CSC421: Artificial Intelligence
Assignment 2

Rafael Solorzano

June 17, 2018

1 Introduction

In this assignment Probabilist Simulation is explored by simulating two different monopoly games. In the first one, only landings on spots and chance cards occurrences are recorded, whereas the second one includes more rules to handle money and decide a winner between two player over 50 turns each. This is presented in Section 2.

The Naive Bayes learning is done with a set of positive and negative reviews that take into account only a binary Vector of words, in order to get probabilities. The predictions is done both with a 10-fold version and a simple version. This is presented in Section 3.

2 Monopoly

2.1 Dice Generation

The dice generation for the Monopoly game is done by using the randint from python. It will generate the rolling of two numbers between 1 and 6 that will represent the two dice. The code for the dice generation is shown in Figure 1. The code to handle doubles is done during the Monopoly game simulation, the dice generation just simulates the rolling of two dice.

```
def roll_dice():
    dice_1 = randint(1,6)
    dice_2 = randint(1,6)
    return (dice_1,dice_2)
```

Figure 1: The Dice Generator.

2.2 Chance Cards

Chance cards are simulated in the game, with most of them implemented in either the Probability or Multiplayer version of the game. The Get Out of Jail Free and Pay Repair cards are the only cards that are not implemented.

The chance cards are held in a dictionary that represent all the 16 cards in the game. This dictionary contains the name, and the number of times each chance card has been pulled. They are placed in order in a "Deck" and the shuffled at the beginning of the simulation. Once a call is pulled it is placed back at the end of the deck. The code that handles getting a chance card is shown in Figures 2, 3, and 4 below.

```
deck = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

Figure 2: Shuffle the global deck, only called once.

```
def shuffle_chance():
    random.shuffle(deck)
```

Figure 3: Shuffle the global deck, only called once.

```
def get_chance_card(cur_pos,player):
    #Define chance cards here, and draw from them.
    #After drawing the card is placed at the back of
    #'Deck' again.
    card = deck[0]
    #print(deck)

    for i in range(len(deck)-1):
        deck[i] = deck[i+1]
    deck[15] = card
    #print(deck)

    cards = {
    1:  ('Advance to Go', 0),
    2:  ('Advance to Illinois Ave', 0),
    3:  ('Advance to St. Charles Place', 0),
    4:  ('Advance to Nearest Utility', 0),
    5:  ('Advance to nearest Railroad', 0),
    6:  ('Bank Pays You 50', 0),
    7:  ('Get out of Jail Free', 0),
    8:  ('Go Back 3 Spaces', 0),
    9:  ('Go to Jail', 0),
    10: ('Pay Repairs', 0),
    11: ('Pay tax of 15', 0),
    12: ('Go to Reading Railroad', 0),
    13: ('Go to Boardwalk', 0),
    14: ('Pay each player 50', 0),
    15: ('Collect 150', 0),
    16: ('Collect 100', 0),
    }

    cards[card] = (cards[card][0], cards[card][1]+1)

    position = handle_chance(card, cur_pos,player)

    return position
```

Figure 4: The get chance card function.

Aside from the get chance card function, a handle chance function is used in order to handle the card that was pulled. It will return a position to move to if the card yielded a result that requires movement or 0 otherwise. A sample code from the non-multiplayer version is shown in Figure 5 below, the changes for the multiplayer version will be shown in that section.

```python
#Will handle the chance card according to its rule
#will call get chance card and process accordingly
#Takes current position to move backward forward if necessary
#NOTE: Get out of jail is IGNORED.
def handle_chance(card,pos):
    #Go
    if card == 1:
        return 1
    #Illinois
    elif card == 2:
        return 25
    #St Charles
    elif card == 3:
        return 12
    #Nearest Utility
    elif card == 4:
        electric = 13 - pos
        water = 29 - pos
        utilities = [electric, water]
        minimum =  min(utilities)
        if minimum == electric:
            return 13
        else:
            return 29
    #Nearest Railroad
    elif card == 5:
        reading_pos = 6 - pos
        penn_pos = 16 - pos
        bao_pos = 26 - pos
        sho_pos = 36 - pos
        trains = [reading_pos, penn_pos, bao_pos, sho_pos]
        minimum = min(trains)
        if minimum == reading_pos:
            return 6
        elif minimum == penn_pos:
            return 16
        elif minimum == bao_pos:
            return 26
        elif minimum == sho_pos:
            return 36
    #Get 50
    elif card == 6:
        #NOT IMPLEMENTED
        return 0
    #Get out of Jail Free
    elif card == 7:
        #NOT IMPLEMENTED
        return 0
    #Go Back 3
    elif card == 8:
        temp_pos = position-3
        temp_mod = temp_pos%40
        if temp_mod == 0:
            return 40
        else:
            return temp_mod
    #Go to Jail
    elif card == 9:
        return 11
    #Pay Repairs
    elif card == 10:
        #NOT IMPLEMENTED
        return 0
```

Figure 5: The handle chance card function.

2.3 Game Simulation and Probabilities

The probabilities game simulation consists of a play monopoly function. This function contains a dictionary similar to the chance dictionary that contains the landing for each property as well as the name of each space. The Initial shuffle and board dictionary is shown in Figure 6.

The main game logic is contained in a for loop that will keep rolling the dice and counting the spaces landed on, a check for the doubles is done here. Note that for double calculation the player will not be sent to jail after three consecutive doubles, instead the player will keep rolling until they no longer get doubles. The code for the main game logic is shown in Figure 7 below.

Finally, the code that will run the monopoly game 1000 times and get the probabilities is shown in Figure 8 below. The code will run 1000 independent games of monopoly and will keep the probability of all the railroads, the GO square, Mediterranean Avenue, and Boardwalk as required by the assignment. The summary of these landings is presented in Table 1.

Space	Total Landings	Probability of Landing
Reading Railroad	3651	3.65%
Pennsylvania Railroad	3224	3.22%
BAO Railroad	3196	3.19%
Short Line	2812	2.81%
Go	3015	3.01%
Mediterranean Ave	2576	2.57%
Boardwalk	2980	2.98%

Table 1: Probabilities of landing in selected spaces.

```
def play_monopoly():
    #print(deck)
    shuffle_chance()
    #print(deck)
    #Create each monopoly square and hold it
    board = {
    1: ('Go',0),
    2: ('Mediterranean', 0),
    3: ('Community Chest',0),
    4: ('Baltic',0),
    5: ('Income Tax',0),
    6: ('Reading Railroad',0),
    7: ('Oriental Avenue',0),
    8: ('Chance',0),
    9: ('Vermont',0),
    10: ('Connecticut',0),
    11:('Jail',0),
    12:('St. Charles',0),
    13:('Electric Company',0),
    14:('States Avenue',0),
    15:('Virginia Avenue',0),
    16:('Pennsylvania Railroad',0),
    17:('St. James',0),
    18:('Commnity Chest',0),
    19:('Teneesee',0),
    20:('New York',0),
    21:('Free Parking',0),
    22:('Kentucky',0),
    23:('Chance',0),
    24:('Indiana Avenue',0),
    25:('Illinois Avenue',0),
    26:('B.A.O Railroad',0),
    27:('Atlantic',0),
    28:('Venthos',0),
    29:('Water Works',0),
    30:('Maine',0),
    31:('Go To Jail',0),
    32:('Pacific',0),
    33:('North Carolina',0),
    34:('Community Chest',0),
    35:('Pennsylvania',0),
    36:('Short Line',0),
    37:('Chance',0),
    38:('Park Place',0),
    39:('Luxury Tax',0),
    40:('Boardwalk',0),
    }
```

Figure 6: The initial Monopoly function.

```
def is_jail(space):
    return space == 'Go To Jail'


#Create cur_pos with pos 1 to represent Go
cur_pos = 1
#Roll the dice n times
for i in range(100 ):
    #Roll the dices, store positions
    #Mod the position to stay within 40 squares
    dice1, dice2 = roll_dice()
    cur_pos += (dice1 + dice2)
    cur_pos %= 40
    if (cur_pos == 0):
        cur_pos = 40
    #print(cur_pos)
    #Increase counter
    curr_landings = board[cur_pos][1]+1
    #Replace tuple with new increased counter
    board[cur_pos] = (board[cur_pos][0], curr_landings)
    #print(board[cur_pos])
    #Check if jail and increase accordingly
    if is_jail(board[cur_pos][0]):
        cur_pos = 10
        curr_landings = board[cur_pos][1]+1
        board[cur_pos] = (board[cur_pos][0], curr_landings)
        continue
    #check if chance and increase counters with new position
    #of chance card, if any
    if cur_pos == 8 or cur_pos == 23 or cur_pos == 37:
        position = get_chance_card(cur_pos)
        if position != 0:
            cur_pos = position
            curr_landings = board[cur_pos][1]+1
            board[cur_pos] = (board[cur_pos][0], curr_landings)
    #Roll again if doubles
    #NOTE: 3 consecutive will NOT yield jail
    while (dice1 == dice2):
        dice1, dice2 = roll_dice()
        cur_pos += (dice1 + dice2)
        cur_pos %= 40
        if (cur_pos == 0):
            cur_pos = 40
        #print(cur_pos)
        curr_landings = board[cur_pos][1]+1
        board[cur_pos] = (board[cur_pos][0], curr_landings)
        #print(board[cur_pos])
        if is_jail(board[cur_pos][0]):
            cur_pos = 10
            curr_landings = board[cur_pos][1]+1
            board[cur_pos] = (board[cur_pos][0], curr_landings)
            break

#return the final board with all the tallies
return board
```

Figure 7: The main Monopoly loop.

```
#Create list to store landings
reading_railroad = []
pennsylvania_railroad = []
bao_railroad = []
short_line = []
go = []
mediterranean = []
boardwalk = []

#Run the monopoly game 1000 times
for i in range(1000):
    result = play_monopoly()
    #Store Each Probablities
    reading_railroad.append(result[6][1])
    pennsylvania_railroad.append(result[16][1])
    bao_railroad.append(result[26][1])
    short_line.append(result[36][1])
    go.append(result[1][1])
    mediterranean.append(result[2][1])
    boardwalk.append(result[40][1])
```

Figure 8: The 1000 game Monopoly loop.

2.4 Multiplayer Game Simulation

The multiplayer game simulation includes the addition of money to the game, and only accounts for a 2 player game. A main difference from the single player version is in the board, as it now includes an owner, cost of buy and rent. The code of the modified board dictionary is shown in Figure 9.

The simulation accounts for buying properties, this is done by a landing property function, which will semi randomly decide if a player will buy the property of not. It was decided to add a rich flag to add a little bit more realism to the game, this will make the player always buy a property as long as it has over 400 in money. After the player is below this, buying will be decided with a random generation of 1 and 2, 1 representing buy and 2 representing not buy. The code for this function is shown in Figure 10.

The two tax properties are handled in the game as well, initially they are detected by the land on property function and handled by the check tax function, in which the corresponding amount will be deducted from the player. The code of the tax function is shown in Figure 11.

Paying of rent in a simplified way is handled by the game as well, this is done by using the fixed rent price of each property. Multiple colours, house, or hotels are not accounted for. Additionally, multiple railroads or the way landing in a an utility payments are done is not handled instead they each have a fixed rent. The code to handle rent in shown in Figure 12.

A give go function was also added to the game that will give the players 200 when they pass go. This function is called on the main monopoly loop. The give go function is shown in Figure 13. Additionally, some extra functionality that either adds or subtracts money from a player given a chance cards was added, a sample of the code is shown in Figure 14. This code is a modified version of the code shown in Figure 5, it includes more implementations that deal with money.

The main Monopoly loop was modified to account for changed of handling landings, go, ending the game of bankruptcy, and changing the output. The main additions are shown in Figure 15 and 16. Finally, a sample output of the initial game is shown in Figure 17, note that Chance cards results are not printed.

```
    turn = 1
    global player_1
    global player_2
    player_1 = 500
    player_2 = 500
    print(player_1)
    print(player_2)
    shuffle_chance()
    #Create each monopoly square and hold it
    #Board change order is now, Name, Landings, Owner, Cost, Rent
    #Owner of -1 represents not ownable
    #0 = Unowned, 1 or 2 owned by player 1 or 2 respectively
    #Railroads, and Companies have fixed rent of 25
    #DOES NOT account for multiple colors owned
    board = {
    1: ('Go',0,-1,0,0),
    2: ('Mediterranean', 0,0,60,2),
    3: ('Community Chest',0,-1,0,0),
    4: ('Baltic',0,0,60,4),
    5: ('Income Tax',0,-1,200,200),
    6: ('Reading Railroad',0,0,200,25),
    7: ('Oriental Avenue',0,0,100,6),
    8: ('Chance',0,-1,0,0),
    9: ('Vermont',0,0,100,6),
    10: ('Connecticut',0,0,120,8),
    11:('Jail',0,-1,0,0),
    12:('St. Charles',0,0,140,10),
    13:('Electric Company',0,0,150,25),
    14:('States Avenue',0,0,140,10),
    15:('Virginia Avenue',0,0,160,12),
    16:('Pennsylvania Railroad',0,0,200,25),
    17:('St. James',0,0,180,14),
    18:('Commnity Chest',0,-1,0,0),
    19:('Teneesee',0,0,180,14),
    20:('New York',0,0,200,16),
    21:('Free Parking',0,-1,0,0),
    22:('Kentucky',0,0,220,18),
    23:('Chance',0,-1,0,0),
    24:('Indiana Avenue',0,0,220,19),
    25:('Illinois Avenue',0,0,240,20),
    26:('B.A.O Railroad',0,0,200,25),
    27:('Atlantic',0,0,260,22),
    28:('Venthos',0,0,260,22),
    29:('Water Works',0,0,150,25),
    30:('Maine',0,0,280,24),
    31:('Go To Jail',0,-1,0,0),
    32:('Pacific',0,0,300,26),
    33:('North Carolina',0,0,300,26),
    34:('Community Chest',0,-1,0,0),
    35:('Pennsylvania',0,0,320,28),
    36:('Short Line',0,0,200,25),
    37:('Chance',0,-1,0,0),
    38:('Park Place',0,0,350,35),
    39:('Luxury Tax',0,-1,75,75),
    40:('Boardwalk',0,0,400,50),
    }
```

Figure 9: The Multiplayer board.

```
def land_on_property(space, player):
    global player_1
    global player_2
    if board[space][2] == -1:
        check_tax(space, player)

    #Handle rent payment
    if board[space][2] > 0 and board[space][2] != player:
            pay_rent(space,player)
            return

    #Add an extra rich flag
    #In Which a player will always buy if money is over 400
    rich = False
    if player == 1:
        if  player_1 > 400:
            rich = True
    elif player == 2:
        if player_2 > 400:
            rich = True
    #Handle to Buy
    #Will randombly decide to buy or not to buy
    if board[space][2] != -1:
        to_buy = randint(1,2)
        if to_buy == 1 or rich:
            if board[space][2] == 0:
                if player == 1:
                    if player_1 >= board[space][3]:
                        player_1 -= board[space][3]
                        board[space] = (board[space][0],board[space][1],player,board[space][3],board[space][4])
                        print("Player 1 buys property " + board[space][0])
                        print("Player 1 money is now: " + str(player_1))
                else:
                    if player_2 >= board[space][3]:
                        player_2 -= board[space][3]
                        board[space] = (board[space][0],board[space][1],player,board[space][3],board[space][4])
                        print("Player 2 buys property " + board[space][0])
                        print("Player 2 money is now: " + str(player_2))
        elif to_buy == 2:
            print("Player " + str(player) + " decides not to buy property " + board[space][0])
```

Figure 10: The land on property handler.

```
def check_tax(space,player_to_pay):
    global player_1
    global player_2
    #income Tax
    if space == 5:
        if player_to_pay == 1:
            player_1 -= board[space][4]
            print("Player 1 pays Income Tax of 200")
        else:
            player_2 -= board[space][4]
            print("Player 2 pays Income Tax of 200")
    #Luxury Tax
    elif space == 39:
        if player_to_pay == 1:
            player_1 -= board[space][4]
            print("Player 1 pays Luxury Tax of 75")
            print("Player 1 money is now: " + str(player_1))
        else:
            player_2 -= board[space][4]
            print("Player 2 pays Income Tax of 75")
            print("Player 2 money is now: " + str(player_2))
```

Figure 11: The tax function.

```
def pay_rent(space,player_to_pay):
    global player_1
    global player_2
    rent = board[space][4]
    if player_to_pay == 1:
        player_1 -= rent
        player_2 += rent
        print("Player 1 pays Player 2, " + str(rent) + "for landing on " + board[space][0])
        print("Player 1 money is now: " + str(player_1))
    else:
        player_2 -= rent
        player_1 += rent
        print("Player 2 pays Player 1, " + str(rent) + "for landing on " + board[space][0])
        print("Player 2 money is now: " + str(player_2))
```

Figure 12: The rent function.

```
def give_go(player):
    global player_1
    global player_2
    if player == 1:
        player_1 += 200
        print("Player 1 receives 200 for passing Go")
        print("Player 1 money is now: " + str(player_1))
    else:
        player_2 += 200
        print("Player 2 receives 200 for passing Go")
        print("Player 2 money is now: " + str(player_2))
```

Figure 13: The GO function.

```
    return 40
#------ The Collect or Pay Cards ---------
#Pay each player 50
elif card == 14:
    if player == 1:
        player_1 -= 50
        player_2 += 50
    else:
        player_2 -= 50
        player_1 += 50
    return 0
#Collect 150
elif card == 15:
    if player == 1:
        player_1 += 150
    else:
        player_2 += 150
    return 0
#Collect 100
elif card == 16:
    if player == 1:
        player_1 += 100
    else:
        player_2 += 100
    return 0
```

Figure 14: The Collect and Pay Implementations.

```
def check_bankrupt():
    if player_1 < 0:
        return 2
    elif player_2 <0:
        return 1
    else:
        return 0

def get_winner():
    if player_1 > player_2:
        print("Player 1 wins with $" + str(player_1) + " over $" + str(player_2) + " of player 2.")
    else:
        print("Player 2 wins with $" + str(player_2) + " over $" + str(player_1) + " of player 1.")


#Create cur_pos with pos 1 to represent Go
cur_pos = [0,1,1]
#Roll the dice n times
for i in range(200):
    #Roll the dices, store positions
    #Mod the position to stay within 40 squares
    dice1, dice2 = roll_dice()
    cur_pos[turn] += (dice1 + dice2)
    print("Player " + str(turn) + " rolls " + str(dice1+dice2))
    if cur_pos[turn] > 40:
        give_go(turn)
    cur_pos[turn] %= 40
    if (cur_pos[turn] == 0):
        cur_pos[turn] = 40
    print("Player " + str(turn) + " lands on " + board[cur_pos[turn]][0])
    land_on_property(cur_pos[turn],turn)
```

Figure 15: Additions of main monopoly loop.

```
    if check_bankrupt == 1:
        print("Player 1 wins the game")
        break
    elif check_bankrupt == 2:
        print("Player 2 wins the game")
        break
    if turn == 1:
        turn = 2
    else:
        turn = 1

#return the final board with all the tallies
get_winner()
return board
```

Figure 16: The end game additions.

```
rafa@w134-87-148-024 : ~/Documents/GitHub/CSC421/A2 python3
500
500
Player 1 rolls 7
Player 1 lands on Chance
Player 2 rolls 7
Player 2 lands on Chance
Player 2 buys property St. Charles
Player 2 money is now: 360
Player 1 rolls 3
Player 1 lands on Jail
Player 2 rolls 3
Player 2 lands on Virginia Avenue
Player 2 decides not to buy property Virginia Avenue
Player 1 rolls 7
Player 1 lands on Commnity Chest
Player 2 rolls 7
Player 2 lands on Kentucky
Player 2 buys property Kentucky
Player 2 money is now: 140
Player 1 rolls 7
Player 1 lands on Illinois Avenue
Player 1 buys property Illinois Avenue
Player 1 money is now: 245
Player 2 rolls 7
Player 2 lands on Water Works
Player 2 decides not to buy property Water Works
Player 1 rolls 5
Player 1 lands on Maine
Player 2 rolls 8
Player 2 lands on Chance
Player 1 rolls 8
Player 1 lands on Park Place
Player 1 decides not to buy property Park Place
Player 2 rolls 9
Player 2 receives 200 for passing Go
Player 2 money is now: 440
```

Figure 17: The sample output of two player Monopoly.

3 Naive Bayes

3.1 Word Probabilities

The code that parses the text files and stores them in Vector representation is shown in Figure 18. Note that only the code for the positive reviews is shown, the negative reviews is exactly the same with the only difference of reading txt files from the neg folder.

The probabilities for each dictionary word in the positive and negative reviews are calculated by the code shown in Figure 19. Further, the result of this calculations is shown in Figure 20.

```
path_pos = "/Users/rafa/Downloads/review_polarity/txt_sentoken/pos/"
path_neg = "/Users/rafa/Downloads/review_polarity/txt_sentoken/neg/"
positive_reviews = []
negative_reviews = []
positive_vectors=[]
negative_vectors=[]
words = ["awful","bad","boring","dull","effective","enjoyable","great","hilarious"]

i=0
for pos_reviews_file in os.listdir(path_pos):
    #Append Path
    positive_reviews.append(path_pos+pos_reviews_file)
    #Read the file
    pos_review = open(path_pos+pos_reviews_file, "r")
    #Create initial Vector
    vector = [0,0,0,0,0,0,0,0]
    #Append empty Vector
    positive_vectors.append(vector)
    #Read file line by line
    #And change corresponing position according to words
    for line in pos_review:
        for word in line.split():
            if word.lower() == "awful":
                positive_vectors[i][0] = 1
            elif word.lower() == "bad":
                positive_vectors[i][1] = 1
            elif word.lower() == "boring":
                positive_vectors[i][2] = 1
            elif word.lower() == "dull":
                positive_vectors[i][3] = 1
            elif word.lower() == "effective":
                positive_vectors[i][4] = 1
            elif word.lower() == "enjoyable":
                positive_vectors[i][5] = 1
            elif word.lower() == "great":
                positive_vectors[i][6] = 1
            elif word.lower() == "hilarious":
                positive_vectors[i][7] = 1
    i+=1
```

Figure 18: Parsing the txt files.

```
#Turn into Numpy Arrays for easier processing
positive = np.asarray(positive_vectors)
negative = np.asarray(negative_vectors)

#Get the Probabilities of each word
probs_pos = (positive.sum(axis=0).astype(float)+1.0)/1000
probs_neg = (negative.sum(axis=0).astype(float)+1.0)/1000

#Print them
print(probs_pos)
print(words)

print(probs_neg)
print(words)
```

Figure 19: Calculating the probabilities for each word.

```
[rafa@w134-87-148-024 : ~/Documents/GitHub/CSC421/A2 python3 movies.py
[0.02  0.256 0.049 0.024 0.121 0.096 0.409 0.126]
['awful', 'bad', 'boring', 'dull', 'effective', 'enjoyable', 'great', 'hilarious']
[0.102 0.506 0.17  0.092 0.047 0.054 0.287 0.051]
['awful', 'bad', 'boring', 'dull', 'effective', 'enjoyable', 'great', 'hilarious']
```

Figure 20: The result of the probabilities.

3.2 Naive Bayes Classifier

The probabilities estimates can be combined to form a Naive Bayes classifier in the sense that they are a binary representation, therefore we don't care on the total number of occurrences of each word but instead on the sole occurrence of each word. We calculate the likelihood of each word using a simplified Bayes in which we only use probability of the word appearing. This is done in the likelihood function shown in Figure 21.

```
def likelihood(review, probs_for_type):
    probability_product = 1.0
    for (i,w) in enumerate(review):
        if (w==1):
            probability=probs_for_type[i]
        else:
            probability= 1.0 - probs_for_type[i]
        probability_product *= probability
    return probability_product
```

Figure 21: The likelihood calculation, using Naive Bayes.

3.3 Classification Accuracy and Confusion Matrix

The confusion matrix for this problem is shown in Table 2. Further, the code to calculate the classification accuracy for this problem is shown in Figure 22. The results obtained by these code

Real\Believe	True (Positive)	False (Negative)
True (Positive)	756	244
False (Negative)	408	592

Table 2: Confusion Matrix.

are a classification accuracy of 75.6% for the positive review set and 59.2% for the negative reviews set.

```
def predict(review):
    scores = [likelihood(review,probs_pos), likelihood(review,probs_neg)]
    labels = ['positive', 'negative']
    return labels[np.argmax(scores)]

def predict_set(test_set, truth_label):
    score = 0
    for r in test_set:
        if predict(r) == truth_label:
            score+=1
    return score/10.0

print(predict_set(positive,'positive'))
print(predict_set(negative,'negative'))
```

Figure 22: Calculate the accuracy of each review prediction.

3.4 10-fold cross-validation

The 10-fold version of the code, yielded better accuracy for the predictions. First, the code is parsed in a way to account for the 000 - 099, way of creating the folds for the positive and negative reviews. The code of the parsing is shown in Figure 23. Further, probabilities for each fold, are calculated in a similar matter to Figure 19, using numpy. Finally, the likelihood, predict, and predict_set were modified in order to account for a fold parameter, to account for the current fold being used, this way the correct probabilities associated with that fold is used, that was the only change done to this functions, nevertheless the code is shown in Figure 24.

To get the total accuracy, the individual accuracy per fold is stored in a list, this is then averaged and we get the total accuracy. The code that calculated this is shown in Figure 25. The results obtained by these code is shown in Figure 26. Finally, the confusion matrix for the 10-fold version is shown in Table 3.

Real\Believe	True (Positive)	False (Negative)
True (Positive)	757	243
False (Negative)	381	619

Table 3: Confusion Matrix.

```
for pos_reviews_file in os.listdir(path_pos):
    #Regex to get only the number associated to file
    m = re.search('(?<=cv)(.*)(?=_)',pos_reviews_file)
    num = int(m.group())

    all_reviews.append(path_pos+pos_reviews_file)
    positive_reviews.append(path_pos+pos_reviews_file)
    pos_review = open(path_pos+pos_reviews_file, "r")
    vector = [0,0,0,0,0,0,0,0]

    #Append to correct fold
    fold = 0
    if num <= 99:
        folds_pos[0].append(vector)
        fold = 0
    elif num >99 and num <= 199:
        folds_pos[1].append(vector)
        fold = 1
    elif num > 199 and num <= 299:
        folds_pos[2].append(vector)
        fold = 2
    elif num > 299 and num <= 399:
        folds_pos[3].append(vector)
        fold = 3
    elif num > 399 and num <= 499:
        folds_pos[4].append(vector)
        fold = 4
    elif num > 499 and num <= 599:
        folds_pos[5].append(vector)
        fold = 5
    elif num > 599 and num <= 699:
        folds_pos[6].append(vector)
        fold = 6
    elif num > 699 and num <= 799:
        folds_pos[7].append(vector)
        fold = 7
    elif num > 799 and num <= 899:
        folds_pos[8].append(vector)
        fold = 8
    elif num > 899 and num <= 999:
        folds_pos[9].append(vector)
        fold = 9

    #Update vector in corresponding fold
    for line in pos_review:
        for word in line.split():
            if word.lower() == "awful":
                folds_pos[fold][len(folds_pos[fold])-1][0] = 1
            elif word.lower() == "bad":
                folds_pos[fold][len(folds_pos[fold])-1][1] = 1
            elif word.lower() == "boring":
                folds_pos[fold][len(folds_pos[fold])-1][2] = 1
            elif word.lower() == "dull":
                folds_pos[fold][len(folds_pos[fold])-1][3] = 1
            elif word.lower() == "effective":
                folds_pos[fold][len(folds_pos[fold])-1][4] = 1
            elif word.lower() == "enjoyable":
                folds_pos[fold][len(folds_pos[fold])-1][5] = 1
            elif word.lower() == "great":
                folds_pos[fold][len(folds_pos[fold])-1][6] = 1
            elif word.lower() == "hilarious":
                folds_pos[fold][len(folds_pos[fold])-1][7] = 1
```

Figure 23: Distributing vectors, into 10-folds.

```
def likelihood(review, probs_for_type,fold):
    probability_product = 1.0
    for (i,w) in enumerate(review):
        if (w==1):
            probability=probs_for_type[fold][i]
        else:
            probability= 1.0 - probs_for_type[fold][i]
        probability_product *= probability
    return probability_product


def predict(review,fold):
    scores = [likelihood(review,probs_pos,fold), likelihood(review,probs_neg,fold)]
    labels = ['positive', 'negative']
    return labels[np.argmax(scores)]

def predict_set(test_set, truth_label,fold):
    score = 0
    for r in test_set:
        if predict(r,fold) == truth_label:
            score+=1
    return score
```

Figure 24: Updated code to include folds.

```
averages_positive = []
averages_negative = []

print("Calculating Positive Reviews 10-fold")
for i in range(10):
    pos_score = predict_set(folds_pos[i],'positive',i)
    averages_positive.append(pos_score)
    print("Fold " + str(i) + " accuracy: " + str(pos_score))

print("Calculating Negative Reviews 10-fold")
for j in range(10):
    neg_score = predict_set(folds_neg[j],'negative',j)
    averages_negative.append(neg_score)
    print("Fold " + str(j) + " accuracy: " + str(neg_score))


print("Average Positive Review Accuracy")
print(np.mean(averages_positive))
print("Average Negative Review Accuracy")
print(np.mean(averages_negative))
```

Figure 25: The 10-fold accuracy calculations.

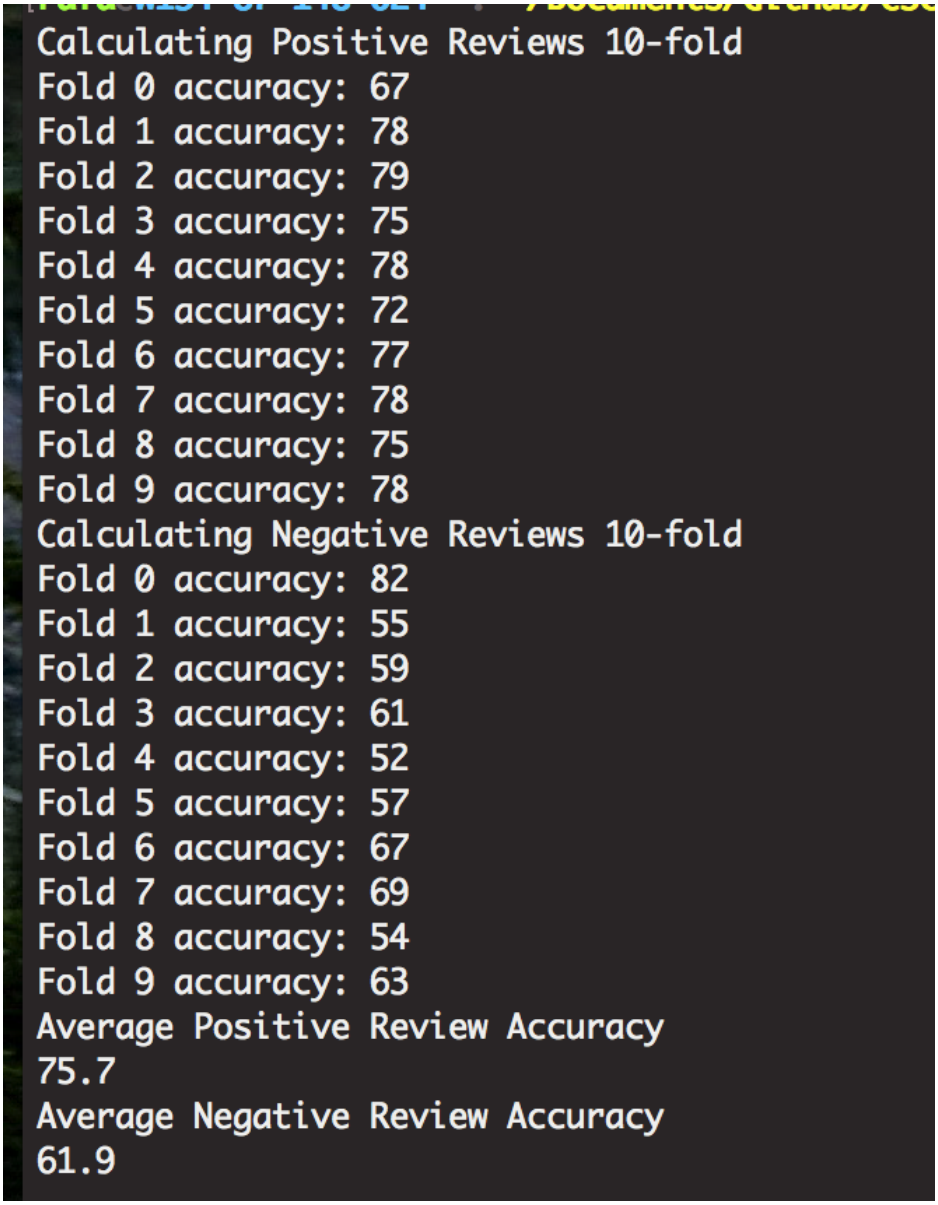

Figure 26: The 10-fold results.