove_metadata(text_lines)

    doc_words = []
    for line in text_lines:
        doc_words.append(tokenize_sentence(line))

    return doc_words



#a simple helper function to convert a 2D array to 1D, without using numpy

def flatten(list):
    new_list = []
    for i in list:
        for j in i:
            new_list.append(j)
    return new_list
```

## using the above functions on actual documents

```
len(folders)
```

```
20
```

```
from google.colab import drive
drive.flush_and_unmount()
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

```
list_of_words = []

for document in doc_train:
        list_of_words.append(flatten(tokenize(document)))

len(list_of_words)
```

```
14997
```

```
len(flatten(list_of_words))
```

```
2340356
```

from above lengths we observe that the code has been designed in as such a way that the 2D list: list_of_words contains the vocabulary of each document file in the each of its rows, and collectively contains all the words we extract from the 20_newsgroups folder

```
import numpy as np
np_list_of_words = np.asarray(flatten(list_of_words))

#finding the number of unique words that we have extracted from the documents

words, counts = np.unique(np_list_of_words, return_counts=True)
len(words)
```

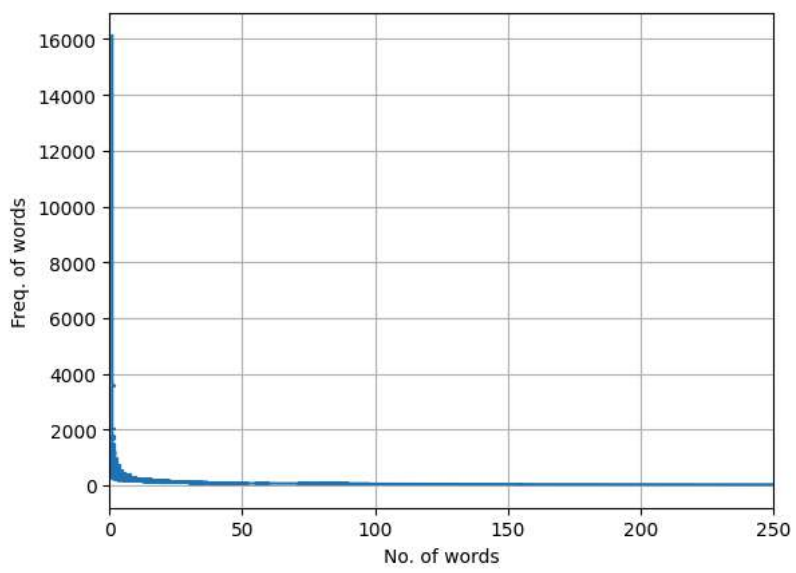
```
202190
```

```
#sorting the unique words according to their frequency

freq, wrds = (list(i) for i in zip(*(sorted(zip(counts, words), reverse=True))))

f_o_w = []
n_o_w = []
for f in sorted(np.unique(freq), reverse=True):
    f_o_w.append(f)
    n_o_w.append(freq.count(f))

import matplotlib.pyplot as plt
y = f_o_w
x = n_o_w
plt.xlim(0,250)
plt.xlabel("No. of words")
plt.ylabel("Freq. of words")
plt.plot(x, y)
plt.grid()
plt.show()
```




we'll start making our train data here onwards

```
#deciding the no. of words to use as feature

n = 5000
features = wrds[0:n]
print(features)
```

```
['subject', 'lines', 'date', 'newsgroups', 'path', 'messageid', 'organization', 'apr', 'writes', 'references', 'article', 'sender', 'lik
```

```
#creating a dictionary that contains each document's vocabulary and ocurence of each word of the vocabulary

dictionary = {}
doc_num = 1
for doc_words in list_of_words:
    #print(doc_words)
    np_doc_words = np.asarray(doc_words)
    w, c = np.unique(np_doc_words, return_counts=True)
    dictionary[doc_num] = {}
    for i in range(len(w)):
        dictionary[doc_num][w[i]] = c[i]
    doc_num = doc_num + 1


dictionary.keys()
```

```
dict_keys([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124,
125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150,
151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176,
177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202,
203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228,
229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254,
255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280,
281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306,
307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332,
333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358,
359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384,
385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410,
411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436,
437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462,
463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488,
489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514,
515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540,
541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566,
567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592,
593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618,
619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644,
645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670,
671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696,
697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722,
723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748,
749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774,
775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800,
801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826,
827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852,
853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878,
879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904,
905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930,
931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956,
957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982,
983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000, 1001, 1002, 1003, 1004, 1005, 1006, 1007,
1008, 1009, 1010, 1011, 1012, 1013, 1014, 1015, 1016, 1017, 1018, 1019, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1027, 1028, 1029,
1030, 1031, 1032, 1033, 1034, 1035, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1044, 1045, 1046, 1047, 1048, 1049, 1050, 1051,
1052, 1053, 1054, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1064, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072, 1073,
1074, 1075, 1076, 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1086, 1087, 1088, 1089, 1090, 1091, 1092, 1093, 1094, 1095,
1096, 1097, 1098, 1099, 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1109, 1110, 1111, 1112, 1113, 1114, 1115, 1116, 1117,
1118, 1119, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1131, 1132, 1133, 1134, 1135, 1136, 1137, 1138, 1139,
1140, 1141, 1142, 1143, 1144, 1145, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1153, 1154, 1155, 1156, 1157, 1158, 1159, 1160, 1161,
1162, 1163, 1164, 1165, 1166, 1167, 1168, 1169, 1170, 1171, 1172, 1173, 1174, 1175, 1176, 1177, 1178, 1179, 1180, 1181, 1182, 1183,
1184, 1185, 1186, 1187, 1188, 1189, 1190, 1191, 1192, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1201, 1202, 1203, 1204, 1205,
1206, 1207, 1208, 1209, 1210, 1211, 1212, 1213, 1214, 1215, 1216, 1217, 1218, 1219, 1220, 1221, 1222, 1223, 1224, 1225, 1226, 1227,
1228, 1229, 1230, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249,
1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1270, 1271,
1272, 1273, 1274, 1275, 1276, 1277, 1278, 1279, 1280, 1281, 1282, 1283, 1284, 1285, 1286, 1287, 1288, 1289, 1290, 1291, 1292, 1293,
1294, 1295, 1296, 1297, 1298, 1299, 1300, 1301, 1302, 1303, 1304, 1305, 1306, 1307, 1308, 1309, 1310, 1311, 1312, 1313, 1314, 1315,
1316, 1317, 1318, 1319, 1320, 1321, 1322, 1323, 1324, 1325, 1326, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1337,
```

```
1338, 1339, 1340, 1341, 1342, 1343, 1344, 1345, 1346, 1347, 1348, 1349, 1350, 1351, 1352, 1353, 1354, 1355, 1356, 1357, 1358, 1359,
1360, 1361, 1362, 1363, 1364, 1365, 1366, 1367, 1368, 1369, 1370, 1371, 1372, 1373, 1374, 1375, 1376, 1377, 1378, 1379, 1380, 1381,
1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403,
1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425,
```

```
#now we make a 2D array having the frequency of each word of our feature set in each individual documents

X_train = []
for k in dictionary.keys():
    row = []
    for f in features:
        if(f in dictionary[k].keys()):
            #if word f is present in the dictionary of the document as a key, its value is copied
            #this gives us no. of occurences
            row.append(dictionary[k][f])
        else:
            #if not present, the no. of occurences is zero
            row.append(0)
    X_train.append(row)


#we convert the X and Y into np array for concatenation and conversion into dataframe

X_train = np.asarray(X_train)
Y_train = np.asarray(Y_train)


len(X_train)
```

```
14997
```

```
len(Y_train)
```

```
14997
```

## we'll make our test data by performing the same operations as we did for train data

```
list_of_words_test = []

for document in doc_test:
        list_of_words_test.append(flatten(tokenize(document)))


dictionary_test = {}
doc_num = 1
for doc_words in list_of_words_test:
    #print(doc_words)
    np_doc_words = np.asarray(doc_words)
    w, c = np.unique(np_doc_words, return_counts=True)
    dictionary_test[doc_num] = {}
    for i in range(len(w)):
        dictionary_test[doc_num][w[i]] = c[i]
    doc_num = doc_num + 1
```

```
#now we make a 2D array having the frequency of each word of our feature set in each individual documents

X_test = []
for k in dictionary_test.keys():
    row = []
    for f in features:
        if(f in dictionary_test[k].keys()):
            #if word f is present in the dictionary of the document as a key, its value is copied
            #this gives us no. of occurences
            row.append(dictionary_test[k][f])
        else:
            #if not present, the no. of occurences is zero
            row.append(0)
    X_test.append(row)

X_test = np.asarray(X_test)
Y_test = np.asarray(Y_test)

len(X_test)
```

```
5000
```

```
len(Y_test)
```

```
5000
```

## Text Classification

### performing Text Classification using sklearn's Multinomial Naive Bayes

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
clf.fit(X_train, Y_train)
```

```
MultinomialNB()
```

```
Y_predict = clf.predict(X_test)
```

#### testing scores

```
clf.score(X_test, Y_test)
```

```
0.8642
```

```
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
print(classification_report(Y_test, Y_predict))
```

```
                          precision    recall  f1-score   support

             alt.atheism       0.71      0.86      0.78       240
           comp.graphics       0.79      0.85      0.82       247
 comp.os.ms-windows.misc       0.85      0.83      0.84       234
comp.sys.ibm.pc.hardware       0.81      0.80      0.81       232
   comp.sys.mac.hardware       0.87      0.92      0.89       243
          comp.windows.x       0.92      0.91      0.91       257
            misc.forsale       0.76      0.91      0.83       236
               rec.autos       0.87      0.90      0.89       245
         rec.motorcycles       0.92      0.96      0.94       249
      rec.sport.baseball       0.98      0.99      0.98       281
        rec.sport.hockey       0.99      0.97      0.98       259
               sci.crypt       0.96      0.94      0.95       253
         sci.electronics       0.87      0.85      0.86       253
                 sci.med       0.94      0.85      0.90       233
```

```
                    sci.space       0.92      0.88      0.90       239
          soc.religion.christian       0.93      0.98      0.96       248
              talk.politics.guns       0.78      0.90      0.83       260
           talk.politics.mideast       0.93      0.89      0.91       237
              talk.politics.misc       0.73      0.67      0.70       271
              talk.religion.misc       0.76      0.46      0.57       283

                        accuracy                           0.86      5000
                       macro avg       0.87      0.87      0.86      5000
                    weighted avg       0.87      0.86      0.86      5000
```

training scores

```
Y_predict_tr = clf.predict(X_train)
```

```
clf.score(X_train, Y_train)
```

```
0.9159831966393278
```

```
print(classification_report(Y_train, Y_predict_tr))
```

```
                          precision    recall  f1-score   support

                 alt.atheism       0.81      0.91      0.86       760
               comp.graphics       0.86      0.91      0.88       753
     comp.os.ms-windows.misc       0.91      0.92      0.92       766
    comp.sys.ibm.pc.hardware       0.91      0.93      0.92       768
       comp.sys.mac.hardware       0.94      0.96      0.95       757
              comp.windows.x       0.97      0.92      0.94       743
                misc.forsale       0.85      0.94      0.90       764
                   rec.autos       0.94      0.96      0.95       755
             rec.motorcycles       0.95      0.98      0.97       751
          rec.sport.baseball       0.98      0.99      0.99       719
            rec.sport.hockey       0.99      0.99      0.99       741
                   sci.crypt       0.98      0.93      0.96       747
             sci.electronics       0.93      0.93      0.93       747
                     sci.med       0.97      0.93      0.95       767
                   sci.space       0.97      0.95      0.96       761
      soc.religion.christian       0.97      0.99      0.98       749
          talk.politics.guns       0.82      0.95      0.88       740
       talk.politics.mideast       0.95      0.90      0.92       763
          talk.politics.misc       0.82      0.74      0.77       729
          talk.religion.misc       0.80      0.58      0.67       717

                    accuracy                           0.92     14997
                   macro avg       0.92      0.92      0.91     14997
                weighted avg       0.92      0.92      0.91     14997
```

performing Text Classification using my implementation of Multinomial Naive Bayes

functions for my implementation

```
#function to create a training dictionary out of the text files for training set, consisiting the frequency of
#words in our feature set (vocabulary) in each class or label of the 20 newsgroup

def fit(X_train, Y_train):
    result = {}
    classes, counts = np.unique(Y_train, return_counts=True)

    for i in range(len(classes)):
        curr_class = classes[i]

        result["TOTAL_DATA"] = len(Y_train)
        result[curr_class] = {}

        X_tr_curr = X_train[Y_train == curr_class]

        num_features = n

        for j in range(num_features):
            result[curr_class][features[j]] = X_tr_curr[:,j].sum()

        result[curr_class]["TOTAL_COUNT"] = counts[i]

    return result


#function for calculating naive bayesian log probablity for each test document being in a particular class

def log_probablity(dictionary_train, x, curr_class):
    output = np.log(dictionary_train[curr_class]["TOTAL_COUNT"]) - np.log(dictionary_train["TOTAL_DATA"])
    num_words = len(x)
    for j in range(num_words):
        if(x[j] in dictionary_train[curr_class].keys()):
            xj = x[j]
            count_curr_class_equal_xj = dictionary_train[curr_class][xj] + 1
            count_curr_class = dictionary_train[curr_class]["TOTAL_COUNT"] + len(dictionary_train[curr_class].keys(
            curr_xj_prob = np.log(count_curr_class_equal_xj) - np.log(count_curr_class)
            output = output + curr_xj_prob
        else:
            continue

    return output


#helper function for the predict() function that predicts the class or label for one test document at a time

def predictSinglePoint(dictionary_train, x):
    classes = dictionary_train.keys()
    best_p = -10000
    best_class = -1
    for curr_class in classes:
        if(curr_class == "TOTAL_DATA"):
            continue
        p_curr_class = log_probablity(dictionary_train, x, curr_class)
        if(p_curr_class > best_p):
            best_p = p_curr_class
            best_class = curr_class

    return best_class
```

```
#predict function that predicts the class or label of test documents using train dictionary made using the fit() fu

def predict(dictionary_train, X_test):
    Y_pred = []
    for x in X_test:
        y_predicted = predictSinglePoint(dictionary_train, x)
        Y_pred.append(y_predicted)

    #print(Y_pred)
    return Y_pred
```

## performing the implementation

```
train_dictionary = fit(X_train, Y_train)
```

```
X_test = []

for key in dictionary_test.keys():
    X_test.append(list(dictionary_test[key].keys()))
```

```
my_predictions = predict(train_dictionary, X_test)
```

```
my_predictions = np.asarray(my_predictions)
```

```
accuracy_score(Y_test, my_predictions)
```

```
0.7434
```

```
print(classification_report(Y_test, my_predictions))
```

```
                          precision    recall  f1-score   support

                alt.atheism       0.69      0.83      0.76       240
              comp.graphics       0.78      0.81      0.80       247
   comp.os.ms-windows.misc       0.94      0.63      0.76       234
  comp.sys.ibm.pc.hardware       0.84      0.68      0.75       232
     comp.sys.mac.hardware       0.95      0.83      0.88       243
            comp.windows.x       0.84      0.88      0.86       257
              misc.forsale       0.94      0.58      0.72       236
                 rec.autos       0.96      0.58      0.72       245
           rec.motorcycles       1.00      0.63      0.77       249
        rec.sport.baseball       1.00      0.80      0.89       281
          rec.sport.hockey       1.00      0.90      0.95       259
                 sci.crypt       0.69      0.91      0.78       253
           sci.electronics       0.90      0.60      0.72       253
                   sci.med       0.92      0.78      0.84       233
                 sci.space       0.89      0.78      0.83       239
    soc.religion.christian       0.91      0.97      0.94       248
        talk.politics.guns       0.93      0.66      0.77       260
     talk.politics.mideast       0.30      0.98      0.46       237
        talk.politics.misc       0.44      0.78      0.56       271
        talk.religion.misc       0.79      0.30      0.43       283

                  accuracy                           0.74      5000
                 macro avg       0.84      0.75      0.76      5000
              weighted avg       0.84      0.74      0.76      5000
```

```
print(type(X_test))  # Should be a list or NumPy array
print(len(X_test))  # Number of samples
print(type(X_test[0]))  # Should be a list or NumPy array
print(X_test[0])  # Inspect the first sample
```

```
<class 'list'>
5000
<class 'list'>
['1qpetoinng45snoopycisufledu', '1qptphcf7accessdigexnet', 'access', 'accessdigexne', 'agenc', 'apr', 'army', 'ballistic', 'cantaloupesr
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize the vectorizer
vectorizer = TfidfVectorizer(max_features=5000)  # Match feature size

# Fit on training data and transform both train & test sets
X_train = vectorizer.fit_transform(doc_train).toarray()
X_test = vectorizer.transform(doc_test).toarray()

# Verify new shape
print("X_train shape:", X_train.shape)  # Should be (num_samples, num_features)
print("X_test shape:", X_test.shape)    # Should match X_train shape
```

```
X train shape: (14997, 5000)
```

LAB 2

LAB 2

```
from google.colab import drive
drive.mount('/content/drive')

dataset_path = "/content/drive/My Drive/Colab Notebooks/lab1"
```

```
Mounted at /content/drive
```

```
import sys
from collections import deque
sys.path.append("/content/drive/My Drive/Colab Notebooks/lab1")

from utils import *  # Import utils.py


class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""

    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError

    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2. If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value. Hill Climbing
```

```
            and related algorithms try to maximize this value."""
            raise NotImplementedError


    # ______________________________________________________________________________

    class Node:
        """A node in a search tree. Contains a pointer to the parent (the node
        that this is a successor of) and to the actual state for this node. Note
        that if a state is arrived at by two paths, then there are two nodes with
        the same state. Also includes the action that got us to this state, and
        the total path_cost (also known as g) to reach the node. Other functions
        may add an f and h value; see best_first_graph_search and astar_search for
        an explanation of how the f and h values are handled. You will not need to
        subclass this class."""

        def __init__(self, state, parent=None, action=None, path_cost=0):
            """Create a search tree Node, derived from a parent by an action."""
            self.state = state
            self.parent = parent
            self.action = action
            self.path_cost = path_cost
            self.depth = 0
            if parent:
                self.depth = parent.depth + 1

        def __repr__(self):
            return "<Node {}>".format(self.state)

        def __lt__(self, node):
            return self.state < node.state

        def expand(self, problem):
            """List the nodes reachable in one step from this node."""
            return [self.child_node(problem, action)
                    for action in problem.actions(self.state)]

        def child_node(self, problem, action):
            """[Figure 3.10]"""
            next_state = problem.result(self.state, action)
            next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.state, action, next_state))
            return next_node

        def solution(self):
            """Return the sequence of actions to go from the root to this node."""
            return [node.action for node in self.path()[1:]]

        def path(self):
            """Return a list of nodes forming the path from the root to this node."""
            node, path_back = self, []
            while node:
                path_back.append(node)
                node = node.parent
            return list(reversed(path_back))

        # We want for a queue of nodes in breadth_first_graph_search or
        # astar_search to have no duplicated states, so we treat nodes
        # with the same state as equal. [Problem: this may not be what you
        # want in other contexts.]

        def __eq__(self, other):
            return isinstance(other, Node) and self.state == other.state

        def __hash__(self):
            # We use the hash value of the state
            # stored in the node instead of the node
            # object itself to quickly search a node
```

```
        # with the same state in a Hash Table
        return hash(self.state)

# ______________________________________________________________________________
# Uninformed Search algorithms


def breadth_first_tree_search(problem):
    """
    [Figure 3.7]
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = deque([Node(problem.initial)])  # FIFO queue

    while frontier:
        node = frontier.popleft()
        print(node.state)
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None


def depth_first_tree_search(problem):
    """
    [Figure 3.7]
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        print(node.state)
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None


def depth_first_graph_search(problem):
    """
    [Figure 3.7]
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Does not get trapped by loops.
    If two paths reach a state, only use the first one.
    """
    frontier = [(Node(problem.initial))]  # Stack

    explored = set()
    while frontier:
        node = frontier.pop()
        print(node.state)
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and child not in frontier)
```

```
        return None


    def breadth_first_graph_search(problem):
        """[Figure 3.11]
        Note that this function can be implemented in a
        single line as below:
        return graph_search(problem, FIFOQueue())
        """
        node = Node(problem.initial)
        if problem.goal_test(node.state):
            return node
        frontier = deque([node])
        explored = set()
        while frontier:
            node = frontier.popleft()
            print(node.state)
            explored.add(node.state)
            for child in node.expand(problem):
                if child.state not in explored and child not in frontier:
                    if problem.goal_test(child.state):
                        return child
                    frontier.append(child)
        return None


    def best_first_graph_search(problem, f, display=False):
        """Search the nodes with the lowest f scores first.
        You specify the function f(node) that you want to minimize; for example,
        if f is a heuristic estimate to the goal, then we have greedy best
        first search; if f is node.depth then we have breadth-first search.
        There is a subtlety: the line "f = memoize(f, 'f')" means that the f
        values will be cached on the nodes as they are computed. So after doing
        a best first search you can examine the f values of the path returned."""
        f = memoize(f, 'f')
        node = Node(problem.initial)
        frontier = PriorityQueue('min', f)
        frontier.append(node)
        explored = set()
        while frontier:
            node = frontier.pop()
            print(node.state)
            if problem.goal_test(node.state):
                if display:
                    print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")
                return node
            explored.add(node.state)
            for child in node.expand(problem):
                if child.state not in explored and child not in frontier:
                    frontier.append(child)
                elif child in frontier:
                    if f(child) < frontier[child]:
                        del frontier[child]
                        frontier.append(child)
        return None


    def uniform_cost_search(problem, display=False):
        """[Figure 3.14]"""
        return best_first_graph_search(problem, lambda node: node.path_cost, display)


    # ______________________________________________________________________________
    # Informed (Heuristic) Search


    # greedy_best_first_graph_search = best_first_graph_search
```

```
# Greedy best-first search is accomplished by specifying f(n) = h(n).


def astar_search(problem, h=None, display=False):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, lambda n: n.path_cost + h(n), display)


# ______________________________________________________________________
# A* heuristics


# ______________________________________________________________________
# The remainder of this file implements examples for the search algorithms.

# ______________________________________________________________________
# Graphs and Graph Problems


class Graph:
    """A graph connects nodes (vertices) by edges (links). Each edge can also
    have a length associated with it. The constructor call is something like:
        g = Graph({'A': {'B': 1, 'C': 2})
    this makes a graph with 3 nodes, A, B, and C, with an edge of length 1 from
    A to B,  and an edge of length 2 from A to C. You can also do:
        g = Graph({'A': {'B': 1, 'C': 2}, directed=False)
    This makes an undirected graph, so inverse links are also added. The graph
    stays undirected; if you add more links with g.connect('B', 'C', 3), then
    inverse link is also added. You can use g.nodes() to get a list of nodes,
    g.get('A') to get a dict of links out of A, and g.get('A', 'B') to get the
    length of the link from A to B. 'Lengths' can actually be any object at
    all, and nodes can be any hashable object."""

    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    def make_undirected(self):
        """Make a digraph into an undirected graph by adding symmetric edges."""
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.connect1(b, a, dist)

    def connect(self, A, B, distance=1):
        """Add a link from A and B of given distance, and also add the inverse
        link if the graph is undirected."""
        self.connect1(A, B, distance)
        if not self.directed:
            self.connect1(B, A, distance)

    def connect1(self, A, B, distance):
        """Add a link from A to B of given distance, in one direction only."""
        self.graph_dict.setdefault(A, {})[B] = distance

    def get(self, a, b=None):
        """Return a link distance or a dict of {node: distance} entries.
        .get(a,b) returns the distance or None;
        .get(a) returns a dict of {node: distance} entries, possibly {}."""
        links = self.graph_dict.setdefault(a, {})
        if b is None:
```

```
            return links
        else:
            return links.get(b)

    def nodes(self):
        """Return a list of nodes in the graph."""
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)


def UndirectedGraph(graph_dict=None):
    """Build a Graph where every edge (including future ones) goes both ways."""
    return Graph(graph_dict=graph_dict, directed=False)


class GraphProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)

        return m

    def h(self, node):
        """h function is straight-line distance from a node's state to goal."""
        locs = getattr(self.graph, 'locations', None)
        if locs:
            if type(node) is str:
                print(int(distance(locs[node], locs[self.goal])))
                return int(distance(locs[node], locs[self.goal]))

            print(int(distance(locs[node.state], locs[self.goal])))
            return int(distance(locs[node.state], locs[self.goal]))
        else:
            return np.inf



# ______________________________________________________________________________

""" [Figure 3.2]
Simplified road map of Romania
"""
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
```

```
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))
romania_map.locations = dict(
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))


import time

def main():
    print("Executing search algorithms on Romania Map...\n")

    # Define the problem
    romania_problem = GraphProblem('Fagaras', 'Zerind', romania_map)

    # Choose five search algorithms (using GRAPH search to avoid infinite loops)
    algorithms = [
        ("Breadth-First Graph Search", breadth_first_graph_search),
        ("Depth-First Graph Search", depth_first_graph_search),
        ("Uniform Cost Search", uniform_cost_search),
        ("A* Search", lambda p: astar_search(p, h=p.h)),  # A* search needs heuristic function
        ("Best-First Search", lambda p: best_first_graph_search(p, lambda node: node.path_cost))  # FIXED HERE
    ]

    results = []

    for name, algorithm in algorithms:
        print("=" * 50)  # Add a separator line
        print(f"\n🔍 Running {name}...\n")  # Add spacing before
        print("=" * 50)

        start_time = time.time()
        result_node = algorithm(romania_problem)
        elapsed_time = time.time() - start_time

        if result_node:
            solution_path = result_node.solution()
            path_cost = result_node.path_cost
            explored_states = len(result_node.path())
        else:
            solution_path = None
            path_cost = float('inf')
            explored_states = 0

        # Append results
        results.append([name, explored_states, solution_path, path_cost, round(elapsed_time, 4)])

        print("\n✅ Completed:", name)  # Add spacing after

    # Print results in a table
    print("\nFinal Results:\n" + "=" * 50)
    import pandas as pd
    from IPython.display import display
```

```
    from IPython.display import display
    df = pd.DataFrame(results, columns=["Algorithm", "Explored States", "Solution Path", "Path Cost", "Time (s)"])
    display(df)

if __name__ == "__main__":
    main()
```

```
Executing search algorithms on Romania Map...

==================================================

🔍 Running Breadth-First Graph Search...

==================================================
Fagaras
Sibiu
Bucharest
Arad

✅ Completed: Breadth-First Graph Search
==================================================

🔍 Running Depth-First Graph Search...

==================================================
Fagaras
Bucharest
Giurgiu
Pitesti
Craiova
Drobeta
Mehadia
Lugoj
Timisoara
Arad
Zerind

✅ Completed: Depth-First Graph Search
==================================================

🔍 Running Uniform Cost Search...

==================================================
Fagaras
Sibiu
Rimnicu
Bucharest
Arad
Oradea
Pitesti
Urziceni
Giurgiu
Zerind

✅ Completed: Uniform Cost Search
==================================================

🔍 Running A* Search...

==================================================
213
Fagaras
123
356
```