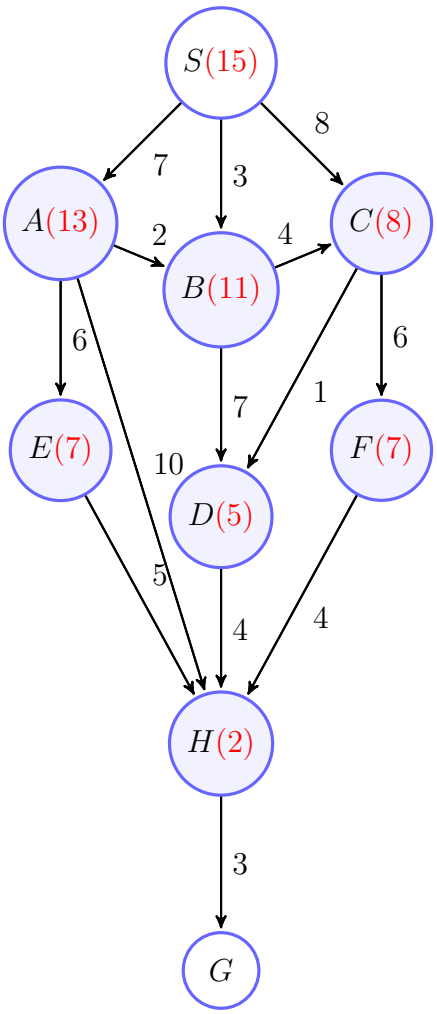
Introduction to AI Assignment 2

Vdovenko Anton

April 12, 2021

Question 1

For the below graph (heuristic values are provided in red color and actual costs are in black color), please provide answers to the following answers. Also note that S is start state and G is goal state.



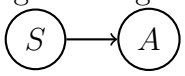
(a) Is the heuristic admissible? Provide justification.

A **heuristic** function is said to be **admissible** if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

Node n	**Minimum cost to reach goal from n**	**h(n)**
S	15 {S,B,C,D,H,G}	15
A	13 {A,H,G}	13
B	12 {B,C,D,H,G}	11
C	8 {C,D,H,G}	8
E	8 {E,H,G}	7
D	7 {D,H,G}	5
F	7 {F,H,G}	7
H	3 {H,G}	2

Answer: The heuristic is **admissible**.

(b) Is the heuristic consistent? Provide justification.

The heuristic is consistent, or monotone, if its estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour. Lets look on Trail 

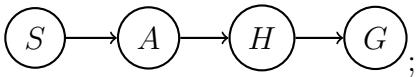
we have h(S) = 15, while sum(h(A)=13 , cost (S,A)=7) = 20. Hence we can conclude that heuristic is not consistent.

Answer: The heuristic is **not consistent**.

(c) Provide the search steps (as discussed in class) with DFS, BFS (with and without priority queue), Best First Search and A* search. Specify for each algorithm if the open list is queue, stack, or priority queue. Feel free to add rows/columns, or other details in the table below. Open list contains nodes that are to be explored, and "Nodes to add" are the successors of the node that is recently popped.

DFS using stack:

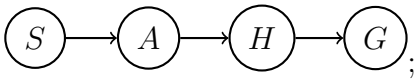
Step #	Open List	POP	Nodes to add
1		S	A,B,C
2	A^S, B^S, C^S	A^S	E,H
3	E^A, H^A, B^S, C^S	E^A	
4	H^A, B^S, C^S	H^A	G

Path: 

; Cost: **20**

BFS without priority queue:

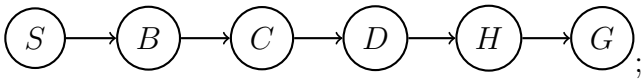
Step #	Open List	POP	Nodes to add
1		S	A,B,C
2	A^S, B^S, C^S	A^S	E,H
3	B^S, C^S, E^A, H^A	B^S	D
4	C^S, E^A, H^A, D^B	C^S	F
5	E^A, H^A, D^B, F^C	E^A	
6	H^A, D^B, F^C	H^A	G

Path: 

; Cost: **20**

BFS with priority queue:

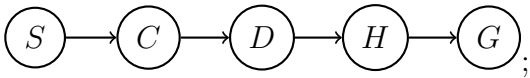
Step #	Open List	Cost	POP	Nodes to add
1			S	A,B,C
2	B^S, A^S, C^S	3,7,8	B^S	D,C
3	A^S, C^B, D^B	7,7,10	A^S	E,H
4	C^B, D^B, E^A, H^A	7,10,13,17	C^B	D,F
5	D^C, E^A, F^C, H^A	8,13,13,17	D^C	H
6	H^D, E^A, F^C, H^A	12,13,13,17	H^D	G

Path: 

; Cost: **15**

Best First Search with priority queue:

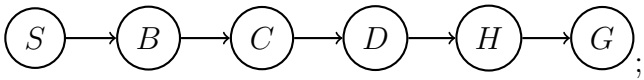
Step #	Open List	Heuristic Cost	POP	Nodes to add
1			S	A,B,C
2	C^S, B^S, A^S	8,11,13	C^S	D,F
3	D^C, F^C, B^S, A^S	5,7,11,13	D^C	H
4	H^D, F^C, B^S, A^S	2,7,11,13	H^D	G

Path: 

; Cost: **16**

A* with priority queue:

Step #	Open List	f(n) = g(n) + h(n)	POP	Nodes to add
1			S	A,B,C
2	B^S, C^S, A^S	14,16,20	B^S	D,C
3	D^B, C^B, A^S	15,15,20	D^B	H
4	C^B, H^D, A^S	15,17,20	C^B	D,F
5	D^C, H^B, A^S, F^C	13,17,20,20	D^C	H
6	H^D, A^S, F^C	14,20,20	H^D	G

Path: 

; Cost: **15**

Question 2

```
import random
import math
from environment import Agent, Environment
from simulator import Simulator
import sys
from searchUtils import searchUtils
class SearchAgent(Agent):
    """ An agent that drives in the Smartcab world.
        This is the object you will be modifying. """
    def __init__(self, env,location=None):
        super(SearchAgent, self).__init__(env)      # Set the agent in the evironment
        self.valid_actions = self.env.valid_actions  # The set of valid actions
        self.action_sequence=[]
        self.searchutil = searchUtils(env)
    def choose_action(self):
        """ The choose_action function is called when the agent is asked to choose
            which action to take next"""

        # Set the agent state and default action
        action=None
        if len(self.action_sequence) >=1:
            action = self.action_sequence[0]
        if len(self.action_sequence) >=2:
            self.action_sequence=self.action_sequence[1:]
        else:
            self.action_sequence=[]
        return action

    def drive(self,goalstates,inputs):
        # Implementation of pseudo code
        def A_Star(start, goal):
            open_set = [start]
            close_set = []
            previous_state = {}
            g_score = {start['location']: 0}
            f_score = {start['location']: heuristic_cost_estimate(start['location'], goal['
                                                                  location'])}
            current = []

            while open_set is not None:

                if len(open_set) == 1:
                    current = open_set[0]
                else:
                    if len(f_score) >= 1:
                        min_f_score = min(f_score, key=f_score.get)
                        current = [d for d in open_set if d['location'] == min_f_score][0]

                    else:
                        return (False, [None])

                if current['location'] == goal['location']:
                    return (True, reconstruct_path(previous_state, current))

                open_set.remove(current)

                try:
                    del f_score[current['location']]
                except:
                    pass
                close_set.append(current)

                for action in ['forward-3x', 'forward-2x', 'forward', 'left', 'right', None]:
                    neighbor = self.env.applyAction(self, current, action)
                    if neighbor['location'] == current['location']:
                        continue
                    tentative_g_score = g_score[current['location']] + 1

                    if neighbor not in open_set:
                        open_set.append(neighbor)
                    elif tentative_g_score >= g_score[neighbor["location"]]:
                        continue

                    previous_state[neighbor['location']] = (current["location"], action)
```

```
                    g_score[neighbor['location']] = tentative_g_score
                    f_score[neighbor['location']] = g_score[neighbor['location']] +
                                                            heuristic_cost_estimate(
                        neighbor['location'], goal['location'])

            if current == goal:
                return (True, reconstruct_path(previous_state, current))
            else:
                return (False, reconstruct_path(previous_state, current))

        def reconstruct_path(previous_state, current):
            total_path = []
            action_step = []

            while 'previous' in current:
                total_path.append(current)
                current = current['previous']

            total_path.append(current)
            total_path = [*reversed(total_path)]

            for i in range(len(total_path) - 1):
                action_step.append(self.env.getAction(total_path[i], total_path[i + 1]))

            return action_step

        def heuristic_cost_estimate(start, goal):
            return (abs(goal[0] - start[0]) + abs(goal[1] - start[1])) / 2

        if_reach, act_sequence = A_Star(self.state, goalstates[0])

        x_pos = self.state['location'][1]
        if if_reach == True:
            return act_sequence
        elif x_pos > 16:
            return [None]
        else:
            return act_sequence[:1]

    def update(self):
        """ The update function is called when a time step is completed in the
            environment for a given trial. This function will build the agent
            state, choose an action, receive a reward, and learn if enabled. """
        startstate = self.state
        goalstates =self.env.getGoalStates()
        inputs = self.env.sense(self)
        self.action_sequence = self.drive(goalstates,inputs)
        action = self.choose_action()  # Choose an action
        self.state = self.env.act(self,action)
        return


def run(filename):
    """ Driving function for running the simulation.
        Press ESC to close the simulation, or [SPACE] to pause the simulation. """

    env = Environment(config_file=filename,fixmovement=False)

    agent = env.create_agent(SearchAgent)
    env.set_primary_agent(agent)

    ###############
    # Create the simulation
    # Flags:
    #   update_delay - continuous time (in seconds) between actions, default is 2.0 seconds
    #   display      - set to False to disable the GUI if PyGame is enabled
    sim = Simulator(env, update_delay=2)

    ###############
    # Run the simulator
    ###############
    sim.run()


if __name__ == '__main__':
    run(sys.argv[1])
```

Question 3

Taxi drivers in Singapore can pick up customers from any location that is not on highways. However, they require guidance on where to pick up customers when they do not have customers on board and there are no bookings. In this question, we address this guidance problem.There are four locations: L1, L2, L3 and L4 from where taxi drivers can pick up and drop off customers. At any decision epoch, the chances of the taxi driver picking up a customer from different locations are provided in Table 1 below:

Location	**Chance of finding customer**
L1	0.3
L2	0.8
L3	0.1
L4	0.6

Once the taxi driver picks up a customer, the customer determines the destination. Observed probabilities (from past data) of a customer starting from a source location and going to a destination location are given below:

Source → Destination	**Probability**
L1 → L2	0.4
L1 → L3	0.35
L1 → L4	0.25
L2 → L1	0.4
L2 → L3	0.6
L3 → L1	0.6
L3 → L4	0.4
L4 → L1	0.65
L4 → L2	0.35

Travel fare between different locations are as follows:

Source → Destination	**Fare**
L1 → L2	8$
L1 → L3	13$
L1 → L4	15$
L2 → L1	10$
L2 → L3	9$
L3 → L1	13$
L3 → L4	10$
L4 → L1	9$
L4 → L2	7$

Cost of travelling between locations is as follows:

Source → Destination	**Fare**
L1 → L2	1$
L1 → L3	1.5$
L1 → L4	1.25$
L2 → L1	1$
L2 → L3	0.75$
L3 → L1	1.5$
L3 → L4	0.8$
L4 → L1	1.25$
L4 → L2	1$

The taxi driver can either pickup from the current location or move to another location. Pickup corresponds to picking up a customer (if one is found) and dropping them of at their destination. Pickup action succeeds with probabilities specified in Table 1 and if the taxi picks up a customer, destination location is determined by the probabilities in Table 2. When Pickup action fails, the taxi remains in its current location. Move to another location is always successful and taxi moves to the desired location with probability 1.

Both pickup and move actions take one-time step each. Please provide the following:

a) Provide an MDP model that guides the taxi driver on "move and pickup customers".

s (current state)	a (action)	$s\prime$ (new state)	$Pr(s\prime\|s,a)$	$R(s,a,s\prime)$
L1	P & M	L2	0.3 * 0.4 = 0.12	8\$ - 1\$ = 7\$
L1	P & M	L3	0.3 * 0.35 = 0.105	13\$ - 1.5\$ = 11.5\$
L1	P & M	L4	0.3 * 0.25 = 0.075	15\$ - 1.25\$ = 13.75\$
L2	P & M	L1	0.8 * 0.4 = 0.32	10\$ - 1\$ = 9\$
L2	P & M	L3	0.8 * 0.6 = 0.48	9\$ - 0.75\$ = 8.25\$
L3	P & M	L1	0.1 * 0.6 = 0.06	13\$ - 1.5\$ = 11.5\$
L3	P & M	L4	0.1 * 0.4 = 0.04	10\$ - 0.8\$ = 9.2\$
L4	P & M	L1	0.6 * 0.65 = 0.39	9\$ - 1.25\$ = 7.75\$
L4	P & M	L2	0.6 * 0.35 = 0.21	7\$ - 1\$ = 6\$

b) After providing the MDP, show three iterations of value iteration. Initialize $\forall s, V^0(s) = 0$ calculate:

1) $\forall s, V^1(s), V^2(s), V^3(s)$
2) $\forall s, \pi^1(s), \pi^2(s), \pi^3(s)$

s	a	$V^0(s)$	$Q^1(s,a)$	$V^1(s)$	$\pi^1(s)$
L1	P & M → L2		0.12 * (7\$ + 0) = 0.84\$		
L1	P & M → L3	0	0.105 * (11.5\$ + 0) = 1.2075\$	1.2075\$	P & M → L3
L1	P & M → L4		0.075 * (13.75\$ + 0) = 1.03125\$		
L2	P & M → L1	0	0.32 * (9\$ + 0) = 2.88\$	3.96\$	P & M → L3
L2	P & M → L3		0.48 * (8.25\$ + 0) = 3.96\$		
L3	P & M → L1	0	0.06 * (11.5\$ +0) = 0.69\$	0.69\$	P & M → L1
L3	P & M → L4		0.04 * (9.2\$ + 0) = 0.368\$		
L4	P & M → L1	0	0.39 * (7.75\$ + 0) = 3.0225\$	3.0225\$	P & M → L1
L4	P & M → L2		0.21 * (6\$ + 0) = 1.26\$		

$Q^2(s,a)$	$V^2(s)$	$\pi^2(s)$	$Q^3(s,a)$	$V^3(s)$	$\pi^3(s)$
0.12 * (7\$ + 1.2075\$) = 0.9894\$			0.12 * (7\$ + 1.334\$) = 1\$		
0.105 * (11.5\$ + 1.2075\$) = 1.334\$	1.334\$	P & M → L3	0.105 * (11.5\$ + 1.334\$) = 1.35\$	1.35\$	P & M → L3
0.075 * (13.75\$ + 1.2075\$) = 1.1218\$			0.075 * (13.75\$ + 1.334\$) = 1.13\$		
0.32 * (9\$ + 3.96\$) = 4.18\$	5.86\$	P & M → L3	0.32 * (9\$ + 5.86\$) = 4.76\$	6.77\$	P & M → L3
0.48 * (8.25\$ + 3.96\$) = 5.86\$			0.48 * (8.25\$ + 5.86\$) = 6.77\$		
0.06 * (11.5\$ + 0.69\$) = 0.73\$	0.73\$	P & M → L1	0.06 * (11.5\$ +0.73\$) = 0.734\$	0.734\$	P & M → L1
0.04 * (9.2\$ + 0.69\$) = 0.4\$			0.04 * (9.2\$ + 0.73\$) = 0.4\$		
0.39 * (7.75\$ + 3.0225\$) = 4.2\$	4.2\$	P & M → L1	0.39 * (7.75\$ + 4.2\$) = 4.66\$	4.66\$	P & M → L1
0.21 * (6\$ + 3.0225\$) = 1.9\$			0.21 * (6\$ + 4.2\$) = 2.142\$		

Question 4

In this question, you need to install OpenAI gym. Once the gym is installed, you have to implement Q-learning for the FrozenLake environment in python using the gym library and show the rewards obtained.

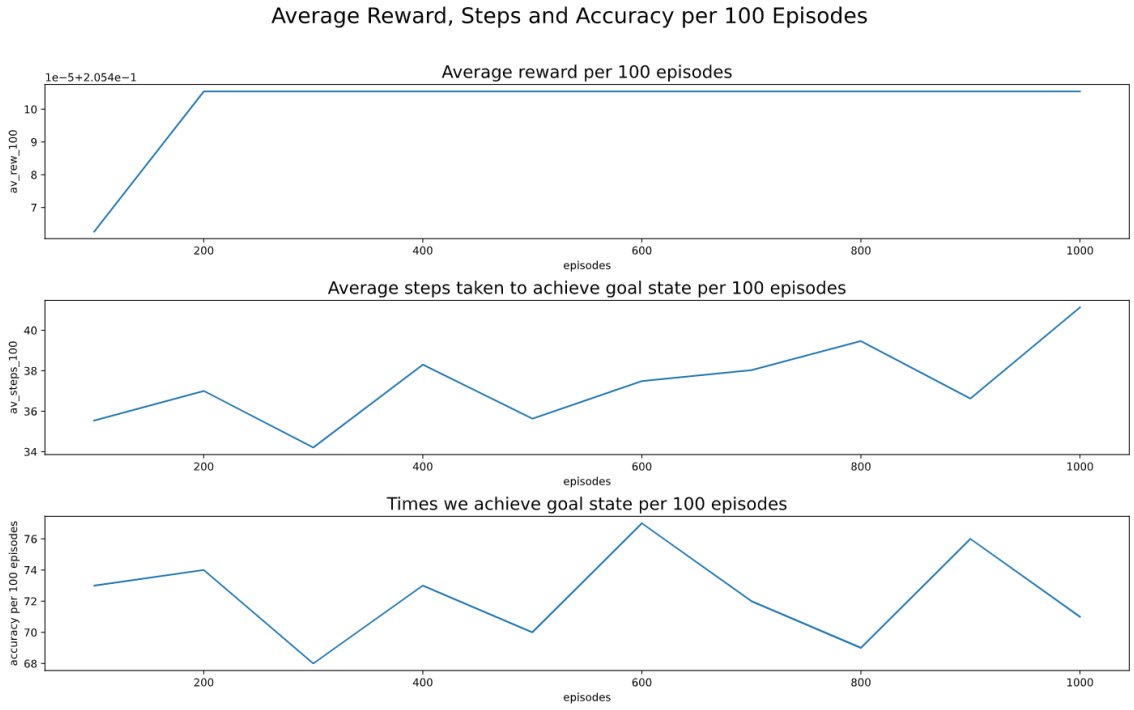
a) Code that implements Q-learning for Frozenlake example in gym.

```
import numpy as np
import gym
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

env = gym.make("FrozenLake-v0")

def stats(env, policy, episodes=100):
  misses = 0
  steps_list = []
  for episode in range(episodes):
    observation = env.reset()
    steps=0
    while True:

      action = policy[observation]
      observation, reward, done, _ = env.step(action)
      steps+=1
      if done and reward == 1:

        steps_list.append(steps)
        break
      elif done and reward == 0:

        misses += 1
        break
  y = (100 - ((misses/episodes) * 100))
  steps_num = np.mean(steps_list)
  return y,steps_num


def policy(env,stateValue, discount=0.95):
  policy = [0 for i in range(env.nS)]
  for state in range(env.nS):
    action_values = []
    for action in range(env.nA):
      action_value = 0
      for i in range(len(env.P[state][action])):
        prob, next_state, r, _ = env.P[state][action][i]
        action_value += prob * (r + discount * stateValue[next_state])
      action_values.append(action_value)
    best_action = np.argmax(np.asarray(action_values))
    policy[state] = best_action
  return policy

def question_4(env, max_iterations=100000, discount=0.95):
  plot_dic = {"accuracy per 100 episodes":[],
  "episodes":[],
  "av_rew_100":[],
  "av_steps_100":[]}
  stateValue = [0 for i in range(env.nS)]
  newStateValue = stateValue.copy()
  for i in range(max_iterations):
    for state in range(env.nS):
      action_values = []
      for action in range(env.nA):
        state_value = 0
        for index in range(len(env.P[state][action])):
          prob, next_state, reward, done = env.P[state][action][index]
          state_action_value = prob * (reward + discount*stateValue[next_state])
          state_value += state_action_value
        action_values.append(state_value)
        best_action = np.argmax(np.asarray(action_values))
        newStateValue[state] = action_values[best_action]
```

```
    if i in [ep for ep in range(100,(max_iterations + 1),100)]: # extract policy and score for
                                                               each 100 episodes

        policy_ = policy(env,newStateValue)
        plot_dic["accuracy per 100 episodes"].append(stats(env,policy_)[0])
        plot_dic["episodes"].append(i)
        plot_dic["av_rew_100"].append(np.mean(newStateValue))
        plot_dic["av_steps_100"].append(stats(env,policy_)[1])

    elif i > 1001:
      if sum(stateValue) - sum(newStateValue) < 1e-04:    # if there is negligible difference
                                                           break the loop
        print("Episodes:",i)
        break

    else:
      stateValue = newStateValue.copy()

df = pd.DataFrame(plot_dic)
fig, axs = plt.subplots(3,figsize=(18, 10),sharex = False)
plt.subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=0.4)
fig.suptitle('Average Reward, Steps and Accuracy per 100 Episodes',fontsize=20)
sns.lineplot(data = df, x="episodes",y="accuracy per 100 episodes", ax=axs[2])
axs[2].set_title("Times we achieve goal state per 100 episodes",fontsize=16)
sns.lineplot(data = df, x="episodes",y="av_rew_100",ax = axs[0])

axs[0].set_title("Average reward per 100 episodes",fontsize=16)
sns.lineplot(data = df, x="episodes",y="av_steps_100",ax = axs[1])
axs[1].set_title("Average steps taken to achieve goal state per 100 episodes",fontsize=16)
plt.plot()
print("Final_policy:",policy(env,stateValue))
```

Episodes: 1002
Final policy: [0, 3, 0, 3, 0, 0, 0, 0, 3, 1, 0, 0, 0, 2, 1, 0]

b) For each episode, compute the total accumulated reward (also called episode return). Plot the average return (over the last 100 episodes) while your agent is learning (x-axis will be the episode number, y-axis will be the average return over the last 100 episodes). Make sure that you train for sufficiently many episodes so that convergence occurs.



Question 5

In this question, you will employ Singular Value Decomposition to obtain word embeddings and compare the generated word embeddings with the word embeddings generated using word2vec. The corpus (or dataset) to be considered is "200Reviews.csv".

a) Parse the reviews in "200Reviews.csv", i.e., divide reviews into sentences and sentencesinto words and remove the stop words. You can employ the "NLP-pipeline-example.ipynb" example we talked about in class.

```
import numpy as np
import pandas as pd
import nltk
from nltk.stem import WordNetLemmatizer
import sys
from nltk.corpus import wordnet
from nltk.corpus import stopwords
from nltk.parse.malt import MaltParser
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('stopwords')

# Read data and put all reviewes in one string called paragraph
df = pd.read_csv("Q5/200Reviews.csv")
paragraph = ""
for i in df["review"]:
    paragraph += i

#Step1: Sentence segmentation

sentences = nltk.sent_tokenize(paragraph)

print("\n\n"+repr(len(sentences))+" sentences found in the paragraph. They are as follows:")
for k in range(len(sentences)):
    print("("+repr(k+1)+") "+sentences[k])

#Step 2: Word tokenization
for k in range(len(sentences)):
    # word tokenizer will keep the punctuations. To get rid of punctuations, use nltk.
                                                    RegexpTokenizer(r'\w+').tokenize(sentences[
                                                    k])
    # words = nltk.word_tokenize(sentences[k])
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    print("Words in sentence "+repr(k+1)+" are: ")
    wordlist=[]
    for w in words:
        wordlist.append(w)
    print(wordlist)

#Step 3: Predicting parts off speech for each token
# You can use nltk.help.upenn_tagset() to get the description of each of pos tag.
#Uncomment following line to print the list of all tags
#nltk.download('tagsets')
#nltk.help.upenn_tagset()
for k in range(len(sentences)):
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    tagged_words = nltk.pos_tag(words)
    print("Tagged Words in sentence "+repr(k+1)+" are: ")
    print(tagged_words)

#Step 4: Text Lemmatization
#As we are using wordnet Lemmatizer and the the standard NLTK pos tags are treebank tags, we
                                                    need to convert the treebank tag
#to wordnet tags.
def get_wordnet_pos(treebank_tag):

    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
```

```
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return ''

wordnet_lemmatizer = WordNetLemmatizer()
for k in range(len(sentences)):
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    tagged_words = nltk.pos_tag(words)
    lemmatized_wordlist=[]
    print("Word:Lemmatized Word in sentence "+repr(k+1)+" are: ")
    for w in tagged_words:
        wordnettag=get_wordnet_pos(w[1])
        if wordnettag == '':
            lemmatizedword = wordnet_lemmatizer.lemmatize(w[0].lower())
        else:
            lemmatizedword = wordnet_lemmatizer.lemmatize(w[0].lower(),pos=wordnettag)
        if w[0].istitle():
            lemmatizedword = lemmatizedword.capitalize()
        elif w[0].upper()==w[0]:
            lemmatizedword = lemmatizedword.upper()
        else:
            lemmatizedword = lemmatizedword
        lemmatized_wordlist.append((w[0],lemmatizedword))

    print(lemmatized_wordlist)

#Step 5: Identifying stop words
stopWords = set(stopwords.words('english'))
for k in range(len(sentences)):
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    wordlist_wo_stopwords=[]
    print("Words in sentence "+repr(k+1)+" without stop words are: ")
    for w in words:
        if w not in stopWords:
            wordlist_wo_stopwords.append(w)
    print(wordlist_wo_stopwords)
```

b) Create the co-occurrence matrix for all the remaining words (after stop words are eliminated), where the window of co-occurrence is 5 on either side of the word.

```
#Step 6: Create general wordlist for co occurence matrix
general_wordlist = []
stopWords = set(stopwords.words('english'))
for k in range(len(sentences)):
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    wordlist_wo_stopwords=[]

    for w in words:
        if w not in stopWords:
            wordlist_wo_stopwords.append(w)
    general_wordlist.append(wordlist_wo_stopwords)

# Step 7 Create co occurence matrix
from collections import defaultdict
def co_occurrence(sentences, window_size):
    d = defaultdict(int)
    vocab = set()
    for text in sentences:

        # iterate over sentences
        for i in range(len(text)):
            token = text[i]
            vocab.add(token)  # add to vocab
            next_token = text[i+1 : i+1+window_size]
            for t in next_token:
                key = tuple( sorted([t, token]) )
                d[key] += 1

    # formulate the dictionary into dataframe
    vocab = sorted(vocab) # sort vocab
    df = pd.DataFrame(data=np.zeros((len(vocab), len(vocab)), dtype=np.int16),
```

```
                        index=vocab,
                        columns=vocab)
    for key, value in d.items():
        df.at[key[0], key[1]] = value
        df.at[key[1], key[0]] = value
    return df

co_occurrence(general_wordlist,5)
```

```
co_occurrence(general_wordlist,5)
```

	0	000	1	10	100	101	11	117	12	13th	...	z	zapped	zero	zip	zombie	zombies	zone	zoom	zooms	Êxtase
0	0	0	0	2	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
000	0	0	0	1	0	0	1	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	0	0	1	0	0	0	...	0	0	0	0	3	0	0	0	0	0
10	2	1	2	3	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
zombies	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	1	0	0	0	0	0
zone	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
zoom	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
zooms	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
Êxtase	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

7880 rows x 7880 columns

c) Apply SVD and obtain word embeddings of size 100.

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=100)
svd_reduced  = svd.fit_transform(cooccurence)
svd_explained = svd.explained_variance_ratio_.sum()
print('Explained Variance =', round(svd_explained, 3))

#rebuild the cooccurance matrix
cooccurance_reduced = pd.DataFrame(svd_reduced, index = list(cooccurence.index), columns=[[*
                                                    range(1,101)]])
cooccurance_reduced.sample(5)

# Find out which two words are similar, based on cosine similarity
from itertools import combinations
from sklearn.metrics.pairwise import cosine_similarity
words = ['drama', 'zombie', 'enemy', 'suspense']

sim_result = []

for i in [*combinations(words,2)]:

    word_1 = [cooccurance_reduced.loc[i[0]].values]

    word_2 = cooccurance_reduced.loc[i[1]].values.reshape(1, -1)
    cos_sim = cosine_similarity(word_1, word_2)

    sim_result.append([i[0], i[1], cos_sim[0][0]])

df_svd = pd.DataFrame(sim_result, columns=['word1', 'word2', 'cosine_similarity']).sort_values
                                                    (by='cosine_similarity', ascending=False)
print(df_svd)

# finding the top 5 similar words
word = 'drama'

total_score= pd.DataFrame(cooccurence.index, columns = ['word'])
total_score['cosine_sim'] = 0

embedding_1 = cooccurance_reduced.loc[word].values.reshape(1, -1)

for i in cooccurence.index:
```

```
    embedding_2 = cooccurance_reduced.loc[i].values.reshape(1, -1)
    cos_lib = cosine_similarity(embedding_1, embedding_2)
    total_score.loc[total_score['word']==i, 'cosine_sim'] = cos_lib[0][0]

print('top 5 similar words to', word)
print(total_score.sort_values(by=['cosine_sim'], ascending = False)[1:6])
```