Data Interview Problem V2 solution by Norma Dani Risdiandita

1. Algorithmic Problem Solving
   Given a set of N integers
   $A = [a_1, a_2, ..., a_N]$
   where a_i is an integer between A_{low} and A_{high}.
   We try to convert A to $B = [b_1, b_2, ..., b_N]$ in which $max(B) - min(B) \leq maxdiff$ but with the minimum $cost = \sum_{i=1}^{N} (b-a)^2$.

   a) I made a Naive solution as a function solutionNaive(input). write input = "input.txt" and "input_extended.txt" to give a result.
   the input is below:
   input.txt :

```
4 10
1
5
10
15
```

   input_extended.txt :

```
7 10
1
5
10
13
25
35
445
```

   the code is below:

```
def solutionNaive(input):
   """
   find the minimum cost function with textfile input
   consists of size_of_A, MAX_DIFF, list of A

   input: string of textfile name (example: 'input.txt')
   format:
   =======
   size_of_A MAX_DIFF
   A_1
   A_2
   A_3
```

```
...
=======
example:
=======
4 10
1
5
10
13
=======
output: minimum cost function
"""

f = open(input)
i = 0
A = []
A_low = float("inf")
A_high = 0
for val in f:
    if i == 0:
        # assigning the size_of_A and the MAX_DIFF
        read_1 = val.split(" ")
        size_of_A, MAX_DIFF = int(read_1[0]), int(read_1[1])
        i = 1
    else:
        # Adding the row in text file into array
        A.append(int(val.strip("\n")))
        A_low = min(A_low,int(val))
        A_high = max(A_high,int(val))


# define a big number for initializing the cost function
cost = float("inf")

# minB runs from A_low to A_high - 1
for minB in range(A_low, A_high):
    # maxB runs from minB + 1 to the upper bound
    # of A_high
    for maxB in range(minB + 1, min(A_high, minB + MAX_DIFF) + 1):
        #calculating cost function for given minB and maxB
        cost_temp = 0
        B = A.copy()

        # assigning the
        for A_i in range(0, len(A)):
            if A[A_i] < minB:
                B[A_i] = minB
            elif A[A_i] > maxB:
                B[A_i] = maxB
            cost_temp += (A[A_i] - B[A_i])*(A[A_i] - B[A_i])
```

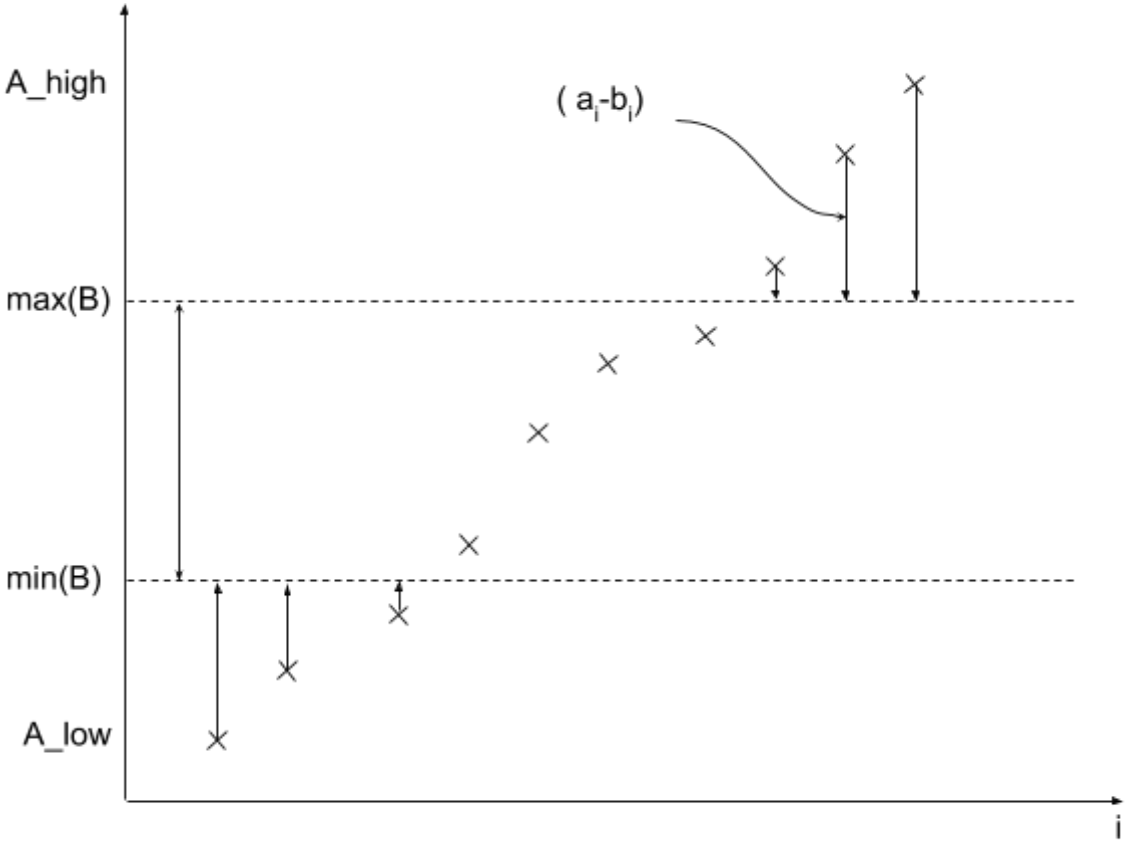
```
            cost = min(cost, cost_temp)
    return cost

print("the minimum cost of the 'input.txt' is
{}".format(solutionNaive("input.txt")))
print("the minimum cost of the 'input_extended.txt' is
{}".format(solutionNaive("input_extended.txt")))
```

the output is

```
the minimum cost of the 'input.txt' is 8
the minimum cost of the 'input_extended.txt' is 152145
```

b) To solve this problem we need see the depiction of the array A and B



From the figure above, we can see that:

- To minimize the cost function, we need to choose the value max(B) - min(B) into maximum, therefore, we already know that max(B) - min(B) = MAX_DIFF.
- Sorting directly after reading the input text-file.
- Most likely, the minimum value is around the **average (centre of mass)** of A, we can scan the cost function from that point **up** and **down**.
- We need to stop going **up** if the cost value increase and stop going **down** if the cost value increase.

- We would add the cost by the component changed (written as B_i) instead of iterating through All A and B components and summing up.

The python code “solutionNotSoNaive(input = “input.txt”)” saved as “newsol.py” is given by

```
def mergeSort(A):
    def merge(left, right): # [1,2,9,8] and [3,6,5]
        result = []
        i, j = 0, 0
        while i< len(left) and j < len(right):
            if left[i] < right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result += left[i:]
        result += right[j:]
        return result
    if len(A) <= 1:
        return A
    mid = len(A) // 2
    lefthalf = mergeSort(A[:mid])
    righthalf = mergeSort(A[mid:])
    return merge(lefthalf, righthalf)
def solutionNotSoNaive(input):
    """
    find the minimum cost function with textfile input
    consists of size_of_A, MAX_DIFF, list of A

    input: string of textfile name (example: 'input.txt')
    format:
    =======
    size_of_A MAX_DIFF
    A_1
    A_2
    A_3
    ...
    =======
    example:
    =======
    4 10
    1
    5
    10
    13
    =======
    output: minimum cost function
    """
```

```
f = open(input)
i = 0
A = []
A_low = float("inf")
A_high = 0
for val in f:
    if i == 0:
        # assigning the size_of_A and the MAX_DIFF
        read_1 = val.split(" ")
        size_of_A, MAX_DIFF = int(read_1[0]), int(read_1[1])
        i = 1
    else:
        # Adding the
        A.append(int(val))
        A_low = min(A_low,int(val))
        A_high = max(A_high,int(val))

# sorting A using mergesort
A = mergeSort(A)
# initialize B
B = A.copy()

# the middle between minB and maxB is the average of A

minB = sum(A) // len(A)
maxB = minB + MAX_DIFF

# first scan at the middle to determine the difference
# in the cost function

cost = 0
for A_index in range(len(A)):
    if A[A_index] < minB:
        B[A_index] = minB
        cost += (A[A_index] - B[A_index])*(A[A_index] - B[A_index])
    elif A[A_index] > maxB:
        B[A_index] = maxB
        cost += (A[A_index] - B[A_index])*(A[A_index] - B[A_index])

# scanning by going down
# initialize newcost as a temporary variable
newcost = 0
# initializing B
B = A.copy()
# initializing new minB and maxB as temporary variables
minBnew = minB
maxBnew = maxB
while True:
    minBnew -= 1
    maxBnew -= 1
```

```
        for A_index in range(len(A)):
            if A[A_index] < minBnew:
                B[A_index] = minBnew
                newcost += (A[A_index] - B[A_index])*(A[A_index] -
B[A_index])
            elif A[A_index] > maxBnew:
                B[A_index] = maxBnew
                newcost += (A[A_index] - B[A_index])*(A[A_index] -
B[A_index])
        # updating the cost, if the newcost is smaller
        if newcost < cost:
            cost = min(cost, newcost)
            newcost = 0
        elif newcost > cost:
            break
        elif minBnew == A_low:
            break

    # scanning by going up
    # initialize newcost as a temporary variable
    newcost = 0
    # initializing B
    B = A.copy()
    # initializing new minB and maxB as temporary variables
    minBnew = minB
    maxBnew = maxB
    while True:
        minBnew += 1
        maxBnew += 1
        for A_index in range(len(A)):
            if A[A_index] < minBnew:
                B[A_index] = minBnew
                newcost += (A[A_index] - B[A_index])*(A[A_index] -
B[A_index])
            elif A[A_index] > maxBnew:
                B[A_index] = maxBnew
                newcost += (A[A_index] - B[A_index])*(A[A_index] -
B[A_index])
        # updating the cost, if the newcost is smaller
        if newcost < cost:
            cost = min(cost, newcost)
            newcost = 0
        elif newcost > cost:
            break
        elif minBnew == A_low:
            break
    return cost

print("The minimum cost for the 'input.txt' is
{}".format(solutionNotSoNaive("input.txt")))
```

```
print("The minimum cost for the 'input_extended.txt' is
{}".format(solutionNotSoNaive("input_extended.txt")))
```

with the input of “input.txt” and “input_extended.txt” shown in part a, the output is

```
The minimum cost for the 'input.txt' is 8
The minimum cost for the 'input_extended.txt' is 152145
```

We get the same results with the part (a)

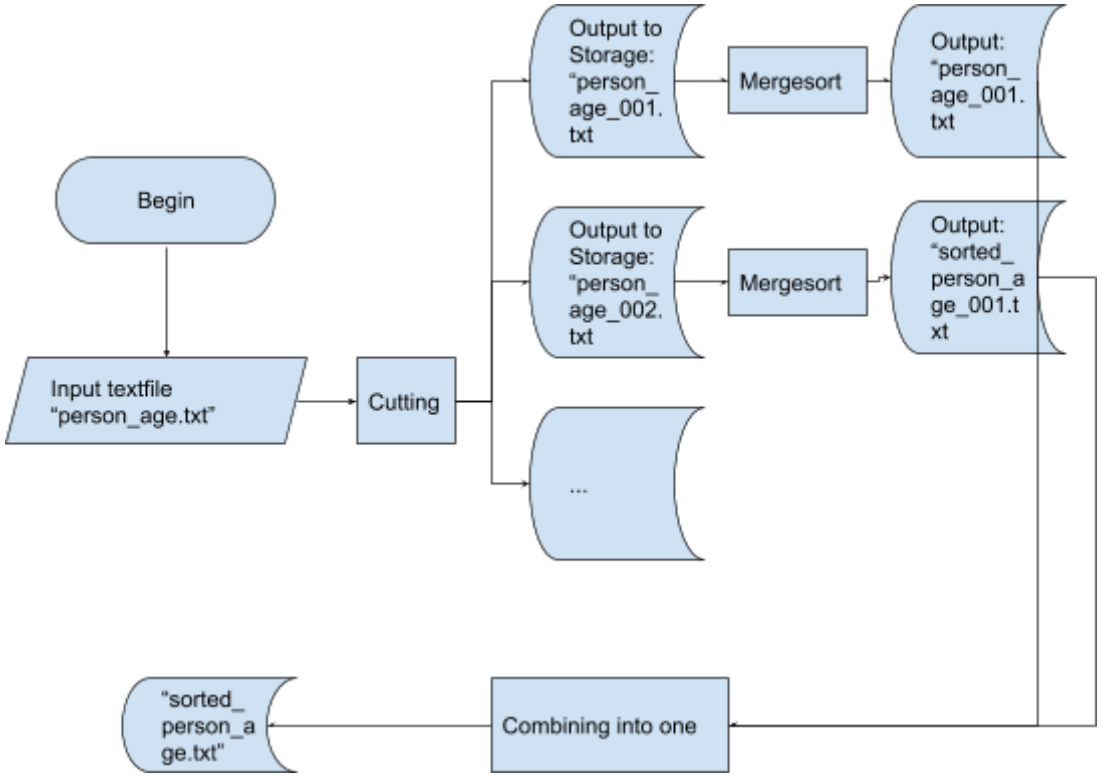
2. Big Data

In this problem we want to sort the input data consists of “Name” and “Age”

a) In this problem, I have made several assumptions:

- The oldest person ever alive recorded in history is **Jeanne Calment** (1875–1997) (source: Wikipedia). She died at the age of 122. Therefore, if we cut the input text-file according to the age, I can make 130 text-files for temporary files in the hard-disk instead of RAM.
- According to certain source (www.ecology.com/birth-death-rates/), approximately, 131.4 million babies are born each year, the names of 131.4 million people could fit in the RAM. But, I think it can be much lower than that number due to the deaths and the real age distribution.

By considering the assumptions above, I wrote a python code to cut the input text-file “person_age.txt” into smaller temporary files “person_age_{0:03}.txt”, sorted each temporary file using merge sort algorithm into “sorted_person_age_{0:03}.txt” files and combine the results into a single text-file (see diagram below)



The Python text-file is given by “person_age.py”

```
# initializing mergesort for sorting purpose
def mergeSort(A):
    def merge(left, right): # [1,2,9,8] and [3,6,5]
        result = []
        i, j = 0, 0
        while i< len(left) and j < len(right):
            if left[i].lower() < right[j].lower():
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result += left[i:]
        result += right[j:]
        return result
    if len(A) <= 1:
        return A
    mid = len(A) // 2
    lefthalf = mergeSort(A[:mid])
    righthalf = mergeSort(A[mid:])
    return merge(lefthalf, righthalf)

# initialize the input phone and age as a text
F = open("person_age.txt")

# creating temporary textfiles per age of the phone
for i in range(1,120+1):
    f = open("tempfile/person_age_{0:03}.txt".format(i), "a")
    f.close()

for row in F:
    row = row.split(" ")
    age = int(row[1].strip("\n"))
    name = row[0]
    G = open("tempfile/person_age_{0:03}.txt".format(age), "a")
    G.write(name + "\n")
G.close()

# sorting the names inside each textfile using mergesort
# and rewrite everything the sorted names into
# new textfiles

for i in range(1,120+1):
```

```
    with open("tempfile/person_age_{0:03}.txt".format(i)) as f:
        with
open("tempfile/sorted_person_age_{0:03}.txt".format(i), "w") as
g:
            for row in mergeSort(f.read().split("\n")[:-1]):
                g.write("{}\n".format(row))


# combining all of the sorted textfile by age and name into one
file
with open("sorted_phone_age.txt", "w") as finalsorted:
    for i in range(1, 120+1):
        with
open("tempfile/sorted_person_age_{0:03}.txt".format(i)) as f:
            names = f.read().split("\n")[:-1]
            for name in names:
                finalsorted.write("{} {}\n".format(name, i))


# delete all of temporary textfiles to save memory
import os
for i in range(1,120+1):
    os.remove("tempfile/person_age_{0:03}.txt".format(i))

os.remove("tempfile/sorted_person_age_{0:03}.txt".format(i))
```

by using an input text-file “person_age.txt” shown below

```
person_A 24
person_B 15
person_C 52
person_D 65
person_E 55
person_F 78
person_G 15
person_H 55
person_I 34
person_J 45
```

would get an output like below (“sorted_person_age.txt”)

```
person_B 15
person_G 15
person_A 24
person_I 34
```

```
person_J 45
person_C 52
person_E 55
person_H 55
person_D 65
person_F 78
```

b) The code of the API Server is given below. The function initialize(blacklist) will input a string of the filename “blacklist.txt" and gives an output of dictionary to initialize the function check_blacklist(name, phone_number). The file is saved as “blacklist.py”

```
def initialize(blacklist):
    """
    This function would convert a textfile consists of
    (name) and (phone_number) and give an output of python
    dictionary with phone number as the key and the
    name as the value. This is because the phone number is
    more likely to be unique in comparison to the name

    input: a string of textfile name (example: "blacklist.txt")
    output: {1341441: "Andi", 8565467:"Melisa", ...}
    """



    with open(blacklist) as f:
        blacklist = f.read()
    blacklist = blacklist.split()
    blacklist_dict = {}
    for i in range(0, len(blacklist), 2):
        blacklist_dict[int(blacklist[i+1])] = blacklist[i]
    return blacklist_dict

API_call = initialize("blacklist.txt")

def check_blacklist(name, phone_number):
    """
    This function would check whether the input (name,
phone_number)
    is within the blacklist or not

    input: name (a string), phone_number (an integer)
    output: True (if the name and phone_number combination is
            within the blacklist), False (the other way around)
    """
```

```
    return API_call.get(phone_number) == name


print("CHECKING THE RESULT")
print(check_blacklist("Andi", 1341441))
print(check_blacklist("Robert", 1311441))
print(check_blacklist("Takeshi", 8321441))
print(check_blacklist("Aslam", 2908345))
```

The script above will show the output

```
CHECKING THE RESULT
True
False
False
True
```

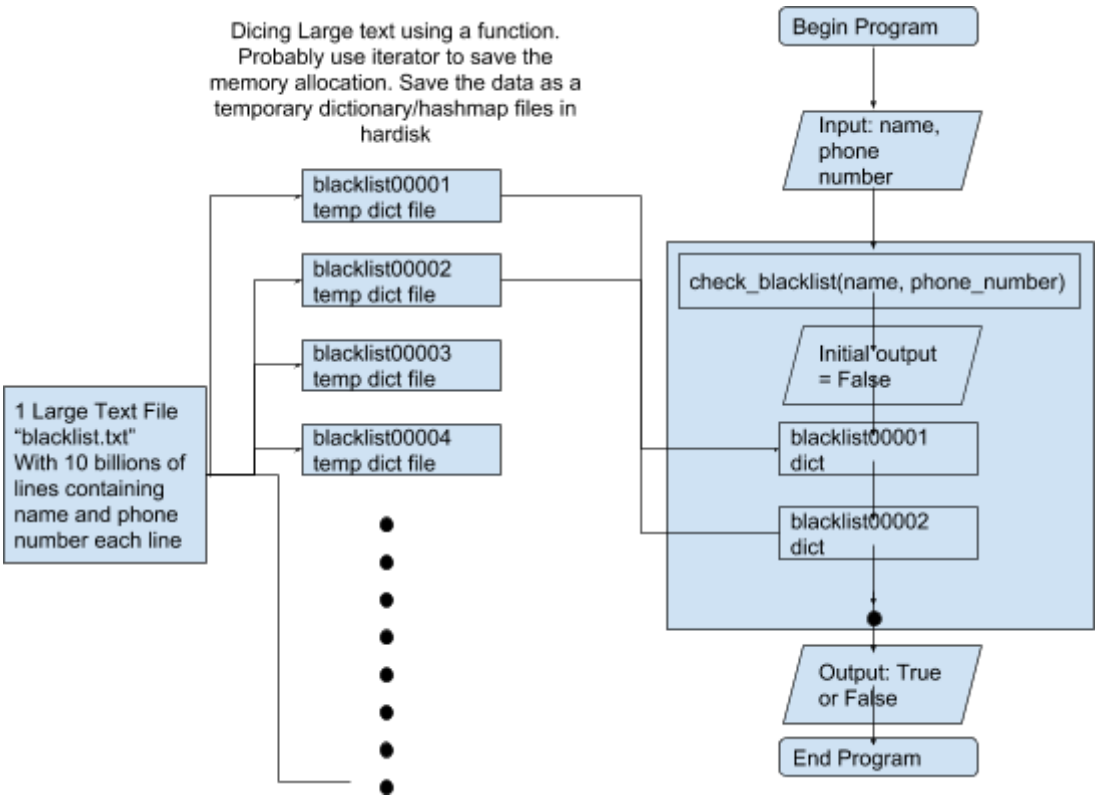
The sample input text file named “blacklist.txt” with three lines of input:

```
Andi 1341441
Melisa 8565467
Aslam 2908345
```

This kind of dictionary method will suffice to handle 1 million blacklisted “name”s and “phone_number”s in a dictionary without hurting our 1 GB of RAM. Initializing the data would have O(N) time and memory complexity and checking the “name” and “phone_number” input would take O(1) time complexity.

c) The problem of designing a system for handling 10 billions fake contacts:

- The system needs to have O(1) Time Complexity of checking the input “name” and “phone_number” to function properly since O(N) is practically impossible for 10 billions of fake accounts. If we use Python without external libraries, dictionary/hashing method would be the best to handle.
- If we use our previous straightforward “text file input to dictionary” method in part (b), our low performance RAM could not save those amounts of data as a single dictionary. O(N) memory complexity would be useless against those gigantic 10 billions lines of data. But I think, we can trick this problem by cutting the data into pieces and save them into hard disk as temporary files and tries to access the temporary files by converting each of them as dictionary temporarily to check the input “name” and “phone_number” to change the initial “False” into “True” if the input is within the blacklists.



The Design of the check_blacklist(name, phone_number) function is shown above

- First of all, We need to know the reasonable number of chopped "blacklist.txt" temporary files. I perform a simple experiment on my laptop to print one million lines of "Andi 1234567" string and save them into a file "checksize.txt" with the size of the file is 14 MB. Therefore, 10 billions of lines of "Andi 1234567" is around 140 GB of text-file. If my RAM is 1 GB, then by assuming 250 MB of free space of RAM is a reasonable option(might grows or shrinks due to the text-file to dictionary conversion). By dividing 140 GB / 0.25 GB = 560 temporary files, we can cut the "blacklist.txt" into "blacklist00001", "blacklist00002", ...,"blacklist00560" temporary python dictionary files and save them to our hard disk. The time complexity of the chopping process is O(N = 10 billions) and the memory allocation used is around 250 MB for each temporary file. This process occurs once to train our computer.
- Secondly, We input the a "name" and a "phone_number" into the check_blacklist(name, phone_number). This function would access "blacklist00001", "blacklist00002", ...,"blacklist00560" temporary python dictionary files. With similar method in part (b) but with each dictionary checking (time complexity of O(1)), the time complexity of the check_blacklist(name, phone_number) function is $560 \times O(1) = O(N = 560)$. With this system, I have designed a system to check each input file with O(N = 560) time complexity with 250 MB of RAM used.

3. Machine Learning
   a) We need to perform feature selection to reduce the features and choose only the most relevant features. Besides, according to https://towardsdatascience.com/feature-selection-techniques-in-machine-learning-with-python-f24e7da3f36e, there are three methods that can be used to perform features selection:

- Univariate Selection using Scikit-Learn's SelectKBest method to rank the importance recorded as feature scores.
  From this method, we can see the top important features with their respective importance scores.
- Feature Importance using tree based classifier (XGBoost and some packages also provide feature importance scorer).
- Making Correlation Heatmap. By showing the correlation matrix, we would know the importance of each feature in correlation to the dependent variable / target. By looking at a correlation matrix, we would know that there are correlations between features - target or features - features. By choosing the highest score correlation threshold, we can choose the highest features-target correlation. If there are two highly correlated two features to each other, we can choose only one.

Feature selection could increase the accuracy and the performance of a model. Moreover, it would make the computation process cheaper.

b) The cause of bad accuracy in the deployment of a model while the training model is considerably good is because of "overfitting". One of the common origins of "overfitting" is because we use highly flexible machine learning algorithm to process few number of of data samples. Here below the step-by-step method I will use to avoid "overfitting":

- **Performing validation of the data**:

  the most basic method of validation technique is by splitting the **original training dataset** into the **training dataset** and **validation dataset**. Two of the basic rule-of-thumbs is by splitting into **#training: #validation = 80:20** or **#training: #validation = 70:30**.
- **Applying machine learning algorithms to the training dataset and validating to the validation dataset**:

  After splitting the data, we can use various range of machine learning algorithms to be used on the training dataset and performing validation on the validation dataset and record the respective accuracy.

  What we need to know is the concept of "bias variance trade-off". If we use a less flexible machine learning algorithm, we will expect to suffer low accuracy to the both of the training dataset and validation dataset. This low accuracy model is called "underfitting". On the other hand, if we apply a too flexible machine learning model to low number of data samples, we would get high accuracy from our training model but getting relatively low accuracy in the validation step. This is called "overfitting".

- **Using a moderately flexible machine learning algorithm**.

  How flexible or not flexible machine learning algorithm is relative to the input data. So we need to have our own rule-of-thumb from performing training and validation test. Sometimes linear regression (regression case) or logistic regression (classification case) work well even though they have low flexibility if the data behave linearly. Otherwise, highly flexible data can be causing overfitting like "kernel" SVR (regression case), SVM (classification case), or "too deep" decision tree method. We can use the algorithms with

the flexibility is in between or tune the parameter of a given algorithm (example: by reducing or increasing the depth of decision tree)

- **Using K-Fold cross-validation instead of traditional splitting of training and validation dataset.**

  if we have a few data, K-Fold cross-validation is better. If we have huge amount of data, we can ignore cross-validation. If we have large amount of data but not sure about ignoring cross-validation, we can use traditional train-validation split method.

c) We can use bagging method by transforming the tweeter's statuses into words and the respective frequency of each word in a single status.

Using CountVectorizer from sklearn.feature_extraction.text.CountVectorizer, we can convert the words into their respective frequency.

Example: We have two sentences. "Wow it's cool and sophisticated" by a man and "it's beautiful and refreshing" by a woman. Using CountVectorizer, these two sentences could be transformed into a matrix (We also change the Gender into binary)

	Wow	it's	cool	and	sophisticated	beautiful	refreshing	Gender
1	1	1	1	1	1	0	0	1
2	0	1	0	1	0	1	1	0

After we add enough data (two above are just an example), we normalize the data using TF-IDF (term-frequency) because sometimes each word appears too often and the others don't (example of TF-IDF method: # count(word) / # total(word)).

Then, we can input the training data to the SVM algorithm.

Disclaimer: Removing **stop words** (and, the, ...) would be useful to reduce the computational complexity or even increase the accuracy. But, if we are not sure about it, we can test and compare.

Source:
https://towardsdatascience.com/machine-learning-nlp-text-classification-using-scikit-learn-python-and-nltk-c52b92a7c73a

4. Statistics

a) What is the expected number of die rolls required to get three same consecutive outcomes (for example: a 111, 222, etc) if we use a 6-sided fair die?

Answer:

Let us find the expected number of rolls required to get a 111 named as E_{111}. This expectation value is actually the weighted sum of the probable number of rolls to get 111 configuration written as

$$E_{111} = \sum_{i=1}^{N} p(X_i)X_i$$

Where X_i is the probable number of rolls. Let us show the possible number of rolls and the respective probabilities as a table

Configuration	Probability	Number of Rolls	$p(X_i)X_i$
111	$\frac{1}{6} \times \frac{1}{6} \times \frac{1}{6} = \frac{1}{216}$	3	$(1/216) \times 3$
1(not 1)	$\frac{1}{6} \times \frac{5}{6} \times 1 = \frac{5}{36}$	$2 + E_{111}$	$(5/36) \times (2 + E_{111})$
11(not 1)	$\frac{1}{6} \times \frac{1}{6} \times \frac{5}{6} \times 1 = \frac{5}{216}$	$3 + E_{111}$	$(5/216) \times (3 + E_{111})$
(not 1)	$\frac{5}{6} \times 1$	$1 + E_{111}$	$(5/6) \times (1 + E_{111})$

By applying the expectation value formulation above and inputting the values in given table

$E = (1/216) \times 3 + (5/36) \times (2 + E_{111}) + (5/216) \times (3 + E_{111}) + (5/6) \times (1 + E_{111})$

calculating the right hand side we would get

$E_{111} = \left(\frac{3}{216} + \frac{60}{216} + \frac{15}{216} + \frac{185}{216}\right) + \left(\frac{30}{216} + \frac{5}{216} + \frac{180}{216}\right) E_{111} = \frac{258}{216} + \frac{215}{216} E_{111}$

$\Rightarrow E_{111} = 258$

We know that for the case of $E_{222}, E_{333}, E_{444}, E_{555}, E_{666}$ which has 258 expected number of rolls each. Therefore the number of expected rolls to have consecutive three numbers is

$E_{111,222,333,444,555,666} = 258/6 = 43$

b) We throw 8 dice and take the sum of the highest 4 outcomes. What is the probability that the sum equals to 24?

answer:

To have a sum of 24 with 4 digits, we need to have at least 4 “sixes” in our outcomes. The probability of the sum equals to 24 is the same as the probability to have AT LEAST 4 “sixes” in the outcome. Therefore

P(at least 4 sixes | 8 rolls) = P(4 sixes | 8 rolls) + P(5 sixes | 8 rolls) + … + P(8 sixes | 8 rolls)

Summarizing the probabilities on a table

(Note: P(x sixes | 8 rolls) = Combination(8, x) × $(⅙)^x(⅚)^{(8-x)}$)

and using Python code to generate the result

```
# defining the factorial function
def factorial(x):
    result = 1
    for i in range(1,x+1):
        result = result*i
    return result

# defining the function of probability as a function of x
def Prob(x):
    combination = (factorial(8)/(factorial(x)*factorial(8-x)))
    events_per_total = ((1/6)**x)*((5/6)**(8-x))
    return combination*events_per_total

# printing for each values
for i in range(4,8+1):
    print("P(x = {}) = ".format(i), end = "")
    print(Prob(i))

# sum over all possible x = 4 to x = 8
Probtotal = 0
print("The total Probability is ", end = "")
for i in range(4,8+1):
    Probtotal += Prob(i)
print(Probtotal)
```

will print the result below

```
P(x = 4) = 0.026047620408474317
P(x = 5) = 0.004167619265355891
P(x = 6) = 0.00041676192653558895
P(x = 7) = 2.3814967230605084e-05
P(x = 8) = 5.95374180765127e-07
The total Probability is 0.030656411941777168
```

rewriting the results into a table

Probability of	Equation	Values
P(4 sixes | 8 rolls)	$\left(\frac{8!}{4!(8-4)!}\right) \times \left(\frac{1}{6}\right)^4 \times \left(\frac{5}{6}\right)^{(8-4)}$	0.026047620408474317
P(5 sixes | 8 rolls)	$\left(\frac{8!}{5!(8-5)!}\right) \times \left(\frac{1}{6}\right)^5 \times \left(\frac{5}{6}\right)^{(8-5)}$	0.004167619265355891
P(6 sixes | 8 rolls)	$\left(\frac{8!}{6!(8-6)!}\right) \times \left(\frac{1}{6}\right)^6 \times \left(\frac{5}{6}\right)^{(8-6)}$	0.00041676192653558895

P(7 sixes | 8 rolls)	$\left(\frac{8!}{7!(8-7)!}\right) \times \left(\frac{1}{6}\right)^7 \times \left(\frac{5}{6}\right)^{(8-7)}$	2.3814967230605084e-05
P(8 sixes | 8 rolls)	$\left(\frac{8!}{8!(8-8)!}\right) \times \left(\frac{1}{6}\right)^8 \times \left(\frac{5}{6}\right)^{(8-8)}$	5.95374180765127e-07
Total = P(at least 4 sixes | 8 rolls)	$\sum_{x=4}^{8} \left(\frac{8!}{x!(8-x)!}\right) \times \left(\frac{1}{6}\right)^x \times \left(\frac{5}{6}\right)^{(8-x)}$	0.030656411941777168

The result is 0.030656411941777168

c) We throw 6 dice at the same time. What is the expected number of distinct outcomes? Example: if the outcomes are 1, 2, 1, 3, 4, and 2, then there are 4 distinct outcomes (i.e. 1,2,3, and 4)

- For the possible distinct outcome of throwing n = 1 die:
  P(k = 1 | n = 1) = 1 (k is the number of distinct outcomes, and n is the number of throwing dice)
- For n = 2
  The probability of having 1 distinct outcome is
  P(k = 1 | n = 2) = 1/6,
  The probability of having 2 distinct outcome is
  P(k = 2 | n = 2) = 5/6.
- For n = 3
  The probability of having 1 distinct outcome is
  $P(k = 1 \mid n = 3) = (1/6)^2$
  The probability of having 2 distinct outcome is
  $P(k = 2 \mid n = 3) = (5 \times 3)/6^2$
  The probability of having 3 distinct outcome is
  $P(k = 3 \mid n = 3) = (5 \times 4)/6^2$

Because I am tired, I will summarize the values above into a recursion relation

P(k, n) = P(k | n - 1) × k / 6 + P(k - 1 | n - 1) × (7 - k) / 6

The formulation of the expectation value would be written as

$$E = \sum_{i=1}^{N} p(X_i)X_i$$

with Xi is the number of possible distinct outcome, by using Python as a calculator with formulation above we would get

```
def Prob(n, k):
    if k == 1 and n == 1:
        return 1
    elif k == 0:
        return 0
    elif k > n:
        return 0
```

```
    if n == 0:
        return 0
    else:
        return Prob(n-1,k)*k/6 + Prob(n-1,k-1)*(7-k)/6

expectation = 0
for i in range(1,6+1):
    expectation += i * Prob(6,i)

print("The expectation value is {}".format(expectation))
```

```
The expectation value is 3.9906121399176957
```

The result is E ≅ 3.99

d) We throw a die 10000 times and record the sum of the outcomes. Approximate the probability that the sum is in the range of [34500, 35500]. (The approximation must be within +/- 0.03%)

Answer:

I performed iterations using Python until I was getting tired of waiting. The probability is oscillating between 0.9965 and 0.9971 (written as 0.9968 (1±0.0003) while approaching 10000 sample cases using the code below. The value does make sense after checking that the extreme cases (sum = 10000 or sum = 60000) are highly improbable.

```
import random

sum_within_range = 0
sum_outside_range = 0
iterator = 0
probability = 1
while True:
    sum = 0
    for i in range(10000):
        sum += random.randint(1,6)
    if 34500 <= sum <= 35500:
        sum_within_range += 1
    else:
        sum_outside_range += 1
    probability = (sum_within_range / (sum_within_range +
sum_outside_range))
    iterator += 1
```

```
    if iterator % 50 == 0:
        print("probability = {}, iteration =
{}".format(probability, iterator))
    if iterator == 10000:
        break
```