Assignment 2_ Mahsa Daneshmandmehrabani_V00892119
Coded in Python

Question 1: Monopoly

Explain how to calculate the cummulative distribution in O(k) time and how to generate a single sample X from it.
Answer:
In probability theory, the cumulative distribution function (CDF, also cumulative density function) of a random variable *X*, evaluated at *x*, is the probability that *X* will take a value less than or equal to *x*.

$$F_X (x) = P(X \leq x)$$

If X is a discrete random variable that it takes each $x_1, ..., x_k$ values each with probability P(X = x_i) = p_i, then the CFD of X will be:

$$F_X (x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i) = \sum_{x_i \leq x} p_i$$

From above equation, it is clear that cummulative distribution function can be calculated in O(k) time. To generate a random sample X by using its CDF function, generate a random number between 0, 1 and then compare the random generated number to the CDF intervals and then decide what number assign to X.

1. Write code to generate random numbers corresponding to rolling a pair of dice and summing the ouput taking into account the doubling rule (when you roll doubles you get to roll again). Include all the code with comments describing what you are doing.
Answer:

I defined the *Dice* class with having the *roll* function that simulates rolling two dices. *Roll* function of class *Dice* returns a pair of dice. I defined the PMF variable in it that means the probability mass function that keeps the probability that a dice is exactly equal to some value between 1 and 6. Then I used the numpy function ***cumsum*** that returns the cumulative sum of the elements along a given axis of the given array. Then I generated two random numbers *r1* and *r2* which are between 0, 1. Then by using *searchsorted* function of numpy, I assign a value to *dice1* and *dice2. searchsorted* finds the indices into PMF array such that, if the corresponding element r1 were inserted before the indices, the order of PMF would be preserved (It actually compares r1 or r2 with each CDF intervals and then decides which value shoud be assigned.

```
def roll(self):
            # Probability mass function that stores the probability that a
dice is exactly equal to some value from 1 to 6
```

```
            PMF = [1/6 for j in range(1,7)]

            #CDF is the cummulative distribution function of random
variable dice
            CDF = np.cumsum(PMF)

            r1 = np.random.rand(1)
            r2 = np.random.rand(1)

            dice1 = np.searchsorted(CDF, r1) + 1
            dice2 = np.searchsorted(CDF, r2) + 1

            return dice1[0],dice2[0]
```

Then I check the rules of rolling dices in the *WhereToGo* method of class *Board.* In the *Board* class, I defined three class variables: *self.BoardConfig = np.zeros(40), self.CurrentPosition = 0, self.doubleCounts = 0.*

self.BoardConfig: An array that stores the number of landing on each square
self.CurrentPosition: Represents the current position of the player
self.doubleCounts: counts the number of double rolls

In Board class, there are 3 methods:

- ***SumTwoDice***, that takes two dices and returns the sum
- ***WhereToGo***, takes two dices and takes care of double rules and updates the *self.CurrentPosition class variable based on sum of the two dices and also some special currents squares such as chance square.* In other words, *self.CurrentPosition* is updated in this method based on the sum of the two dices or if we are on a chance square and from the chance square we are sent to other square.
- ***Get_board_ferequency***, just returns the self.BoardConfig *class variable*
-

Double rolls:

In ***WhereToGo*** method, the part that takes care about the double rolls are brought in the following and the steps are explained as comments in my code:

```
def WhereToGo(self, dice1, dice2):

        C = Chance()

        if dice1 == dice2:
            # If the dices are the same, increment the double counter
            self.doubleCounts = self.doubleCounts + 1


            if self.doubleCounts == 3:
                # In this case, you rolled 3 consecutive doubles, so you
should go to the jail, but we should reset the counter to 0
                self.doubleCounts = 0
                #Increment the jail position
                self.BoardConfig[10] = self.BoardConfig[10] + 1
                #Now you are in jail
                self.CurrentPosition = 10


            elif 0 < self.doubleCounts < 3:

                #If the number of rolls so far is less than 3 but greater
than 0, that's ok, you just need to move by the sum of the dices, we need
to take mod 40 to be able to continue the game from begining
                self.CurrentPosition = (self.CurrentPosition +
self.SumTwoDice(dice1, dice2)) % 40
                #print('After your move you are at: ',
self.CurrentPosition)
                self.BoardConfig[int(self.CurrentPosition)] =
self.BoardConfig[int(self.CurrentPosition)] + 1
                #print('freq of your current place: ',
self.BoardConfig[int(self.CurrentPosition)])
        if dice1 != dice2 :
                #print('your die: ', dice1, dice2)
                self.doubleCounts = 0

                self.CurrentPosition = (self.CurrentPosition +
self.SumTwoDice(dice1, dice2)) % 40

                self.BoardConfig[int(self.CurrentPosition)] =
self.BoardConfig[int(self.CurrentPosition)] + 1
```

Write code for selecting a chance card at random based on the information provided in the article about the math of Monopoly.

Similar to rolling the dice, I used the CDF function for chance cards to simulate the chance cards. I defined the ***Chance*** class with *self.ChanceCards* as the class variable that is a list of the chance cards that I care about them for this assignment (the ones that advance the player to another square). Then I defined the ***chancecard()*** method that returns a card from those we care about them by using CDF function. The code is brought in the following:

```
class Chance():
    def __init__(self):
        self.ChanceCards = ["Advance to Go", "Advance to Illinois Ave",
"Advance to St. Charles Place", "Advance token to nearest Utility",
"Advance token to the nearest Railroad", "Go Back 3 Spaces", "Take a trip
to Reading Railroad", "Advance token to Boardwalk", "get out of jail","Go
to jail", "Go Back 3 Spaces"]

    def chancecard(self):
            # define the mass probability function for chance cards, 10/16
is the probability of cards that we care about them, 6/16 is the
probability of cards that we do not care about them (collecting or paying
money)
            PMF = [10/16, 6/16]

            #Cumulative distribution function
            self.CDF = np.cumsum(PMF)
            c = np.random.rand(1)

            #Find the indices into the sorted array self.CDF such that, if
the corresponding element c were inserted before the indices, the order of
a would be preserved.
            chancecard = np.searchsorted(self.CDF, c)
            if (chancecard[0] == 0):

               index = np.random.randint(0, len(self.ChanceCards))
               self.yourCard = self.ChanceCards[index]

            elif (chancecard[0] == 1):
               # So if the chancecard is not among cards that we care about
output NoCard
               self.yourCard = "NoCard"

            return self.yourCard
```

Then again in the ***WhereToGo*** method of the board class, I created a chance object and then I check if I am currently on one of the chance squares, depending on which chance square we are on it, I decide where should the player go. The corresponding code is brought in the following:

```
def WhereToGo(self, dice1, dice2):

        C = Chance()

        if dice1 == dice2:
            # If the dices are the same, increment the double counter
            self.doubleCounts = self.doubleCounts + 1


            if self.doubleCounts == 3:
                # In this case, you rolled 3 consecutive doubles, so you 
should go to the jail, but we should reset the counter to 0
                self.doubleCounts = 0
                #Increment the jail position
                self.BoardConfig[10] = self.BoardConfig[10] + 1
                #Now you are in jail
                self.CurrentPosition = 10


            elif 0 < self.doubleCounts < 3:

                #If the number of rolls so far is less than 3 but greater 
than 0, that's ok, you just need to move by the sum of the dices, we need 
to take mod 40 to be able to continue the game from begining
                self.CurrentPosition = (self.CurrentPosition + 
self.SumTwoDice(dice1, dice2)) % 40
                #print('After your move you are at: ', 
self.CurrentPosition)
                self.BoardConfig[int(self.CurrentPosition)] = 
self.BoardConfig[int(self.CurrentPosition)] + 1
                #print('freq of your current place: ', 
self.BoardConfig[int(self.CurrentPosition)])
        if dice1 != dice2 :
                #print('your die: ', dice1, dice2)
                self.doubleCounts = 0

                self.CurrentPosition = (self.CurrentPosition + 
self.SumTwoDice(dice1, dice2)) % 40
```

```
                self.BoardConfig[int(self.CurrentPosition)] =
self.BoardConfig[int(self.CurrentPosition)] + 1


        # If your current position is on chance square (positions 7,22,36)
you should decide where to go based on chance card you choose
        if (self.CurrentPosition == 7 or self.CurrentPosition == 22 or
self.CurrentPosition == 36):
            #print('checking chance: ', self.CurrentPosition)
            if C.chancecard() == "Advance to Go":
                #Go square is at position 0
                self.BoardConfig[0] = self.BoardConfig[0] + 1
                self.CurrentPosition = 0

            if C.chancecard() == "Advance to Illinois Ave":
                #Illinois Ave is at position 24
                self.BoardConfig[24] = self.BoardConfig[24] + 1
                self.CurrentPosition = 24

            if C.chancecard() == "Advance to St. Charles Place":
                #St. Charles Place is at square 11
                self.BoardConfig[11] = self.BoardConfig[11] + 1
                self.CurrentPosition = 11

            if C.chancecard() == "Advance token to the nearest Railroad":
                if self.CurrentPosition == 7:
                    # nearest utility to chance square at position 7, is
square 29

                    self.BoardConfig[5] = self.BoardConfig[5] + 1
                    self.CurrentPosition = 5

                if self.CurrentPosition == 22:
                    # nearest utility to chance square at position 22, is
square 25

                    self.BoardConfig[25] = self.BoardConfig[25] + 1
                    self.CurrentPosition = 25

                if self.CurrentPosition == 36:
                    # nearest utility to chance square at position 36, is
square 29

                    self.BoardConfig[35] = self.BoardConfig[35] + 1
                    self.CurrentPosition = 35
            if C.chancecard() == "Go Back 3 Spaces":
                #We just need go back by 3 steps
                b = self.CurrentPosition - 3
                self.BoardConfig[int(b)] = self.BoardConfig[int(b)] + 1
                self.CurrentPosition = b


            if C.chancecard() == "Advance token to nearest Utility":
                if self.CurrentPosition == 7:
                    # nearest utility to chance square at position 8, is
square 29

                    self.BoardConfig[12] = self.BoardConfig[12] + 1
                    self.CurrentPosition = 12

                if self.CurrentPosition == 22:
                    # nearest utility to chance square at position 23, is
square 29

                    self.BoardConfig[28] = self.BoardConfig[28] + 1
                    self.CurrentPosition = 28

                if self.CurrentPosition == 36:
                    # nearest utility to chance square at position 37, is
square 29

                    self.BoardConfig[28] = self.BoardConfig[28] + 1
                    self.CurrentPosition = 28

            if C.chancecard() == "Take a trip to Reading Railroad":
                self.BoardConfig[6] = self.BoardConfig[6] + 1
                self.CurrentPosition = 6
            if C.chancecard() == "Advance token to Boardwalk":
                self.BoardConfig[39] = self.BoardConfig[39] + 1
                self.CurrentPosition = 39
            if C.chancecard() == "Go to jail":
                self.BoardConfig[10] = self.BoardConfig[10] + 1
                self.CurrentPosition = 10
            if C.chancecard() == "get out of jail":
                self.CurrentPosition = self.CurrentPosition

        if (self.CurrentPosition == 30):
            self.CurrentPosition = 10
            self.BoardConfig[10] = self.BoardConfig[10] + 1
        return self.BoardConfig
```

Example of output of the chance card for calling its function for 15 times:

```
The chance card is:  Go Back 3 Spaces
The chance card is:  Go Back 3 Spaces
The chance card is:  Advance token to Boardwalk
The chance card is:  Go to jail
The chance card is:  NoCard
The chance card is:  Advance to St. Charles Place
The chance card is:  get out of jail
The chance card is:  Go Back 3 Spaces
The chance card is:  get out of jail
The chance card is:  NoCard
The chance card is:  Advance to Go
The chance card is:  NoCard
The chance card is:  get out of jail
The chance card is:  Advance token to the nearest Railroad
The chance card is:  Advance token to Boardwalk
```

Write code for simulating a game of Monopoly with a single player, go to jail, and chance cards. Record how many times you land on each square after playing a game consisting of 100 moves. Run the simulation 1000 times and average the landing results. Show in a table the following probabilities: each railway station, the GO square, Mediterenean Avenue and Boardwalk.

Another important class that I have is the ***Play***() class. It has a ***self.moves*** class variable that takes care of the number of moves (here is 100) and counts the double rolls (less than 3 times) as one move. Then there is a *PlayMove* method that returns the array which has the number of frequencies of each square for each play. In *PlayMove,* I create a board class to play. Then for each num_moves which is less than ***self.moves,*** a dice object is created and then the roll method is called. I pass these two dices to *WhereToGo* and then I get the the boardconfig array of the play by calling get_board_ferequency() method. Then in the main function, I define the simulation and moves variables (for our problem they are 1000 and 100 respectively). Then in a for loop, I create a play object and get its boardconfig array and append it to the Matrix. Then I take the average number of landing on each square by *np.mean* function of numpy.

code:

```
class Play():
     def __init__(self, moves):

         self.moves = moves

     def PlayMove(self):

         board = Board()
         num_moves = 0

         while (num_moves < self.moves):
             #print('time: ', num_moves)
             dice = Dice()
             [dice1, dice2] = dice.roll()

             pair = [dice1, dice2]
             #print('pairof dice: ', pair)
             if pair[0] != pair[1]:
                 #increment just when two dices are not the same, in this case
the double rolls will
                 num_moves = num_moves + 1
             board.WhereToGo(pair[0], pair[1])
             boardconfig = board.get_board_ferequency()
             #print('board: ', boardconfig)
         return boardconfig



if __name__ == '__main__':

    simulation = 1000
    moves = 100

    #Matrix is an array that has all the boardconfigs of all plays in it.
    Matrix = []

    for i in range(simulation):
        play = Play(moves)
        config = play.PlayMove()
        Matrix.append(config)


    Probabilities = np.mean(Matrix, axis=0)
    print(Probabilities)
    print('Average of number of landing on GO square: ', Probabilities[0])
    print('Average of number of landing on Mediterenean Avenue square: ',
Probabilities[1])
    print('Average of number of landing on Boardwalk square: ',
Probabilities[30])
    print('Average of number of landing on Pensilvania rail road square: ',
Probabilities[15])
    print('Average of number of landing on Reading rail square: ',
Probabilities[5])
    print('Average of number of landing on short line rail road square: ',
Probabilities[35])
    print('Average of number of landing on B&O rail road square: ',
Probabilities[25])
```

Output:

```
Average of number of landing on GO square:  3.173
Average of number of landing on Mediterenean Avenue square:  2.547
Average of number of landing on Boardwalk square:  3.025
Average of number of landing on Pensilvania rail road square:  3.178
Average of number of landing on Reading rail square:  2.865
Average of number of landing on short line rail road square:  2.966
Average of number of landing on B&O rail road square:  3.411
```

All codes together:

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jun  9 12:54:01 2018

@author: mahsa
"""


import numpy
```

```python
import numpy as np


class Dice():
     def __init__(self):

            self.face = 6

     def roll(self):
            # Probability mass function that stores the probability
that a dice is exactly equal to some value from 1 to 6
            PMF = [1/6 for j in range(1,7)]

            #CDF is the cummulative distribution function of random
variable dice
            CDF = np.cumsum(PMF)

            r1 = np.random.rand(1)
            r2 = np.random.rand(1)

            dice1 = np.searchsorted(CDF, r1) + 1
            dice2 = np.searchsorted(CDF, r2) + 1

            d1 = dice1[0]
            d2 = dice2[0]

            return (d1,d2)




class Chance():
    def __init__(self):
        self.ChanceCards = ["Advance to Go", "Advance to Illinois
Ave", "Advance to St. Charles Place", "Advance token to nearest
Utility", "Advance token to the nearest Railroad", "Go Back 3
Spaces", "Take a trip to Reading Railroad", "Advance token to
Boardwalk", "get out of jail","Go to jail", "Go Back 3 Spaces"]
```

```
    def chancecard(self):
            # define the mass probability function for chance cards,
10/16 is the probability of cards that we care about them, 6/16 is
the probability of cards that we do not care about them (collecting
or paying money)
            PMF = [10/16, 6/16]

            #Cumulative distribution function
            self.CDF = np.cumsum(PMF)
            c = np.random.rand(1)

            #Find the indices into the sorted array self.CDF such
that, if the corresponding element c were inserted before the
indices, the order of a would be preserved.
            chancecard = np.searchsorted(self.CDF, c)
            if (chancecard[0] == 0):

               index = np.random.randint(0, len(self.ChanceCards))
               self.yourCard = self.ChanceCards[index]

            elif (chancecard[0] == 1):
               # So if the chancecard is not among cards that we care
about output NoCard
               self.yourCard = "NoCard"

            return self.yourCard

class Board():
    def __init__(self):
        self.BoardConfig = np.zeros(40)
        self.CurrentPosition = 0
        self.doubleCounts = 0

    def SumTwoDice(self, dice1, dice2):
        return dice1 + dice2
    #WhereToGo returns the board configuration that stores the number
of times we landed at each square
    def WhereToGo(self, dice1, dice2):

        C = Chance()

        if dice1 == dice2:
            # If the dices are the same, increment the double counter
            self.doubleCounts = self.doubleCounts + 1


            if self.doubleCounts == 3:
                # In this case, you rolled 3 consecutive doubles, so
you should go to the jail, but we should reset the counter to 0
                self.doubleCounts = 0
                #Increment the jail position
                self.BoardConfig[10] = self.BoardConfig[10] + 1
                #Now you are in jail
                self.CurrentPosition = 10


            elif 0 < self.doubleCounts < 3:

                #If the number of rolls so far is less than 3 but
greater than 0, that's ok, you just need to move by the sum of the
dices, we need to take mod 40 to be able to continue the game from
begining
                self.CurrentPosition = (self.CurrentPosition +
self.SumTwoDice(dice1, dice2)) % 40
                #print('After your move you are at: ',
self.CurrentPosition)
                self.BoardConfig[int(self.CurrentPosition)] =
self.BoardConfig[int(self.CurrentPosition)] + 1
                #print('freq of your current place: ',
self.BoardConfig[int(self.CurrentPosition)])
        if dice1 != dice2 :
                #print('your die: ', dice1, dice2)
                self.doubleCounts = 0
                self.CurrentPosition = (self.CurrentPosition +
self.SumTwoDice(dice1, dice2)) % 40

                self.BoardConfig[int(self.CurrentPosition)] =
self.BoardConfig[int(self.CurrentPosition)] + 1


        # If your current position is on chance square (positions
7,22,36) you should decide where to go based on chance card you
choose
        if (self.CurrentPosition == 7 or self.CurrentPosition == 22
or self.CurrentPosition == 36):
            #print('checking chance: ', self.CurrentPosition)
            if C.chancecard() == "Advance to Go":
                #Go square is at position 0
                self.BoardConfig[0] = self.BoardConfig[0] + 1
                self.CurrentPosition = 0

            if C.chancecard() == "Advance to Illinois Ave":
                #Illinois Ave is at position 24
                self.BoardConfig[24] = self.BoardConfig[24] + 1
                self.CurrentPosition = 24

            if C.chancecard() == "Advance to St. Charles Place":
                #St. Charles Place is at square 11
                self.BoardConfig[11] = self.BoardConfig[11] + 1
                self.CurrentPosition = 11

            if C.chancecard() == "Advance token to the nearest
Railroad":
                if self.CurrentPosition == 7:
                    # nearest utility to chance square at position 7,
is square 29
                    self.BoardConfig[5] = self.BoardConfig[5] + 1
                    self.CurrentPosition = 5

                if self.CurrentPosition == 22:
                    # nearest utility to chance square at position
22, is square 25
                    self.BoardConfig[25] = self.BoardConfig[25] + 1
                    self.CurrentPosition = 25

                if self.CurrentPosition == 36:
                    # nearest utility to chance square at position
36, is square 29
                    self.BoardConfig[35] = self.BoardConfig[35] + 1
                    self.CurrentPosition = 35

            if C.chancecard() == "Go Back 3 Spaces":
                #We just need go back by 3 steps
                b = self.CurrentPosition - 3
                self.BoardConfig[int(b)] = self.BoardConfig[int(b)] +
1
                self.CurrentPosition = b


            if C.chancecard() == "Advance token to nearest Utility":
                if self.CurrentPosition == 7:
                    # nearest utility to chance square at position 8,
is square 29
                    self.BoardConfig[12] = self.BoardConfig[12] + 1
                    self.CurrentPosition = 12

                if self.CurrentPosition == 22:
                    # nearest utility to chance square at position
23, is square 29
                    self.BoardConfig[28] = self.BoardConfig[28] + 1
                    self.CurrentPosition = 28

                if self.CurrentPosition == 36:
                    # nearest utility to chance square at position
37, is square 29
                    self.BoardConfig[28] = self.BoardConfig[28] + 1
                    self.CurrentPosition = 28
            if C.chancecard() == "Take a trip to Reading Railroad":
                self.BoardConfig[6] = self.BoardConfig[6] + 1
                self.CurrentPosition = 6
            if C.chancecard() == "Advance token to Boardwalk":
                self.BoardConfig[39] = self.BoardConfig[39] + 1
                self.CurrentPosition = 39
            if C.chancecard() == "Go to jail":
                self.BoardConfig[10] = self.BoardConfig[10] + 1
                self.CurrentPosition = 10
            if C.chancecard() == "get out of jail":
                self.CurrentPosition = self.CurrentPosition

        if (self.CurrentPosition == 30):
            self.CurrentPosition = 10
            self.BoardConfig[10] = self.BoardConfig[10] + 1

        return self.BoardConfig



    def get_board_ferequency(self):
        return self.BoardConfig


class Play():
     def __init__(self, moves):

         self.moves = moves

     def PlayMove(self):

         board = Board()
         num_moves = 0

         while (num_moves < self.moves):
             #print('time: ', num_moves)
             dice = Dice()
             [dice1, dice2] = dice.roll()
             pair = [dice1, dice2]
             #print('pairof dice: ', pair)
             if pair[0] != pair[1]:
                 #increment just when two dices are not the same, in
this case the double rolls will
                 num_moves = num_moves + 1
             board.WhereToGo(pair[0], pair[1])
             boardconfig = board.get_board_ferequency()
             #print('board: ', boardconfig)
         return boardconfig



if __name__ == '__main__':

    simulation = 1000
    moves = 100

    #Matrix is an array that has all the boardconfigs of all plays in
it.
    Matrix = []

    for i in range(simulation):
        play = Play(moves)
        config = play.PlayMove()
        Matrix.append(config)


    Probabilities = np.mean(Matrix, axis=0)
    print(Probabilities)
    print('Average of number of landing on GO square: ',
Probabilities[0])
    print('Average of number of landing on Mediterenean Avenue
square: ', Probabilities[1])
    print('Average of number of landing on Boardwalk square: ',
Probabilities[30])
    print('Average of number of landing on Pensilvania rail road
square: ', Probabilities[15])
    print('Average of number of landing on Reading rail square: ',
Probabilities[5])
    print('Average of number of landing on short line rail road
square: ', Probabilities[35])
    print('Average of number of landing on B&O rail road square: ',
Probabilities[25])
```

Question 2: Naive Bayes Text Classification

6. Write code that parses the text files and calculates the probabilities for each dictionary word given the review polarity:

Answer:

Approach:
I defined the class Classifier() and two separate functions *NegativeProbs* and *PositiveProbs* to calculate the probabilities regarding two classes. I used glob module to read the files. The general steps that I took to calculate these probabilities are as follows (steps for both methods are similar):

1. I take the file containing positive / negative reviews as input.
2. I open the files in each category (class) one by one and for each text file which is a movie review, I split the contents of the text file to separate words and store all the words in the list *contents*.
3. Then for each keyword in the vocabulary (["awful","bad","boring","dull","effective","enjoyable","great","hilarious"]) list that I defined it as the class variable, I check if the keyword is in the contents list or not (I check if the word exist in the review or not). So by keyword I mean each of the "awful","bad","boring",... . If the keyword exist in the *contents,* increment the *ncount (pcount)* counter by one.
    - *ncount:* is a counter to count the number of negative files that contains the keyword
    - *pcount:* is a counter to count the number of positive files that contains the keyword

4. Then to calculate the probability of P(keyword | class), I just divided the *ncount (pcount)* of the keyword by the total number of negative (positive) class.

Result:
The probability that I got for each of the words from the *vocabulary* list ["awful","bad","boring","dull","effective","enjoyable","great","hilarious"] given one of the two classes (positive and negative) is as follows:
negative class: [0.101, 0.505, 0.169, 0.091, 0.046, 0.053, 0.286, 0.05]
positive class: [0.019, 0.255, 0.048, 0.023, 0.12, 0.095, 0.408, 0.125]
So as you can see, the probability P(bad | negative) and P(great | positive) are the highest probabilities in each class that seems reasonable.

Code:

```
def NegativeProbs(self, path):
        # NegProbs is an array to store the probabilty of keywords given
the class
        NegProbs = []

        for keyword in self.vocabulary:
            #ncount is a counter to count the number of negative files
thatcontains the keyword
            ncount = 0
            self.path = path
            files = glob.glob(path)
            for name in files:
                with open(name) as file:
                    #content is the list of all words in each text file
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        ncount = ncount + 1


            NegProbs.append(float(ncount/1000))
```

```
def PositiveProbs(self, Path):


        PosProbs = []

        for keyword in self.vocabulary:
            pcount = 0
            self.Path = Path
            files = glob.glob(Path)

            for name in files:
                with open(name) as file:

                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        pcount = pcount + 1

            PosDict.update({keyword : float(pcount/1000)})
            PosProbs.append(float(pcount/1000))
```

2. Explain how these probability estimates can be combined to form a Naive Bayes classifier. You can look up Bernoulli Bayes model for this simple model where only presence/absence of a word is modeled.

Answer:

In naive bayes classifier we are interested in the probability P(Class | keywords) or in other words, P(Class | Awful, Bad, Boring, Dull, Effective, Enjoyable, Great, Hilarious). By considering a Bernoulli Bayes model, For each keyword there are two possibilities: being in a particular review (presence) and not being in a particular review (absence).
Generally, By bayes rule and naive bayes assumption (keywords are conditionally independent) we have:

$$\text{P(Class | Awful, Bad, Boring, Dull, Effective, Enjoyable, Great, Hilarious)} \propto$$

$$\text{P(Class)} \times \prod_{keyword} \text{P(keyword | Class)}$$

The corresponding classifier, a Bayes classifier, is the function that assigns a class label $\hat{y}$ for some keywords as follows:

$$\hat{y} = \text{argmax } \text{P(Class)} \times \prod_{keyword} \text{P(keyword | Class)}$$

In Bernoulli Bayes model, we consider both presence and absence of a keyword in the text files. In other words:

$$\text{P(Keywords | Class)} =$$

$$\prod_{keywords} \text{P(presence of the keyword)(1-P(presence of the keyword))}$$

3. Calculate the classification accuracy and confusion matrix that you would obtain using the whole data set for both training and testing.

Answer:
I created a feature matrix for each negative and positive classes of size 1000X9 where each row of these matrices represents the feature vector of the corresponding movie. Each entry (except the last row)of these matrices is *1* if the corresponding keyword exists in the contents of the file, otherwise it's *0*. The last row of these matrices representing the labels of the file. The last element of each row is *-1* if it belongs to the negative class and is *1* if it belongs to positive class. Then, I combine these two matrices together and build the *Featurematrix* that stores the feature vectors and labels for all 2000 files.

Code:

```
def FeatureNeg(self, path):

        self.path = path
        files = glob.glob(path)
        self.matrixNeg = np.zeros((1000, 9))
        for row in range(self.matrixNeg.shape[0]):
            #last row of the feature matrix for negative files is the label
-1 representing negative class
            self.matrixNeg[row][8] = -1

        for keyword in self.vocabulary:
         for f in files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        i = int(files.index(f))
                        j = int(self.vocabulary.index(keyword))
                        #checks if the keyword exists in the file, if yes
it changes the corresponding entry to 1
                        self.matrixNeg[i][j] = 1

        return self.matrixNeg


    def FeaturePos(self, Path):

        self.Path = Path
        Files = glob.glob(Path)
        self.matrixPos = np.zeros((1000, 9))
        for r in range(self.matrixPos.shape[0]):
            #last row of the feature matrix for positive files is the label
1 representing positive class
            self.matrixPos[r][8] = 1

        for Keyword in self.vocabulary:
         for f in Files:
                with open(f) as file:
                    self.Contents = file.read().strip().split()
                    if Keyword in self.Contents:
                        i = int(Files.index(f))
                        j = int(self.vocabulary.index(Keyword))
                        self.matrixNeg[i][j] = 1

        return self.matrixPos

    def Featurematrix(self, M, N):
        self.Featurematrix = np.concatenate((M, N))
        return self.Featurematrix
```

To calculate the accuracy and confusion matrix by using the whole data for both training and test set, I spilited my data into 70% and 30% parts corresponding to train set and test set respectively. Then, I calculated the probability of each keyword given the class on training set and stored the individual probabilities in arrays *NegProbs* and *PosProbs.* In other words, I built my model on my training set, and the I tested it on the test set. The NegativeProbs and PositiveProbs return four TP, FN, TN and FP values:

TP: predicted positive truly
FN: predicted negative wrongly
TN: predicted negative truly
FP: predicted positive wrongly

Then I use these values to calculate the accuracy which is:

Accuracy = (TP+TN) / (TP+TN+FP+FN)

Then the confusion matrix is 2 by 2 matrix in which the entries are these 4 values.

Code:

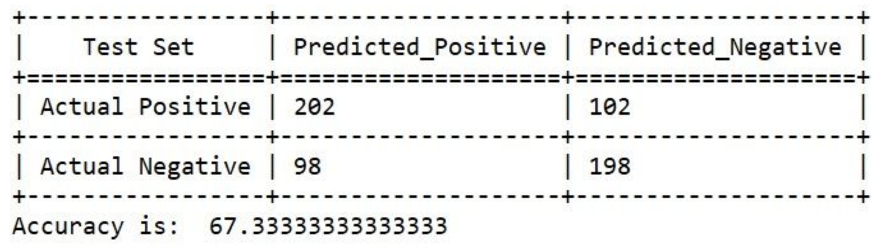
```
def NegativeProbs(self, path):

        NegProbs = []



        for keyword in self.vocabulary:
            #ncount is a counter to count the number of negative files that
contains the keyword
            ncount = 0
            self.path = path
            files = glob.glob(path)

            #Split data into tarin/test set (%70 / %30)
            self.ratio = 0.7
            split_index = floor(len(files) * self.ratio)
            self.training = files[:split_index]
            self.testing = files[split_index:]
            for name in self.training:
                with open(name) as file:
                    #content is the list of all words in each text file
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        ncount = ncount + 1



            NegProbs.append(float(ncount/1000))
            #print('NegProbs: ', NegProbs)

            TN = 0
            FP = 0

        for name in self.testing:

            with open(name) as file:

                self.testList = file.read().strip().split()
                self.NegativeProductProb = 1
                for word in self.vocabulary:
                        if word in self.testList:
                        # Calculating the probability of belong to negative
class given the keywords by naive bayes formula
                            self.NegativeProductProb =
self.NegativeProductProb * NegProbs[self.vocabulary == word]
                        if word not in self.testList:
                            self.NegativeProductProb =
self.NegativeProductProb * (1-NegProbs[self.vocabulary == word])
                if self.NegativeProductProb > 0.5:
                    TN = TN + 1
                else:
                    FP = FP + 1
        #self.Accuracy = float(TN / len(self.testing))*100


        return  TN, FP
```

```
def PositiveProbs(self, Path):

        PosDict = {}
        PosProbs = []

        for keyword in self.vocabulary:
            pcount = 0
            self.Path = Path
            files = glob.glob(Path)
            #shuffle the data
            shuffle(files)
            #Split data into tarin/test set (%70 / %30)
            self.ratio = 0.7
            split_index = floor(len(files) * self.ratio)
            self.training = files[:split_index]
            self.testing = files[split_index:]


            for name in files:
                with open(name) as file:

                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        pcount = pcount + 1

            PosDict.update({keyword : float(pcount/1000)})
            PosProbs.append(float(pcount/1000))
            print('PosProbs: ', PosProbs)

            TP = 0
            FN = 0
        #prediction
        for name in self.testing:

            with open(name) as file:

                self.testList = file.read().strip().split()
                self.PositiveProductProb = 1
                for word in self.vocabulary:
                        if word in self.testList:
                        # Calculating the probability of belong to negative
class given the keywords by naive bayes formula
```

```
                            print('PosProbs[self.vocabulary == word]: ',
PosProbs[self.vocabulary == word])
                            self.PositiveProductProb =
self.PositiveProductProb * PosProbs[self.vocabulary == word]
                            print('self.PositiveProductProb: ',
self.PositiveProductProb)

                        if word not in self.testList:
                            #print('(1-PosProbs[self.vocabulary == word])'
,(1-PosProbs[self.vocabulary == word]))
                            self.PositiveProductProb =
self.PositiveProductProb * (1-PosProbs[self.vocabulary == word])
                            #print('self.PositiveProductProb',
self.PositiveProductProb)
                #print('final self.PositiveProductProb: ',
self.PositiveProductProb)
                if self.PositiveProductProb > 0.5:
                    TP = TP + 1
                else:
                    FN = FN + 1
        #self.Accuracy = float(TN / len(self.testing))*100

        return  TP, FN
```

```
NegPredict =
list(myclassifier.NegativeProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_s
entoken\neg\*.txt'))
    PosPredict =
list(myclassifier.PositiveProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_s
entoken\pos\*.txt'))



    tab = tt.Texttable()
    headings = ['Test Set','Predicted_Positive','Predicted_Negative']
    tab.header(headings)
    names = ['Actual Positive', 'Actual Negative']
    Predicted_Positive = [PosPredict[0], NegPredict[0]]#[TP,FP]
    print('Predicted_Positive [TP,FP]', Predicted_Positive)
    Predicted_Negative = [PosPredict[1], NegPredict[1]]#[FP,FN]
    print('Predicted_Negative [FP,FN]', Predicted_Negative)
    Accuracy = (Predicted_Positive[0] + Predicted_Negative[0]) /
(Predicted_Positive[0] + Predicted_Negative[0] + Predicted_Positive[1] +
Predicted_Negative[1])*100


    for row in zip(names,Predicted_Positive,Predicted_Negative):
        tab.add_row(row)

    s = tab.draw()
    print (s)
    print('Accuracy is: ', Accuracy)
```

Result: I took a screenshot of my result (Accuracy and confusion matrix)



```
+-----------------+--------------------+--------------------+
|    Test Set     | Predicted_Positive | Predicted_Negative |
+=================+====================+====================+
| Actual Positive | 202                | 102                |
+-----------------+--------------------+--------------------+
| Actual Negative | 98                 | 198                |
+-----------------+--------------------+--------------------+
Accuracy is:  67.33333333333333
```

Cross Validation:

I used the cross validation of scikit learn to calculate the accuracy among the folds.

```
def FeatureNeg(self, path):

        self.path = path
        files = glob.glob(path)
        self.matrixNeg = np.zeros((1000, 9))
        for row in range(self.matrixNeg.shape[0]):
            #last row of the feature matrix for negative files is the label
-1 representing negative class
            self.matrixNeg[row][8] = -1

        for keyword in self.vocabulary:
         for f in files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        i = int(files.index(f))
                        j = int(self.vocabulary.index(keyword))
                        #checks if the keyword exists in the file, if yes
it changes the corresponding entry to 1
                        self.matrixNeg[i][j] = 1

        return self.matrixNeg
```

```
 def FeaturePos(self, Path):

        self.Path = Path
        Files = glob.glob(Path)
        self.matrixPos = np.zeros((1000, 9))
        for r in range(self.matrixPos.shape[0]):
            #last row of the feature matrix for positive files is the label
1 representing positive class
            self.matrixPos[r][8] = 1

        for Keyword in self.vocabulary:
         for f in Files:
                with open(f) as file:
                    self.Contents = file.read().strip().split()
                    if Keyword in self.Contents:
                        i = int(Files.index(f))
                        j = int(self.vocabulary.index(Keyword))
                        self.matrixNeg[i][j] = 1

        return self.matrixPos

    def Featurematrix(self, M, N):
        self.Featurematrix = np.concatenate((M, N))
        return self.Featurematrix

    #returns accuracy score
    def support_vector(X_train, y_train, X_test, y_test):
     clf_svm = svm.SVC(kernel="linear")
     clf_svm.fit(X_train, y_train) #train
     y_predict = clf_svm.predict(X_test) #predict
     return(accuracy_score(y_test, y_predict))
if __name__ == '__main__':
    myclassifier = Classifier()
    for i in range(5):
        print('random negative review', myclassifier.NegRandomReview())

   for j in range(5):
       print('random positive review', myclassifier.PosRandomReview())


    FeatureNegative =
myclassifier.FeatureNeg(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
neg\*.txt')
    FeaturePositive =
myclassifier.FeaturePos(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
pos\*.txt')
    M = myclassifier.Featurematrix(FeatureNegative, FeaturePositive)
#binary feature matrix where the last row is the labels so we
    labels = M[:,8]
    data = np.delete(M, -1 , axis=1)
```

```


    svm_acc_lst = []
    fold = 1

    RANDOM_SEED = 1
    k_fold = StratifiedKFold(n_splits=10, shuffle=True,
random_state=RANDOM_SEED)
    for train_idx, test_idx in k_fold.split(data, labels):
            X_train, X_test = data[train_idx], data[test_idx]
            y_train, y_test = labels[train_idx], labels[test_idx]
            svm_acc_lst.append(myclassifier.support_vector(X_train,
y_train, X_test, y_test)) # accuracies of all folds
            fold = fold + 1

    print(svm_acc_lst)
```

Output:

```
[0.67499999999999998, 0.65999999999999997, 0.68999999999999996,
0.71499999999999996, 0.69499999999999996, 0.73999999999999997,
0.66999999999999997, 0.68999999999999996, 0.70999999999999997,
0.65999999999999996]
```

Random reviews:
The idea is that, to generate for example random positive review, for each keyword in the defined list, we generate a random number between 0, 1. If this generated number is greater than the corresponding probability of the keyword given positive class ,then include that keyword in the review.

```
 def NegRandomReview(self):

        NegativeProbabilities =
myclassifier.NegativeProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\neg\*.txt')[0]

        self.review = []
        for w in range(len(self.vocabulary)):
             r = (np.random.rand(1))

             if NegativeProbabilities[w] < r:

                 self.review.append(self.vocabulary[w])
        return self.review



    def PosRandomReview(self):


        PositveProbabilities =
myclassifier.PositiveProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\pos\*.txt')[0]
        self.review = []
        for w in range(len(self.vocabulary)):
             r = (np.random.rand(1))
             print('pos r: ', r)
```

```
            if PositveProbabilities[w] < r:

                self.review.append(self.vocabulary[w])
        return self.review
```

My random generated reviews:

```
random negative review ['awful', 'enjoyable', 'hilarious']
random negative review ['dull', 'bad', 'boring', 'hilarious']
random negative review ['awful', 'enjoyable']
random negative review ['awful', 'bad', 'enjoyable']
random negative review ['effective', 'enjoyable', 'hilarious']

random positive review ['awful', 'enjoyable', 'dull']
random positive review ['awful', 'effective', 'great']
random positive review ['great', 'dull']
random positive review []
random positive review ['effective', 'boring']
```

All codes together:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jun  4 15:16:01 2018

@author: mahsa
"""
import os
import numpy as np
from os import listdir
from os.path import isfile, join
import glob
import errno
from random import shuffle
from math import floor
import texttable as tt
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn import svm
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score




class Classifier():


    def __init__(self):

        #vacaulary containing the words that we are interested to calculate
their accurance or absence probabilities
        self.vocabulary =
["awful","bad","boring","dull","effective","enjoyable","great","hilarious"]






    def NegativeProbs(self, path):

        NegProbs = []




        for keyword in self.vocabulary:
            #ncount is a counter to count the number of negative files that
contains the keyword
            ncount = 0
            self.path = path
            files = glob.glob(path)

            #shuffle the data
            #shuffle(files)
            #Split data into tarin/test set (%70 / %30)
            self.ratio = 0.7
            split_index = floor(len(files) * self.ratio)
            self.training = files[:split_index]
            self.testing = files[split_index:]



            for name in self.training:
                with open(name) as file:
                    #content is the list of all words in each text file
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        ncount = ncount + 1



            NegProbs.append(float(ncount/1000))
            #print('NegProbs: ', NegProbs)

            TN = 0
            FP = 0

        for name in self.testing:

            with open(name) as file:

                self.testList = file.read().strip().split()
                self.NegativeProductProb = 1
                for word in self.vocabulary:
                        if word in self.testList:
                        # Calculating the probability of belong to negative
class given the keywords by naive bayes formula
                            self.NegativeProductProb =
self.NegativeProductProb * NegProbs[self.vocabulary == word]
                        if word not in self.testList:
                            self.NegativeProductProb =
self.NegativeProductProb * (1-NegProbs[self.vocabulary == word])
                if self.NegativeProductProb > 0.5:
                    TN = TN + 1
                else:
                    FP = FP + 1
        #self.Accuracy = float(TN / len(self.testing))*100
        return  NegProbs, TN, FP




    def PositiveProbs(self, Path):

        PosDict = {}
        PosProbs = []

        for keyword in self.vocabulary:
            pcount = 0
            self.Path = Path
            files = glob.glob(Path)
            #shuffle the data
            shuffle(files)
            #Split data into tarin/test set (%70 / %30)
            self.ratio = 0.7
            split_index = floor(len(files) * self.ratio)
            self.training = files[:split_index]
            self.testing = files[split_index:]


            for name in files:
                with open(name) as file:

                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        pcount = pcount + 1

            PosDict.update({keyword : float(pcount/1000)})
            PosProbs.append(float(pcount/1000))
            #print('PosProbs: ', PosProbs)

            TP = 0
            FN = 0
        #prediction
        for name in self.testing:

            with open(name) as file:

                self.testList = file.read().strip().split()
                self.PositiveProductProb = 1
                for word in self.vocabulary:
                        if word in self.testList:
                        # Calculating the probability of belong to negative
class given the keywords by naive bayes formula
                            #print('PosProbs[self.vocabulary == word]: ',
PosProbs[self.vocabulary == word])
                            self.PositiveProductProb =
self.PositiveProductProb * PosProbs[self.vocabulary == word]
                            #print('self.PositiveProductProb: ',
self.PositiveProductProb)

                        if word not in self.testList:
                            #print('(1-PosProbs[self.vocabulary == word])'
,(1-PosProbs[self.vocabulary == word]))
                            self.PositiveProductProb =
self.PositiveProductProb * (1-PosProbs[self.vocabulary == word])
                            #print('self.PositiveProductProb',
self.PositiveProductProb)
                #print('final self.PositiveProductProb: ',
self.PositiveProductProb)
                if self.PositiveProductProb > 0.5:
                    TP = TP + 1
                else:
                    FN = FN + 1
        #self.Accuracy = float(TN / len(self.testing))*100

        return  PosProbs, TP, FN



    def FeatureNeg(self, path):

        self.path = path
        files = glob.glob(path)
        self.matrixNeg = np.zeros((1000, 9))
        for row in range(self.matrixNeg.shape[0]):
            #last row of the feature matrix for negative files is the label
-1 representing negative class
            self.matrixNeg[row][8] = -1

        for keyword in self.vocabulary:
         for f in files:
                with open(f) as file:
                    self.contents = file.read().strip().split()
                    if keyword in self.contents:
                        i = int(files.index(f))
                        j = int(self.vocabulary.index(keyword))
                        #checks if the keyword exists in the file, if yes
it changes the corresponding entry to 1
                        self.matrixNeg[i][j] = 1

        return self.matrixNeg


    def FeaturePos(self, Path):

        self.Path = Path
        Files = glob.glob(Path)
        self.matrixPos = np.zeros((1000, 9))
        for r in range(self.matrixPos.shape[0]):
            #last row of the feature matrix for positive files is the label
1 representing positive class
            self.matrixPos[r][8] = 1

        for Keyword in self.vocabulary:
         for f in Files:
                with open(f) as file:
                    self.Contents = file.read().strip().split()
                    if Keyword in self.Contents:
                        i = int(Files.index(f))
                        j = int(self.vocabulary.index(Keyword))
                        self.matrixNeg[i][j] = 1

        return self.matrixPos

    def Featurematrix(self, M, N):
        self.Featurematrix = np.concatenate((M, N))
        return self.Featurematrix
```

```
#     def crossvalidation(self,data_files):
#         self.fold = 10
#
#         self.fold_size = len(self.data_files) / self.fold
#         for split_index in range(0, len(self.data_files), self.fold_size):
#             self.training =
self.data_files[self.split_index:self.split_index + self.fold_size]
#             self.testing = self.data_files[:self.split_index] +
data_files[self.split_index + self.fold_size:]
#             yield self.training, self.testing


    def NegRandomReview(self):

        NegativeProbabilities =
myclassifier.NegativeProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\neg\*.txt')[0]

        self.review = []
        for w in range(len(self.vocabulary)):
             r = (np.random.rand(1))*0.1

             if NegativeProbabilities[w] < r:

                 self.review.append(self.vocabulary[w])
        return self.review



    def PosRandomReview(self):


        PositveProbabilities =
myclassifier.PositiveProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\pos\*.txt')[0]
        self.review = []
        for w in range(len(self.vocabulary)):
             r = (np.random.rand(1))*0.1
             #print('pos r: ', r)

             if PositveProbabilities[w] < r:
                  self.review.append(self.vocabulary[w])
        return self.review







if __name__ == '__main__':
    myclassifier = Classifier()
#    for i in range(5):
#        print('random negative review', myclassifier.NegRandomReview())
#
#    for j in range(5):
#        print('random positive review', myclassifier.PosRandomReview())


    FeatureNegative =
myclassifier.FeatureNeg(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
neg\*.txt')
    FeaturePositive =
myclassifier.FeaturePos(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
pos\*.txt')
    M = myclassifier.Featurematrix(FeatureNegative, FeaturePositive)
    labels = M[:,8]
    data = np.delete(M, -1 , axis=1)
    kf = KFold(n_splits=2)
    for train, test in kf.split(data):
        scores = cross_val_score(kf, data, labels, cv=10)
        print('score: ',scores)




#    print('Feature matrix of negative files: ',
myclassifier.FeatureNeg(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
neg\*.txt'))
#    M =
myclassifier.FeatureNeg(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
neg\*.txt')
#    N =
myclassifier.FeaturePos(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
pos\*.txt')
#    print('Feature matrix of positive files: ',
myclassifier.FeaturePos(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentoken\
pos\*.txt'))
#    print('total: ', myclassifier.Featurematrix(M, N))


#    print('Neg results: ',
myclassifier.NegativeProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\neg\*.txt')[0])
#
#    print('Pos results: ',
myclassifier.PositiveProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\pos\*.txt')[0])
#
#    NegativeProbabilities =
myclassifier.NegativeProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\neg\*.txt')[0]
#    PositveProbabilities =
myclassifier.PositiveProbs(r'C:\Users\mahsa\OneDrive\Desktop\AI2\txt_sentok
en\pos\*.txt')[0]
#
```