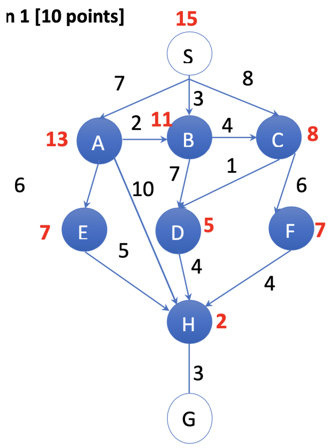
Question 1: Heuristic Search



(a) Is the heuristic admissible? Provide justification.

The Heuristic is admissible. h(n) is always lower than min cost.

Node n	Min Cost	h(n)
S	15	15
A	13	13
B	12	11
C	8	8
E	8	7
D	7	5
F	7	7
H	3	2

(b) Is the heuristic consistent? Provide justification.

The heuristic is not consistent. A consistent heuristic, $h(n) <= c(n,p) + h(p)$, where n is the parent node and p is a child of node n.

$h(f) = 7$; $c(f,h) = 4$; $h(h) = 2$

$h(f) >= c(f,h) + h(h)$, where node f is a parent of node h. Therefore 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.

<u>Algorithm: BFS (Queue); Path: SAHG; Cost: 20</u>

Notation: The Open List contains the following: Node stored, Parent node. They will be represented as follows Np (N = Node in uppercase letters, p = Parent node in lowercase letters)

Step	Open list	POP	Nodes to add
1	S	S	A,B,C
2	As,Bs,Cs	As	E,H
3	Bs,Cs,Ea,Ha	Bs	D
4	Cs,Ea,Ha,Db	Cs	F
5	Ea,Ha,Db,Fc	Ea	
6	Ha,Db,Fc	Ha	G
7	Db,Fc,Gh	Db	
8	Fc,Gh	Fc	
9	Gh	Gh	

<u>Algorithm: DFS (Stack); Path: SAHG; Cost: 20</u>

Notation: The Open List contains the following: Node stored, Parent node. They will be represented as follows Np (N = Node in uppercase letters, p = Parent node in lowercase letters)

Step	Open list	POP	Nodes to add
1	S	S	A,B,C
2	As,Bs,Cs	As	E,H
3	Ea,Ha,Bs,Cs	Ea	
4	Ha,Bs,Cs	Ha	G
5	Gh,Bs,Cs	Gh	

<u>Algorithm: BFS (Priority Queue); Path: SBCDHG; Cost: 15</u>

Notation: The Open List contains the following: Node stored, Parent node, and Cost from source to node. They will be represented as follows Np20 (N = Node in uppercase letters, p = Parent node in lowercase letters, 20 = cost from source to N)

Step	Open list	POP	Nodes to add
1	S	S	A,B,C
2	Bs3,As7,Cs8	Bs3	C,D
3	As7,Cb7,Db10	As7	E,H
4	Cb7,Db10,Ea13,Ha17	Cb7	D,F
5	Dc8,Ea13,Fc13,Ha17	Dc8	H
6	Hc12,Ea13,Fc13	Hc12	G
7	Ea13,Fc13,Gh15	Ea13	H
8	Fc13,Gh15,He18	Fc13	H
9	Gh15,Hf17	Gh15	

Algorithm: Best First (Priority Queue); Path: SCDHG; Cost: 16

Notation: The Open List contains the following: Node stored, Parent node, and heuristic value. They will be represented as follows Np20 (N = Node in uppercase letters, p = Parent node in lowercase letters, 20 = heuristic value of N)

Step	Open list	POP	Nodes to add
1	S	S	A,B,C
2	Cs8,Bs11,As13	Cs8	D,F
3	Dc5,Fc7,Bs11,As13	Dc5	H
4	Hd2,Fc7,Bs11,As13	Hd2	G
5	Gh0,Fc7,Bs11,As13	Gh0	

Algorithm: A* (Priority Queue); Path: SBCDHG; Cost: 15

Notation: The Open List contains the following: Node stored, Parent node, and the sum of heuristic and incurred value - f(n) = g(n) + h(n). They will be represented as follows Np20 (N = Node in uppercase letters, p = Parent node in lowercase, 20 = f(n))

Step	Open list	POP	Nodes to add
1	S	S	A,B,C
2	Bs14,Cs16,As20	Bs14	D,C
3	Cb15,Db15,As20	Cb15	D,F
4	Dc13,As20,Fc21	Dc13	H
5	Hd14,As20,Fc21	Hd14	G
6	Gh15,As20,Fc21	Gh15	

Question 2: Self Driving Car (A* Search)

Node: The node represents each location in the grid. For each of these locations, we record the cost to get to that location (self.g), the heuristic cost between that location and the goal location using the manhattan distance (self.h), the sum of the 2 (self.f), the current location of the node (self.loc), and lastly the parent node (self.parent)

```
class Node():
    def __init__(self, parent=None, loc=None):
        self.parent = parent
        self.loc = loc
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.loc[0] == other.loc[0] and self.loc[1] == other.loc[1]
    def __lt__(self, other):
        return self.f < other.f
    def __gt__(self, other):
        return self.f > other.f
    def __str__(self):
        return str(self.loc) + str(self.f)
    def __repr__(self):
        return str(self.loc) + str(self.f)
```

Drive Function: this function is called on every iteration.

1. It first gets the start and end locations from the environment variable.
2. Then runs the Astar algorithm to get the optimal path between current location and the goal
3. Lastly, it converts the path given by the A star algorithm to an action sequence

```
def drive(self,goalstates,inputs):
    """Write your algorithm for self driving car"""
    # get start and end
    start = self.state['location']
    end = goalstates[0]['location']

    # run astar algorithm, get path
    goalReached, path = self.a_star(inputs, start, end)

    # return action sequence from path
    act_sequence=[]
    prev = None
    for step in path:
        if prev is None:
            prev = step
            continue
        x = prev[0] - step[0]
        y = prev[1] - step[1]
        if x == 1 and y == 1:
            act_sequence.append('right')
        elif x == -1 and y == 1:
            act_sequence.append('left')
        elif x == 0 and y == 1:
            act_sequence.append('forward')
        elif x == 0 and y == 2:
            act_sequence.append('forward-2x')
        elif x == 0 and y == 3:
            act_sequence.append('forward-3x')
    return act_sequence
```

A* Algorithm: Find optimum path, given maze, start, and end

```python
def a_star(self, maze, start, end):
    maze1dLength = len(maze)
    maze2dLength = len(maze[0])

    startNode = Node(None, start)
    endNode = Node(None, end)
    openList = []
    closedList = []

    openList.append(startNode)
    while len(openList) > 0:
        # get current Node
        currentNode = heapq.heappop(openList)
        closedList.append(currentNode)

        # create path
        if currentNode == endNode:
            return endNode, self.createPath(currentNode)

        # Generate children
        children = []
        for action in [(1, 1), (-1, 1), (0, 1), (0, 2), (0, 3)]: # actions
            loc = (currentNode.loc[0] + action[0], currentNode.loc[1] + action[1])
            if not (loc[0] < maze1dLength and loc[1] < maze2dLength and loc[0] >= 0 and loc[1] >= 0): # not in maze
                continue
            if maze[loc[0]][loc[1]] == 1: # have car
                continue
            if action[1] > 1 and maze[loc[0]][loc[1] - 1] == 1: # if moves 2 steps or more, check car between
                continue
            if action[1] > 2 and maze[loc[0]][loc[1] - 2] == 1: # if moves 3 steps or more, check car between
                continue
            newNode = Node(currentNode, loc)
            children.append(newNode)

        # add child to set
        for child in children:
            # not in closed List
            if self.isInList(child,closedList) is not None:
                continue

            # derive cost
            child.g = currentNode.g + 1
            child.h = self.hCost(child,endNode)
            child.f = child.g + child.h

            # not in openList and not greaterCost than in openList
            childInOpenList = self.isInList(child,openList)
            if childInOpenList is not None and child.f > childInOpenList.f:
                continue

            # add to openList
            heapq.heappush(openList, child)
    return None, [None]
```

Helper Functions: called by the A* Algorithm

1. hCost: the heuristic cost of a location to get to the goal location given by the manhattan distance.
2. createPath: gets the path from start to end using the pointer to its parent in each node.

3. isInList: a simple helper function to check is item x is in a list, either openList or closeList can use this function.

```
def hCost(self,child,endNode):
    return (abs(child.loc[0] - endNode.loc[0]) + abs(child.loc[1] - endNode.loc[1]))

def createPath(self,currentNode):
    path = []
    current = currentNode
    while current is not None:
        path.append(current.loc)
        current = current.parent
    return path

def isInList(self,x,xList):
    for item in xList:
        if x == item:
            return item
    return None
```

Question 3: Taxi Driver MDP

(a) You need to provide an MDP model that guides the taxi driver on "move and pickup customers". MDP is the tuple <S, A, P, R>

s (current state)	a (action)	s' (new state)	Pr(s' | s,a)	R(s,a,s')
L1	L1	L1	0.7	0
		L2	0.12	7
		L3	0.105	11.5
		L4	0.075	13.75
	L2	L2	1	-1
	L3	L3	1	-1.5
	L4	L4	1	-1.25
L2	L2	L2	0.2	0
		L1	0.32	9
		L3	0.48	8.25
	L1	L1	1	-1
	L3	L3	1	-0.75
	L4	L4	1	-inf
L3	L3	L3	0.9	0
		L1	0.06	11.5
		L4	0.04	9.2
	L1	L1	1	-1.5
	L2	L2	1	-inf
	L4	L4	1	-0.8
L4	L4	L4	0.4	0
		L1	0.39	7.75
		L2	0.21	6
	L1	L1	1	-1.25
	L2	L2	1	-1
	L3	L3	1	-inf

(b) After providing the MDP, show three iterations of value iteration. Initialize $\forall s,\ V_0(s) = 0$ *and* calculate

(i) $\forall s,\ V_1(s),\ V_2(s)$ *and* $V_3(s)$

(ii) $\forall s,\ \pi_1(s),\ \pi_2(s)$ *and* $\pi_3(s)$

$Q^{t+1}(s,a) = \sum Pr(s' \mid s,a)\ [\ R(s,a,s') + \gamma V^t(s')\ ]$, given $\gamma = 0.99$

Iteration 1

s	a	s'	$V^0(s)$	Pr(s'|s,a) [R(s,a,s')+γV^t(s')]	$Q^1(s,a)$	$V^1(s)$	$\pi^1(s)$
L1	L1	L1	0	0	3.07875	3.07875	L1
	L1	L2		0.84			
	L1	L3		1.2075			
	L1	L4		1.03125			
	L2	L2		-1	-1		
	L3	L3		-1.5	-1.5		
	L4	L4		-1.25	-1.25		
L2	L1	L1	0	-1	-1	6.84	L2
	L2	L2		0	6.84		
	L2	L1		2.88			
	L2	L3		3.96			
	L3	L3		-0.75	-0.75		
	L4	L4		**-inf**	**-inf**		
L3	L1	L1	0	-1.5	-1.5	1.058	L3
	L2	L2		**-inf**	**-inf**		
	L3	L3		0	1.058		
	L3	L1		0.69			
	L3	L4		0.368			
	L4	L4		-0.8	-0.8		
L4	L1	L1	0	-1.25	-1.25	4.2825	L4
	L2	L2		-1	-1		
	L3	L3		**-inf**	**-inf**		
	L4	L4		0	4.2825		
	L4	L1		3.0225			
	L4	L2		1.26			

Iteration 2

s	a	s'	$V^1(s)$	Pr(s'|s,a) [R(s,a,s')+γV^t(s')]	$Q^2(s,a)$	$V^2(s)$	$\pi^2(s)$
L1	L1	L1	3.07875	2.13357375	6.1267125	6.1267125	L1
	L1	L2		1.2057555			
	L1	L3		1.527536063			
	L1	L4		1.259847188			
	L2	L2		5.7716	5.7716		
	L3	L3		-0.45258	-0.45258		
	L4	L4		2.989675	2.989675		
L2	L1	L1	6.84	2.0479625	2.0479625	13.6116	L2
	L2	L2		1.35432	13.6116		
	L2	L1		5.046912			
	L2	L3		7.210368			
	L3	L3		0.29742	0.29742		
	L4	L4		**-inf**	**-inf**		
L3	L1	L1	1.058	1.5479625	1.5479625	3.439675	L4
	L2	L2		**-inf**	**-inf**		
	L3	L3		0.942678	2.10542		
	L3	L1		0.7528452			
	L3	L4		0.4098968			
	L4	L4		3.439675	3.439675		
L4	L1	L1	4.2825	1.7979625	1.7979625	8.522175	L4
	L2	L2		5.7716	5.7716		
	L3	L3		**-inf**	**-inf**		
	L4	L4		1.69587	8.522175		
	L4	L1		4.67597325			
	L4	L2		2.15033175			

Iteration 3

s	a	s'	$V^2(s)$	$Pr(s'\|s,a)$ $[R(s,a,s')+\gamma V^t(s')]$	$Q^3(s,a)$	$V^3(s)$	$\pi^3(s)$
L1	L1	L1	6.1267125	4.245811763	9.144195375	12.475484	L2
	L1	L2		1.567853445			
	L1	L3		1.844371764			
	L1	L4		1.486158403			
	L2	L2		12.475484	12.475484		
	L3	L3		1.90527825	1.90527825		
	L4	L4		7.18695325	7.18695325		
L2	L1	L1	13.6116	5.065445375	5.065445375	20.315484	L2
	L2	L2		2.6950968	20.315484		
	L2	L1		7.19215488			
	L2	L3		10.42823232			
	L3	L3		2.65527825	2.65527825		
	L4	L4		**-inf**	**-inf**		
L3	L1	L1	3.439675	4.565445375	4.565445375	7.63695325	L4
	L2	L2		**-inf**	**-inf**		
	L3	L3		3.064750425	4.46327825		
	L3	L1		0.894316695			
	L3	L4		0.50421113			
	L4	L4		7.63695325	7.63695325		
L4	L1	L1	8.522175	4.815445375	4.815445375	12.71945325	L4
	L2	L2		12.475484	12.475484		
	L3	L3		**-inf**	**-inf**		
	L4	L4		3.3747813	12.71945325		
	L4	L1		6.312911768			
	L4	L2		3.031760183			

Question 4: OpenAIGym

(1) Initializing Environment

```
[21] import gym
     import numpy as np
     import random
     from pylab import *
```

```
[22] env = gym.make('FrozenLake-v0')
```

(2) Initializing FrozenLake Class

```
class FrozenLake:
  def __init__(self):
    # attributes
    self.Q = np.zeros([env.observation_space.n,env.action_space.n])
    self.reward_total = []
    self.steps_total = []
    # parameters
    self.num_episodes = 2000
    self.max_steps = 100
    self.learning_rate = 0.8
    self.gamma = 0.95
    # self.egreedy = 0.90
    # self.egreedy_decay = 0.999
    # self.egreedy_final = 0.005
```

(3) Run (Learning)

```
def run(self):
  for i in range(self.num_episodes):
    reward_episode = 0
    steps_episode = 0
    state = env.reset()

    # for each step (action made) in one episode
    while steps_episode < self.max_steps:
      # get action (with randomness)
      actions_for_state = self.Q[state,:] + np.random.randn(1, env.action_space.n) / (i + 1) #
      action = np.argmax(actions_for_state)

      # # egreedy
      # if np.random.rand(1)[0] < self.egreedy:
      #   action = env.action_space.sample()
      # if self.egreedy > self.egreedy_final:
      #   self.egreedy *= self.egreedy_decay


      # make step with action
      new_state, reward, done, _ = env.step(action)

      # update Q table
      old_estimate = self.Q[state,action]
      step_size = self.learning_rate
      target = reward + self.gamma * np.max(self.Q[new_state,:])
      self.Q[state,action] = old_estimate + step_size * (target - old_estimate)

      # update variables
      state = new_state
      reward_episode += reward
      steps_episode += 1

      if done:
        break
    # update graphing arrays
    self.reward_total.append(reward_episode)
    self.steps_total.append(steps_episode)

    if i%100 == 0: # printing
      print(f'Step{i}: reward - {reward_episode} :: steps - {steps_episode}')
```

(4) Printing Graph

```
def plot(self):
  subplot(211)
  plot(self.avr_pass(self.reward_total), 'b', label="Mean Reward per 100 Episode")
  legend(loc="best")

  subplot(212)
  plot(self.avr_pass(self.steps_total), label="Total Number of Steps per Episode")
  legend(loc="best")
  show()

def low_pass(self, data, alpha=0.99):
  low_pass = [data[0]]
  for i in range(1,len(data)):
    low_pass.append(alpha*low_pass[-1] + (1.0-alpha)*data[i] )
  return low_pass

def avr_pass(self, data, alpha=100):
  avr_pass = []
  for i in range(alpha - 1,len(data), alpha):
    avr_pass.append(np.mean(data[i - alpha + 1:i]))
  return avr_pass
```

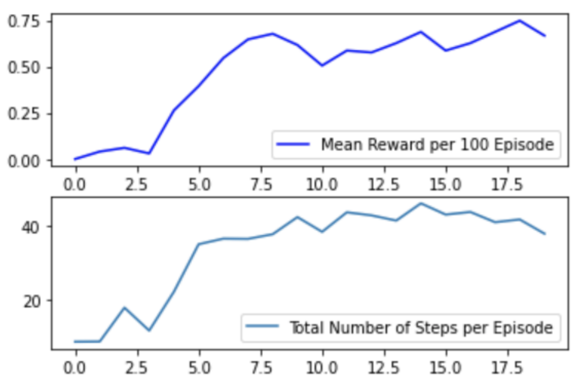
(5) Graph: Rewards

```
lake = FrozenLake()

lake.run()
lake.plot()
```

We ran the algorithm initially with the (1) epsilon greedy code, in concert with the (2) action randomization code. We got pretty similar results but the epsilon greedy code was overall a worse code, as it took longer to converge and it also resulted in lower converged rewards, due to the greater chance of using a randomized action using the epsilon greedy code. Perhaps, it might work better to use one or the other in the future.

Epsilon greedy code + Action randomization

Action randomization



Question 5: SVD Word Embeddings

(1) Load Data

Building an NLP Pipeline

[1]
```
from google.colab import drive
drive.mount('/content/drive/')
```

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

[2]
```
# load file into dataframe
csv_dir = '/content/drive/My Drive/Colab Notebooks/CS420/NLPCodes'
```

```
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')

import pandas as pd
import numpy as np

df = []
with open(f'{csv_dir}/200Reviews.csv', 'r') as file:
  df = pd.read_csv(file)
df.head()
```

(2) Data Preprocessing

[4]
```
#Step1: Sentence segmentation
df['review'] = df['review'].apply(lambda sentences: nltk.sent_tokenize(sentences))
df.iloc[0].review[0]
```

```
'"With all this stuff going down at the moment with MJ i\'ve started listening to his music, watching the odd documentary here and there, watched The Wiz and
```

[5]
```
#Step 2: Word tokenization
df['review'] = df['review'].apply(lambda sentences: [nltk.word_tokenize(sentence) for sentence in sentences])
# df.iloc[0].review[0]
```

[6]
```
#Step 3: Predicting parts off speech for each token
df['review'] = df['review'].apply(lambda sentences: [nltk.pos_tag(sentence) for sentence in sentences])
# df.iloc[0].review[0]
```

[7]
```
#Step 4: Text Lemmatization
# Get Wordnet Tag
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 ''
# Get Lemmatized word
def getLemmatizedWord(word, wordnettag):
  wordnet_lemmatizer = WordNetLemmatizer()
  lemmatizedword = ''
  if wordnettag == '':
      lemmatizedword = wordnet_lemmatizer.lemmatize(word.lower())
  else:
      lemmatizedword = wordnet_lemmatizer.lemmatize(word.lower(),pos=wordnettag)
  if word.istitle():
      lemmatizedword = lemmatizedword.capitalize()
  elif word.upper()==word:
      lemmatizedword = lemmatizedword.upper()
  else:
      lemmatizedword = lemmatizedword
  return lemmatizedword

df['review'] = df['review'].apply(lambda sentences: [[getLemmatizedWord(word[0], get_wordnet_pos(word[1])) for word in sentence] for sentence in sentences])
# df.iloc[0].review[0]
```

```
#Step 5: Identifying stop words
stopWords = set(stopwords.words('english'))
df['review'] = df['review'].apply(lambda sentences: [[word for word in sentence if word not in stopWords] for sentence in sentences])
# df.iloc[0].review[0]
```

(3) Co-occurrence matrix Class

Co-occurence Matrix

```
[9] class CoOccurenceMatrix:
      def __init__(self, window):
        self.vocabulary = []
        self.matrix = []
        self.window = window
        self.embedding = []
        self.u = None
        self.s = None
        self.vh = None

      def add(self, word, window):
        self.newWord(word) # add word
        index = self.vocabulary.index(word)
        # add all words in window
        for coword in window:
          self.newWord(coword) # add coword
          self.matrix[index][self.vocabulary.index(coword)] += 1

      def newWord(self, word):
        # add word to matrix and vocabulary if not in vocabulary
        if word not in self.vocabulary:
          self.vocabulary.append(word)
          for line in self.matrix:
            line.append(0)
          self.matrix.append([0 for i in range(len(self.vocabulary))])

      def addSentence(self, sentence):
        # for each word create window, and add word
        for count, word in enumerate(sentence):
          window = sentence[max(0, count - self.window) : min(len(self.vocabulary), count + self.window + 1)]
          if word in window:
            window.remove(word)
          self.add(word, window)

      def createEmbedding(self, size=100):
        self.u, self.s, self.vh = np.linalg.svd(matrix.matrix, full_matrices=False)
        for word in self.u:
          self.embedding.append(word[:size])

      def printMatrix(self, maxSizePrint=10):
        print(f'matrix width: {len(self.matrix[0])}')
        print(f'matrix height: {len(self.matrix)}')
        print(f'vocab size: {len(self.vocabulary)}')
        print(f'vocab list: {self.vocabulary[:maxSizePrint]}')
        for i in range(min(maxSizePrint, len(self.vocabulary))):
          print(self.matrix[i][:min(maxSizePrint, len(self.vocabulary))])

      def printEmbedding(self, maxSizePrint=10):
        print(f'embedding width: {len(self.embedding[0])}')
        print(f'embedding height: {len(self.embedding)}')
        for i in range(min(maxSizePrint, len(self.vocabulary))):
          print(self.embedding[i][:min(maxSizePrint, len(self.embedding[0]))])
```

(4) Create SVD Word Embedding

```
[10] #create co-occurence matrix
     matrix = CoOccurenceMatrix(5)
     for index, row in df.iterrows():
       for sentence in row.review:
         matrix.addSentence(sentence)
```

```
matrix.printMatrix()
```

```
[12] # get first 100 column of each matrix
     matrix.createEmbedding()
```

```
matrix.printEmbedding(4)
```

```
embedding width: 100
embedding height: 7417
[-0.00319844 -0.03302494  0.0024539   0.0214006 ]
[-0.00056103 -0.00072078 -0.00038473  0.00099664]
[-1.39089096e-04 -2.77688367e-03  3.47213244e-05  1.42663313e-03]
[-4.07631138e-03 -3.56803406e-02  9.96355636e-05  2.17062358e-02]
```

(5) Create Word2Vec Word Embedding

Word2Vec

```
[57] from gensim.test.utils import common_texts
     from gensim.models import Word2Vec
     import math

     # Get the interactive Tools for Matplotlib
     import matplotlib.pyplot as plt
     from sklearn.decomposition import PCA
```

```
[58] sentences = []
     for index, row in df.iterrows():
       for sentence in row.review:
         sentences.append(sentence)
```

```
[65] # Creating the model and setting values for the various parameters
     num_features = 100  # Word vector dimensionality
     min_word_count = 10 # Minimum word count
     num_workers = 4     # Number of parallel threads
     context = 5         # Context window size
     downsampling = 1e-3 # (0.001) Downsample setting for frequent words

     model = Word2Vec(sentences,
                      size=num_features,
                      window=context,
                      min_count=min_word_count,
                      workers=num_workers,
                      sample=downsampling)
```

(6) EDA of Data Embeddings

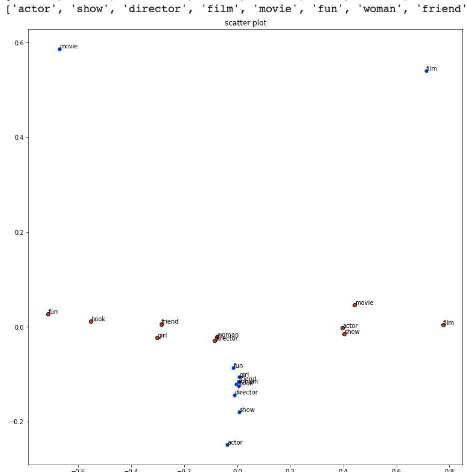
▾ Plot

```
[70] print(len(model.wv.index2word))
     print(model.wv.index2word)
     # print(sentences)

     475
     [',', '.', "''", '/', '<', 'br', '>', 'I', "'s", 'film', 'movie', 'The', '\\', ')', '(', '!', "n't", 'one', '``', 'make', '?', 'like', 'good', 'get', 'It', '
```

```
# get random words in word2vec model
w2v_words = ['actor', 'show', 'director', 'film', 'movie', 'fun', 'woman', 'friend', 'girl', 'book']
# for i in np.random.rand(5):
#   i = math.floor(i * len(model.wv.index2word))
#   w2v_words.append(model.wv.index2word[i])

svd_words = [word for word in w2v_words if word in matrix.vocabulary]

print(w2v_words)
print(svd_words)

# get word vectors
w2v_word_vectors = np.array([model.wv[w] for w in w2v_words])
svd_word_vectors = [matrix.embedding[matrix.vocabulary.index(word)] for word in svd_words]
# tranform to 2d
w2v_twodim = PCA().fit_transform(w2v_word_vectors)[:,:2]
svd_twodim = PCA().fit_transform(svd_word_vectors)[:,:2]

# plt on graph
fig=plt.figure(figsize=(10,10))
ax=fig.add_axes([0,0,1,1])
ax.set_title('scatter plot')
# w2v
ax.scatter(w2v_twodim[:,0], w2v_twodim[:,1], edgecolors='k', c='r')
for word, (x,y) in zip(w2v_words, w2v_twodim):
  ax.text(x+0.001, y+0.001, word)
# svd
ax.scatter(svd_twodim[:,0], svd_twodim[:,1], edgecolors='c', c='b')
for word, (x,y) in zip(svd_words, svd_twodim):
  ax.text(x+0.001, y+0.001, word)
plt.show()
```

(7) Plotted words

```
['actor', 'show', 'director', 'film', 'movie', 'fun', 'woman', 'friend', 'girl', 'book']
['actor', 'show', 'director', 'film', 'movie', 'fun', 'woman', 'friend', 'girl', 'book']
```



The blue data points are the SVD embedding, and the red the word2vec embedding.

The SVD embeddings are a lot more inaccurate than the word2vec embeddings. The 3 data points furthest apart for the SVD are the words ‘actor, film and movie’. These words are linguistically most related to each other in this set of words, whilst words like ‘fun, girl, book’ are much closer together despite being arbitrary in comparison with each other.

The word2vec embeddings do much better in this set of words, with ‘actor, movie’ being necessarily close, and the word ‘film’ being a little further out, but still the next closest word in the set.