

سوال اول:

قسمت الف: کالاهای 1, 2, 3, 4, ..., 20 کالاهای پرتکرار خواهند بود زیرا حداقل در پنج transaction می آیند. به دلیل اینکه هر کالا فقط در سبد هایی می آید که بتواند شماره ی آن سبد را عاد کند. به عنوان مثال کالای شماره ی 20 در سبد های

$$1 \times 20$$

$$2 \times 20$$

$$3 \times 20$$

$$4 \times 20$$

$$5 \times 20$$

می آید. اما از کالای 21 به بعد می دانیم در کمتر از 5 سبد می آید. (سبد: transaction)

support	itemsets
1	frozenset({1})
0.5	frozenset({2})
0.33	frozenset({3})
0.25	frozenset({4})
0.2	frozenset({5})
0.16	frozenset({6})
0.14	frozenset({7})
0.12	frozenset({8})
0.11	frozenset({9})
0.1	frozenset({10})
0.09	frozenset({11})
0.08	frozenset({12})
0.07	frozenset({13})
0.07	frozenset({14})
0.06	frozenset({15})
0.06	frozenset({16})
0.05	frozenset({17})
0.05	frozenset({18})
0.05	frozenset({19})
0.05	frozenset({20})

قسمت ب: برای زوج کالا های پر تکرار مشاهده ی ما این است که هر کدام از آن ها باید خودشان پرتکرار باشند تا زوج آن ها پر تکرار شود. همچنین باید در حداقل 5 سبد کنار هم ظاهر شده باشند. پس باید تعداد مضرب های مشترک کمتر از 100 آن ها بیشتر یا مساوی 5 باشد. پس پاسخ بصورت زیر خواهد بود:

$$\text{ans} = \{(a, b) \mid a, b \in \{1, \dots, 20\} \text{ and } \#\{\text{common multiples}(a, b)\} \geq 5\}$$

support	itemsets	support	itemsets	support	itemsets
0.5	frozenset({1, 2})	0.11	frozenset({9, 3})	0.06	frozenset({1, 15})
0.33	frozenset({1, 3})	0.11	frozenset({9, 1})	0.06	frozenset({8, 16})
0.16	frozenset({2, 3})	0.05	frozenset({9, 6})	0.06	frozenset({16, 4})
0.25	frozenset({2, 4})	0.05	frozenset({9, 2})	0.06	frozenset({16, 2})
0.25	frozenset({1, 4})	0.1	frozenset({10, 5})	0.06	frozenset({16, 1})
0.08	frozenset({3, 4})	0.1	frozenset({10, 2})	0.05	frozenset({17, 1})
0.2	frozenset({1, 5})	0.1	frozenset({1, 10})	0.05	frozenset({9, 18})
0.1	frozenset({2, 5})	0.05	frozenset({10, 4})	0.05	frozenset({18, 6})
0.06	frozenset({3, 5})	0.09	frozenset({1, 11})	0.05	frozenset({18, 3})
0.05	frozenset({4, 5})	0.08	frozenset({12, 6})	0.05	frozenset({18, 2})
0.16	frozenset({3, 6})	0.08	frozenset({4, 12})	0.05	frozenset({1, 18})
0.16	frozenset({2, 6})	0.08	frozenset({3, 12})	0.05	frozenset({1, 19})
0.16	frozenset({1, 6})	0.08	frozenset({2, 12})	0.05	frozenset({10, 20})
0.08	frozenset({4, 6})	0.08	frozenset({1, 12})	0.05	frozenset({20, 5})
0.14	frozenset({1, 7})	0.07	frozenset({1, 13})	0.05	frozenset({4, 20})
0.07	frozenset({2, 7})	0.07	frozenset({14, 7})	0.05	frozenset({2, 20})
0.12	frozenset({8, 4})	0.07	frozenset({2, 14})	0.05	frozenset({1, 20})
0.12	frozenset({8, 2})	0.07	frozenset({1, 14})		
0.12	frozenset({8, 1})	0.06	frozenset({5, 15})		
		0.06	frozenset({3, 15})		

قسمت ج: در واقع هر تراکنش به تعداد مقسوم علیه های خودش کالا دارد پس کل تعداد کالا های خریداری شده برابر است با

$$\text{ans} = \sum_{i \in \{1, 2, \dots, 100\}} \#\{\text{divisors of } i\} = 482$$

سوال دوم:

tid	itemset
$t_1$	ABCD
$t_2$	ACDF
$t_3$	ACDEG
$t_4$	ABDF
$t_5$	BCG
$t_6$	DFG
$t_7$	ABG
$t_8$	CDFG

با استفاده از الگوریتم Apriori و  $\text{minsup}=3$  داریم:

$\emptyset$						
A (5)	B (4)	C (5)	D (6)	E (1)	F (4)	G (5)
AB (3)	BC (2)	CD (4)	DF (4)		FG (2)	
AC (3)	BD (2)	CF (2)	DG (3)			
AD (4)	BF (1)	CG (3)				
AF (2)	BG (2)					
AG (2)						
ABC (1)		CDG (2)	DFG (2)			
ABD (2)						
ACD (3)						

در هر مرحله با استفاده از آپراتور  $\oplus$  (plus) o کاندید های مرحله ی بعد حساب شده اند. چک کردن جواب در پایتون:

	support	itemsets
0	0.625	(A)
1	0.500	(B)
2	0.625	(C)
3	0.750	(D)
4	0.500	(F)
5	0.625	(G)
6	0.375	(A, B)
7	0.375	(A, C)
8	0.500	(D, A)
9	0.500	(D, C)
10	0.375	(G, C)
11	0.500	(D, F)
12	0.375	(D, G)
13	0.375	(D, A, C)

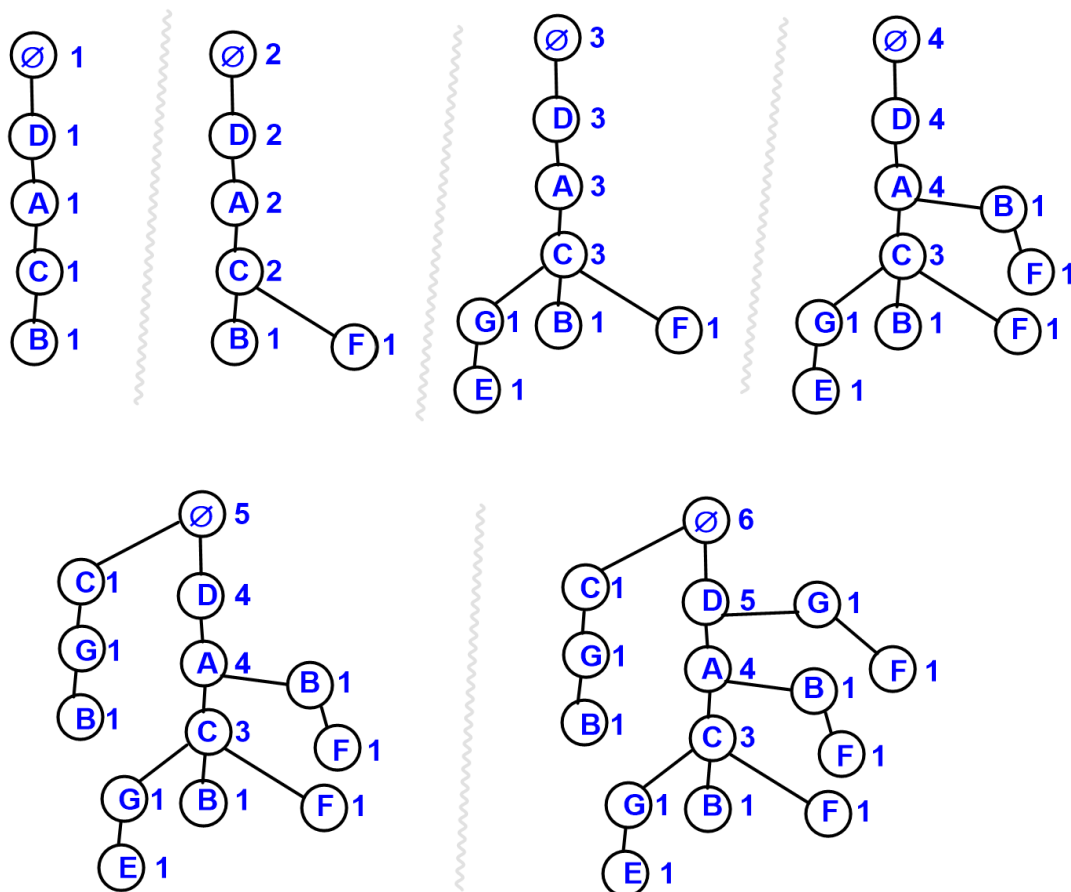
سوال دوم قسمت دوم:

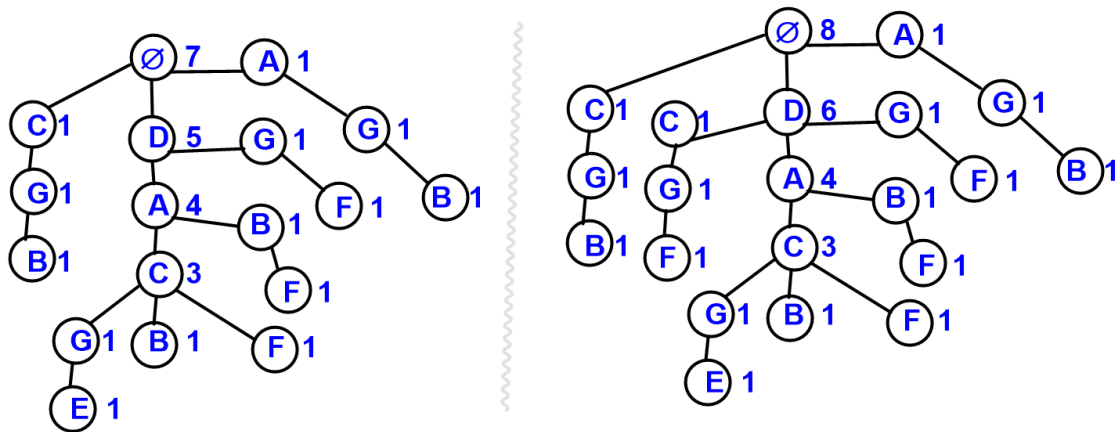
با استفاده از الگوریتم FP-Growth و  $\text{minsup}=3$  داریم:  
مرحله اول: مرتب کردن اقلام در هر سبد با توجه به تعداد تکرار آن ها:

order: D(6) A(5) C(5) G(5) B(4) F(4) E(1)

tid	itemset	
$t_1$	ABCD	DACB
$t_2$	ACDF	DACF
$t_3$	ACDEG	DACGE
$t_4$	ABDF	DABF
$t_5$	BCG	CGB
$t_6$	DFG	DGF
$t_7$	ABG	AGB
$t_8$	CDFG	DCGF

سپس سیدها را در درخت FP-Tree وارد می کنیم.





مرحله ی دوم: استخراج F از FP-Tree با استفاده از درخت های شرطی:

درخت شرطی	شکل درخت	مجموعه اقلام استخراج شده
D		D (6)
A		A (5) DA (4)
C		C (5) AC (3) DC (4) DAC (3)
G	 خط خورده چون 2 تا هست و کمتر از minsup هست.	G (5) DG (3) CG (3)

درخت شرطی	شکل درخت	مجموعه اقلام استخراج شده
B		B (4) AB (2+1)
F		F (4) DF (4)
E		هیچ موردی پیدا نشد.

سوال سوم: برای پیدا کردن قوانین پرتکرار و قوی که بفرم  $A \Rightarrow ?$  هستند ابتدا اقلام پرتکرار را از عناصر F که شامل A هستند پیدا می کنیم.

$\emptyset$						
A (5)	B (4)	C (5)	D (6)	E (1)	F (4)	G (5)
AB (3)	BC (2)	CD (4)	DF (4)		FG (2)	
AC (3)	BD (2)	CF (2)	DG (3)			
AD (4)	BF (1)	CG (3)				
AF (2)	BG (2)					
AG (2)						
ABC (1)		CDG (2)	DFG (2)			
ABD (2)						
ACD (3)						

	sup	confidence	
$A \rightarrow B$	3	$3/5 = 0.6$	پرتکرار و قوی
$A \rightarrow C$	3	$3/5 = 0.6$	پرتکرار و قوی
$A \rightarrow D$	4	$4/5 = 0.8$	پرتکرار و قوی
$A \rightarrow CD$	3	$3/5 = 0.6$	پرتکرار و قوی

با فرض  $\text{min confidence} = 0.6$

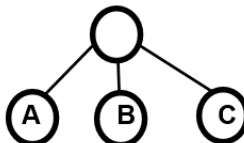
سوال چهارم: تحلیل size و depth درخت FP:

n تراکنش و m کالا

حداکثر	حداقل	
m	1	عمق (depth)
$m + \sum_{i=1}^m i(m-i)$	m	اندازه (size)

عمق به وضوح 0 نمی تواند باشد چون هر کالا حداقل در یک تراکنش آمده است. اما عمق می تواند در شرایطی حتی یک هم شود. باید داشته باشیم  $n \geq m$  و:

tid	itemset
1	A
2	B
3	C

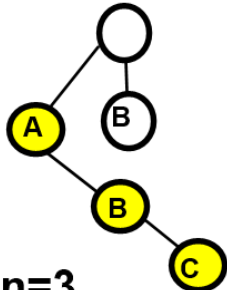


$m=n=3$

البته اگر  $n < m$  باشد حداقل عمق  $\lceil m/n \rceil$  خواهد بود وقتی که سبد ها تقریباً تعداد مساوی کالا داشته باشند.

عمق نمی تواند بیشتر از m باشد. زیرا عمق طولانی ترین مسیر از ریشه به یک برگ است و اگر چنین مسیری باشد درواقع در اثر وارد کردن یک تراکنش ایجاد شده همچنین می دانیم هر تراکنش نمی تواند بیشتر از m کالا داشته باشد. (فرض کنید شخصی کل اقلام فروشگاه را خریده است.)

tid	itemset
1	A
2	B
3	ABC

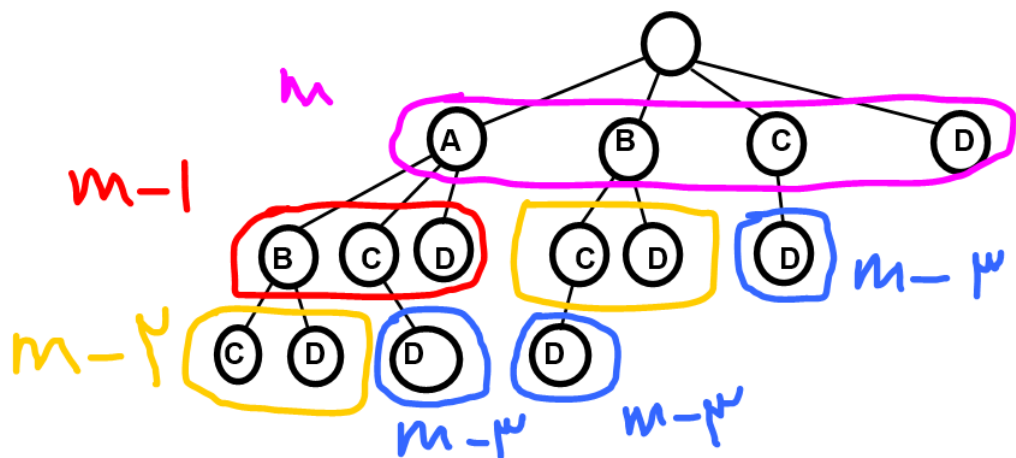


$m=n=3$

در مورد تعداد رئوس هم حداقل m هستند چون هر کالا حداقل یکجا در یک سبدی ظاهر شده پس باید همچنین راسی در درخت هنگام وارد کردن تراکنش ها به وجود آمده باشد. و حالت حداقلی هم واقعا می تواند پیش بیاید می توان از همان مثال اول برای عمق یک استفاده کرد.



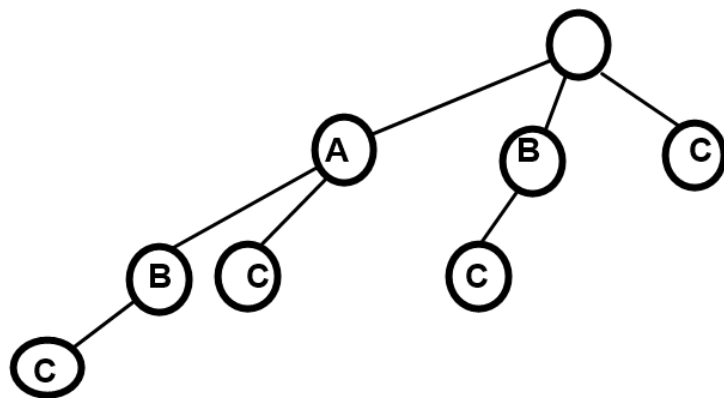
بیشترین تعداد رئوس هم می توان بررسی کرد.



$$m + m-1 + 2(m-2) + 3(m-3) + \dots$$

اگر فرض کنیم  $A B C D \dots$  به ترتیب پر تکرار به کم تکرار باشند درخت فرم بالا را دارد و امکان بیشتر شدن رئوس با توجه به نحوه ی وارد کردن transaction ها وجود ندارد. در ضمن این حالت واقعا ممکن است اتفاق بیافتد و این کران tight هست. مثال:

tid	itemset
1	A
2	AB
3	ABC
4	A
5	B
6	AC
7	BC
8	C
9	A
10	B



$$m + \sum_{i=1}^m i(m-i) = 3 + 1(3-1) + 2(3-2) + 3(3-3) = 7$$

Order: A(6) B(5) C(4)

سوال پنجم (تمرین برنامه نویسی):

(آ) در گام اول محتوای فایل‌های را خوانده و در ساختار داده مناسب قرار دهید. دقت کنید کاراکترها کدگذاری utf8 دارند. لذا موقع خواندن نوع کدگذاری را باید مشخص کنید.

## Reading all the files into a list (data)

```
data=[]

for i in range (0,491):
    file=open('sample/'+str(i)+'.txt', 'r', encoding="utf8")
    data.append(file.read())
    file.close()
```

همه ی داده های متنی را از آدرس فایل sample از 0 تا 490 را می خوانیم و در لیست data قرار می دهیم.

(ب) در گام دوم می‌خواهیم فاصله ویرایشی edit distance بین هر زوج فایل را بدست آوریم و از طریق آن مقدار تشابه برای هر زوج (بر اساس فاصله ویرایشی) را محاسبه کنیم. اگر  $ed(x, y)$  فاصله ویرایشی دو رشته  $x$  و  $y$  باشد، آنگاه تشابه  $x$  و  $y$  بصورت زیر است.

$$sim_{ed}(x, y) = \frac{\max\{|x|, |y|\} - ed(x, y)}{\max\{|x|, |y|\}}$$

## Calculating edit distances

```
file1=open('sample/distances.txt', 'w')
file2= open('sample/max_distances.txt', 'w')
file1.write("Pairs      Edit_Similarity\n")
file2.write("MaxPairs  Max Edit_Similarity\n")
```

ابتدا در همان آدرس فایل sample دو فایل متنی می سازیم با نام های distances و max\_distances که در اولی همه ی فاصله های ویرایشی و در دومی جفت های ماکسیمم را می خواهیم بنویسیم. سطر اول این فایل های متنی نام ستون ها را نوشته ایم.

```

import Levenshtein
import time

start = time.time()
for i in range (0,491):
    max_sim=0
    max_sim_index=-1

    for j in range (i+1,491):
        max_string_length=max(len(data[i]),len(data[j]))
        edit_similarity=\
            (max_string_length - Levenshtein.distance(data[i],data[j]))/max_string_length

        if(edit_similarity>=max_sim):
            max_sim=edit_similarity
            max_sim_index=j

        file1.write(str(i)+" "+str(j)+" "+str(round(edit_similarity, 4))+ "\n")

    file2.write(str(i)+" "+str(max_sim_index)+" "+str(round(max_sim, 4))+ "\n")

file1.close()
file2.close()
end = time.time()

print('It took', round((end-start)/60,2), \
      'minutes to calculate edit distances and their maximum pairs')

```

با استفاده از تابع `time` قبل و بعد از انجام محاسبات `edit_distance` زمان کل محاسبات را اندازه می گیریم. در قسمت محاسبات با دو حلقه ی `for` تمامی `edit distance` ها با استفاده از تابع آماده `Levenshtein.distance` محاسبه شده است و در فایل های مورد نظر نوشته شده اند.

خروجی:

```
It took 7.64 minutes to calculate edit distances and their maximum pairs
```

همان طور که می بینید حدود 7 دقیقه زمان برده تا تمامی `edit distance` ها حساب شوند.

فایل های نتیجه:

sample		distances - Notepad		max_distances - Notepad	
		File Edit Format View Help	File Edit Format View Help	File Edit Format View Help	File Edit Format View Help
		Pairs Edit_Similarity	MaxPairs Max Edit_Similarity		
		0 1 0.1998	0 89 0.2804		
		0 2 0.2307	1 413 0.5997		
		0 3 0.2319	2 263 0.2806		
		0 4 0.229	3 10 0.2941		
		0 5 0.1987	4 296 0.5714		
		0 6 0.229	5 87 0.283		
		0 7 0.2605	6 105 0.2775		
		0 8 0.2337	7 99 0.2742		
		0 9 0.2501	8 288 0.2811		
		0 10 0.2733	9 399 0.5486		
		0 11 0.2209	10 127 0.2954		
		0 12 0.2095	11 136 0.276		
		0 13 0.2422	12 327 0.2732		
		0 14 0.2112	13 136 0.2487		
		0 15 0.2194	14 78 0.2877		
		0 16 0.2343	15 456 0.5046		
		0 17 0.263	16 50 0.2873		
		0 18 0.2194	17 84 0.28		
		0 19 0.2237	18 327 0.2828		
		0 20 0.2192	19 220 0.2763		
		0 21 0.215	20 84 0.2853		
		0 22 0.2676	21 136 0.2775		
		0 23 0.2356	22 84 0.2869		
		0 24 0.2156	23 455 0.6155		
		0 25 0.2264	24 327 0.2816		
		0 26 0.2209	25 84 0.2765		
		0 27 0.2176	26 84 0.2726		
		0 28 0.2258	27 327 0.2829		
		0 29 0.2051	28 99 0.2882		
		0 30 0.2237	29 393 0.5913		
		0 31 0.1948	30 350 0.2688		
		0 32 0.2714	31 307 0.4335		
		0 33 0.2801	32 99 0.2871		
		0 34 0.2597	33 435 0.2854		
		0 35 0.2315	34 390 0.2659		
		0 36 0.2377	35 84 0.2825		

(ج) در گام سوم می‌خواهیم فاصله جاکارد هر دو فایل را بدست آوریم. برای این منظور از ایده bag of words و Shingling که در کلاس توضیح داده شد استفاده می‌کنیم. مقدارهای مختلف برای  $k$  را امتحان کنید. با توجه به خروجی کار، بهترین مقدار را برای  $k$  پیشنهاد دهید. مانند گام قابل زمان اجرای این گام را هم اندازه بگیرید.

## Calculating Jaccard (bag of words)

```
data_in_bags=[]

for j in range (0,491):
    mylist=[]
    words=data[j].split()
    data_in_bags.append(words)

file1=open('sample/jac_sim_bags.txt', 'w')
file2= open('sample/max_jac_sim_bags.txt', 'w')
file1.write("Pairs      Jac_Similarity_Bags\n")
file2.write("MaxPairs  Max Jac_Similarity_Bags\n")
```

ابتدا متن های داخل لیست data را split می کنیم تا به کلمات تبدیل شوند و حاصل را در لیست جدید data\_in\_bags ذخیره می کنیم. همچنین فایل های لازم را می سازیم در آدرس قبلی و برچسب های ستون ها را می نویسیم.

```
def jaccard(a,b):
    return len(a.intersection(b)) / len(a.union(b))
```

تابع برای محاسبه ی jaccard می نویسیم. a و b مجموعه در پایتون هستند.

```
start = time.time()
for i in range (0,491):
    max_jac_sim=0
    max_jac_sim_index=-1

    for j in range (i+1,491):
        jac_similarity= jaccard(set(data_in_bags[i]),set(data_in_bags[j]))

        if(jac_similarity>=max_jac_sim):
            max_jac_sim=jac_similarity
            max_jac_sim_index=j

        file1.write(str(i)+" "+str(j)+" "+str(round(jac_similarity, 4))+"\n")

        file2.write(str(i)+" "+str(max_jac_sim_index)+" "+str(round(max_jac_sim, 4))+"\n")

file1.close()
file2.close()
end = time.time()
```

سپس مراحل مشابه انجام شده است با این تفاوت که بجای edit\_distance از jaccard استفاده کرده ایم.

خروجی:

```
print('It took', round((end-start),2), ' seconds to calculate Jaccard distances\
and their maximum pairs (BAG OF WORDS)')
```

It took 15.13 seconds to calculate Jaccard distances and their maximum pairs (BAG OF WORDS)

همان طور که می بینید حدود 15 ثانیه زمان برده تا تمامی jaccard ها حساب شوند. (این روش بسیار سریع تر از edit\_distance بوده است).

فایل های نتیجه:

jac_sim_bags - Notepad		max_jac_sim_bags - Notepad	
File	Edit Format View Help	File	Edit Format View Help
Pairs	Jac_Similarity_Bags	MaxPairs	Max Jac_Similarity_Bags
0 1	0.08	0 171	0.1241
0 2	0.1071	1 413	0.4286
0 3	0.0963	2 333	0.1432
0 4	0.0704	3 251	0.1675
0 5	0.0811	4 296	0.3829
0 6	0.0928	5 8	0.129
0 7	0.1096	6 200	0.1173
0 8	0.0889	7 333	0.1228
0 9	0.0559	8 234	0.1497
0 10	0.0885	9 399	0.363
0 11	0.0627	10 98	0.1303
0 12	0.0033	11 140	0.0978
0 13	0.0745	12 24	0.1433
0 14	0.0036	13 333	0.0964
0 15	0.1069	14 284	0.1451
0 16	0.0867	15 456	0.3662
0 17	0.0917	16 70	0.1369
0 18	0.0049	17 333	0.1048
0 19	0.0799	18 186	0.139
0 20	0.0747	19 302	0.111
0 21	0.078	20 202	0.1386
0 22	0.0758	21 184	0.1259
0 23	0.0865	22 465	0.1286
0 24	0.0049	23 455	0.4349
0 25	0.1039	24 118	0.1065
0 26	0.0464	25 420	0.1424
0 27	0.0025	26 311	0.102
0 28	0.0917	27 152	0.1315
0 29	0.0897	28 99	0.1359
0 30	0.0662	29 393	0.4384
0 31	0.0026	30 234	0.0966
0 32	0.095	31 307	0.2093
0 33	0.1122	32 234	0.1401
0 34	0.0664	33 333	0.1357
0 35	0.088	34 395	0.0856
0 36	0.1119	35 206	0.1199

محاسبه jaccard با استفاده از ایده ی shingles:

## Calculating Jaccard (shingles)

```
import numpy as np
k=3
data_in_shingles=[]
jac_sim_matrix=np.zeros((491, 491))

for j in range (0,491):
    myset=set([])
    for i in range(1,len(data[j])-k+1):
        shingle=data[j][i:i+k]
        temp_str=''
        for r in range(k):
            temp_str+=str(ord(shingle[r]))
        myset.add(int(temp_str))
    data_in_shingles.append(myset)

file1=open('sample/jac_sim_Shingles.txt', 'w')
file2= open('sample/max_jac_sim_Shingles.txt', 'w')
file1.write("Pairs      Jac_Similarity_Shingles\n")
file2.write("MaxPairs  Max Jac_Similarity_shingles\n")
```

ابتدا طول shingle را 3 قرار داده ایم.  $k=3$ . همچنین یک ماتریس 491 در 491 ایجاد کرده ایم تا از قسمت بالا مثلی آن استفاده کنیم و ضرایب جاکارد را ذخیره کنیم. (این ماتریس بعداً برای محاسبه ی تعداد false positive ها و false negative ها مورد استفاده قرار می گیرد.)

سپس با یک حلقه کل متن ها را می خوانیم و کد عددی شینگل ها را در لیست data\_in\_shingles قرار می دهیم. درواقع data\_in\_shingles یک لیستی از لیست ها هست. که مثلاً می دانیم data\_in\_shingles[i] خود یک لیست شامل تمام شینگل های  $k=3$  برای متن i ام هست که بصورت کد عددی ذخیره شده اند (از مجموعه استفاده شده است تا اثر کلمات تکراری مثل "the" حذف شود). سپس فایل های مورد نظر ایجاد شده و برچسب ستون ها نوشته شده اند.

```

start = time.time()
for i in range (0,491):
    max_jac_sim=0
    max_jac_sim_index=-1

    for j in range (i+1,491):
        jac_similarity= jaccard(data_in_shingles[i],data_in_shingles[j])

        if(jac_similarity>=max_jac_sim):
            max_jac_sim=jac_similarity
            max_jac_sim_index=j

        file1.write(str(i)+" "+str(j)+" "+str(round(jac_similarity, 4))+"\n")
        jac_sim_matrix[i][j]=round(jac_similarity, 4)

    file2.write(str(i)+" "+str(max_jac_sim_index)+" "+str(round(max_jac_sim, 4))+"\n")

file1.close()
file2.close()
end = time.time()

print('It took', round((end-start),2), ' seconds to calculate\
Jaccard distances and their maximum pairs')
```

سپس همان کد ها استفاده شده اند با این تفاوت که از `data_in_shingles` بجای `data_in_bags` استفاده شده و همچنین ضرایب جاکارد در قسمت بالا مثلثی ماتریس `jac_sim_matrix` ذخیره شده اند. بعداً از این ماتریس برای راحت تر کردن محاسبه ی تعداد `false_positives/negatives` ها در قسمت بعدی استفاده می کنیم.

خروجی:

```
It took 30.32  seconds to calculate Jaccard distances and their maximum pairs
```

همان طور که می بینید حدود 30 ثانیه زمان برده تا محاسبات کامل شوند.



فایل های نتیجه:

jac_sim_Shingles - Notepad		max_jac_sim_Shingles - Notepad	
File Edit Format View Help		File Edit Format View Help	
Pairs	Jac_Similarity_Shingles	MaxPairs	Max Jac_Similarity_shingles
0 1	0.3159	0 134	0.4321
0 2	0.4273	1 413	0.5487
0 3	0.2981	2 67	0.4149
0 4	0.3317	3 280	0.3738
0 5	0.3119	4 296	0.5506
0 6	0.3334	5 344	0.3448
0 7	0.37	6 304	0.3549
0 8	0.2895	7 316	0.3675
0 9	0.2714	8 234	0.3398
0 10	0.3596	9 399	0.5433
0 11	0.3092	10 98	0.4115
0 12	0.1751	11 107	0.3366
0 13	0.3265	12 24	0.356
0 14	0.2037	13 193	0.3261
0 15	0.4107	14 284	0.4319
0 16	0.3079	15 456	0.5312
0 17	0.3882	16 332	0.354
0 18	0.1655	17 67	0.4057
0 19	0.3302	18 141	0.3684
0 20	0.319	19 84	0.3564
0 21	0.3078	20 84	0.391
0 22	0.3541	21 42	0.3371
0 23	0.3122	22 89	0.4048
0 24	0.1753	23 455	0.5583
0 25	0.3604	24 118	0.3537
0 26	0.2624	25 88	0.3854
0 27	0.184	26 84	0.3304
0 28	0.3366	27 239	0.374
0 29	0.3063	28 84	0.4085
0 30	0.3144	29 393	0.5787
0 31	0.1705	30 332	0.3326
0 32	0.3737	31 307	0.3983
0 33	0.3916	32 471	0.4296
0 34	0.3497	33 89	0.4391
0 35	0.3462	34 312	0.354
0 36	0.4025	35 42	0.3797

این ایده را برای چند  $k$  دیگر امتحان کرده ام و نتایج را در جدول زیر آورده ام.

زمان		نتیجه
$k=1$	1.45 ثانیه	بسیار سریع هست ولی نتیجه جالب نیست چون تقریباً همه ی زوج متن ها ضریب جاکارد بالا (بالای 0.7) دارند و متن ها را خوب تفکیک نمی کند.
$k=2$	9.85 ثانیه	کمی کند شده است ولی همچنین قدرت تفکیک خوبی ندارد چون بعنوان مثال متن شماره 0 تقریباً با همه متون دیگر 1, 2, ..., 490 تشابه جاکارد حدود 0.5 دارد و انتخاب کردن یک متن خاص مشابه با متن 0 سخت هست.
$k=3$	30.32 ثانیه	قدرت تفکیک نسبتاً خوبی دارد زوج های همراه با 0 از جاکارد 0.1 تا 0.43 هستند.
$k=4$	43.96 ثانیه	خوب نیست چون اغلب زوج متن ها تشابه 0.1 دارند. و خیلی با هم تفاوتی ندارند.
$k=5$	54.49 ثانیه	تشابه ها خیلی پایین هستند بجز (1,413) که 0.3 هست بقیه همه در حد 0.01 یا 0.1 هستند.
$k=6$	61 ثانیه	مانند $k=5$ عمل می کند با تشابه کوچک تر و زمان بیشتر.
$k=10$	71.39 ثانیه	تقریباً همه ی تشابه ها در حد 0.001 هستند و زمان بر است.

بنظر من و تجربه روی این  $k$  ها.  $K=3$  گزینه ی بهتری هست.

البته می توانیم از متن ها اثر انگشت بگیریم و از فرمول سریع تر ضریب جاکارد را تقریب بزنیم:

$$\text{sim}(h(S), h(T)) = \frac{\#\{h_i(S) = h_i(T)\}}{d}$$

این تکنیک (بجز گرفتن اثر انگشت ها) کار مقایسه را ساده تر می کند.

$$d \geq \frac{1}{\delta \epsilon^2 J(S, T)}$$

اگر جاکارد و دلتا را ثابت در نظر بگیریم حدوداً در حد  $1/\epsilon^2$  تابع hash نیاز داریم. خطای دور شدن از ضریب جاکارد هم  $e=0.1$  قرار دهیم. باید حدوداً 100 تا تابع hash بسازیم.

## 100 hash functions to make comparison easier

```

Universal_set=set([])
k=3
n=34231 #first prime after (34,227= size of Universal set)

start = time.time()
for j in range (0,491):
    for i in range(1,len(data[j])-k+1):
        shingle=data[j][i:i+k]
        temp_str=''
        for r in range(k):
            temp_str+=str(ord(shingle[r]))
        mytuple=(shingle,int(temp_str))
        Universal_set.add(mytuple)

end = time.time()
print('time used to compute universal words: ', end-start)

Universal_list=list(Universal_set)
matrix_text=[] #tells us which text has which words

```

ابتدا همه ی data را دوباره می خوانیم و همه ی کلمات مجزا را در Universal\_set می ریزیم. (از مجموعه استفاده می کنیم که هر کلمه (3-shingle) فقط یک بار در مجموعه مادر ظاهر شود.) لیست matrix\_text درواقع لیستی است که قرار است پر کنیم و بگوییم در i امین متن کدام کلمه های 3 حرفی وجود دارند.

همچنین چون کل کلمات 3 حرفی در 491 متن تعداد 34231 است ما عدد 34256 را عدد اول p در نظر می گیریم.

```

start = time.time()

for i in range (0,491):
    column=[]
    for j in range (0,34227):
        if Universal_list[j][1] in data_in_shingles[i]:
            column.append(j)
    matrix_text.append(column)

end = time.time()

print('time used to compute membership of shingles in documents: ', end-start)

```

Matrix\_text حساب می شوند و حالا وضعیت عضویت 3-shingle ها در 491 متن را داریم.

```
def minHash(list,a,b,p):    #ax+b mod p daryaft mikone va listi az hameye
                            #kalamati ke yek document dare
                            # list zir majmue az 3000 kalame madar ast

    min=34231
    min_index=-1
    for i in range (0,len(list)):
        temp=(a*list[i]+b)%p
        if temp < min:
            min= temp
    return min
```

این تابع ثابت های  $a$  و  $b$  و عدد اول  $p$  و کلماتی که یک متن دارد را در فرمت یک لیست می گیرد و  $\text{minHash}$  را حساب کرده و بر می گرداند.

```
def hash_sim(list1,list2):
    count=0
    for i in range (0,len(list1)):
        if list1[i]==list2[i]:
            count+=1
    return round(count/len(list1),4)
```

این تابع هم شباهت تقریبی را حساب می کند.

$$\text{sim}(h(S),h(T)) = \frac{\#\{h_i(S) = h_i(T)\}}{d}$$

```
one_hundred_hash=[]
count=0
import random
while True:
    a=random.randint(1,100)
    b=random.randint(1,100)
    tup=(a,b)

    check=tup in one_hundred_hash

    if check==False:
        count+=1
        one_hundred_hash.append(tup)
    if count==100:
        break
```

در این قسمت 100 زوج  $(a,b)$  مجزا ساخته می شود که بعنوان 100 تابع hash استفاده خواهند شد.

```

one_hundred_hash_result=[]
start = time.time()
for k in range (0,491):
    temp=[]
    for j in range (0,100):
        temp.append(minHash(matrix_text[k],one_hundred_hash[j][0],one_hundred_hash[j][1],34231))
    one_hundred_hash_result.append(temp)
end = time.time()

print('time used to compute hash results on 491 texts: ', end-start)
#print(len(tenhash_result))

```

و در ادامه نتایج توابع hash برای 491 متن حساب می شوند. یک لیست حاوی 491 لیست دیگر که هر کدام 100 عنصر دارند.

```

file1=open('sample/jac_sim_Shingles_hash_estimate.txt', 'w')
file2= open('sample/max_jac_sim_Shingles_hash_estimate.txt', 'w')
file1.write("Pairs      Jac_Similarity_Shingles_hash_estimate\n")
file2.write("MaxPairs  Max Jac_Similarity_shingles_hash_estimate\n")

start = time.time()
for i in range (0,491):
    max_jac_sim=0
    max_jac_sim_index=-1

    for j in range (i+1,491):
        jac_similarity= hash_sim(one_hundred_hash_result[i],one_hundred_hash_result[j])

        if(jac_similarity>=max_jac_sim):
            max_jac_sim=jac_similarity
            max_jac_sim_index=j

        file1.write(str(i)+" "+str(j)+"          "+str(jac_similarity)+"\n")

        file2.write(str(i)+" "+str(max_jac_sim_index)+"          "+str(max_jac_sim)+"\n")

file1.close()
file2.close()
end = time.time()

print('It took', round((end-start),2), ' seconds to calculate\
Jaccard distances and their maximum pairs using 100 hash_estimates')

```

خروجی:

```

time used to compute universal words:  9.094392776489258
time used to compute membership of shingles in documents:  6.729145526885986

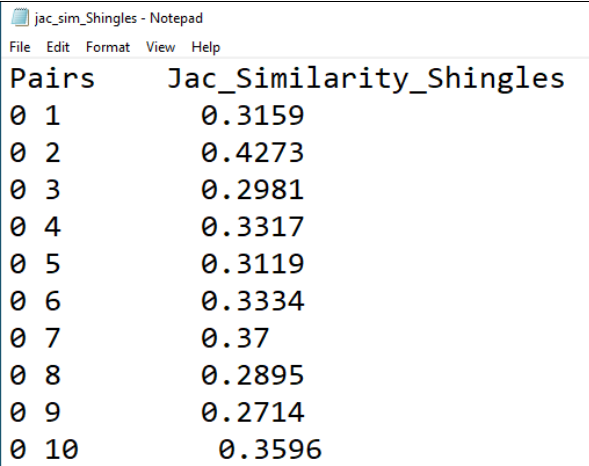
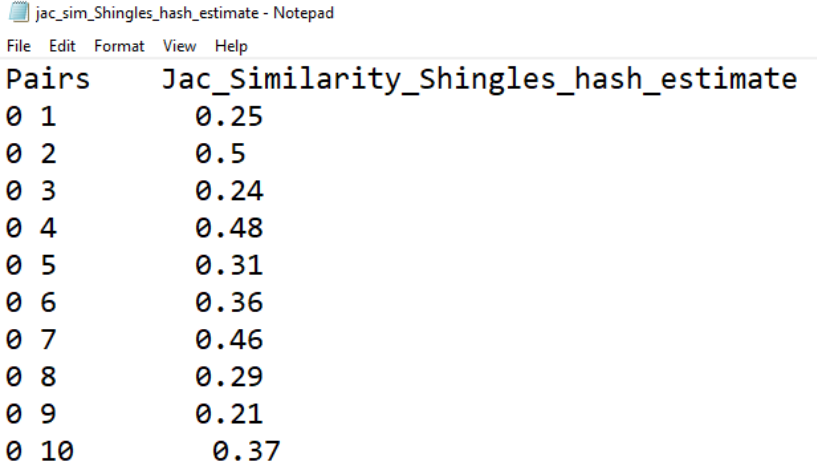
```

```
time used to compute hash results on 491 texts:  22.768384218215942
```

```
It took 2.24  seconds to calculate Jaccard distances and their maximum pairs using 100 hash_e
stimates
```

همان طور که می بینید حدوداً 37 ثانیه گرفتن اثر انگشت ها طول می کشد ولی در نهایت قسمت انجام محاسبات در کمتر از 3 ثانیه انجام می شود چون اثر انگشت های 100 تایی متن ها مقایسه می شوند نه خود متن ها. البته واضح هست که جواب ها تقریبی خواهند بود و با بیشتر کردن توابع hash می توان دقیق تر هم ضرایب جاکارد را تقریب زد.

مقایسه محاسبه ضریب جاکارد و تقریب ضریب جاکارد (با hash):

نتیجه	زمان انجام	
 <pre> jac_sim_Shingles - Notepad File Edit Format View Help Pairs      Jac_Similarity_Shingles 0 1         0.3159 0 2         0.4273 0 3         0.2981 0 4         0.3317 0 5         0.3119 0 6         0.3334 0 7         0.37 0 8         0.2895 0 9         0.2714 0 10        0.3596 </pre>	30 ثانیه	محاسبه ضریب جاکارد
 <pre> jac_sim_Shingles_hash_estimate - Notepad File Edit Format View Help Pairs      Jac_Similarity_Shingles_hash_estimate 0 1         0.25 0 2         0.5 0 3         0.24 0 4         0.48 0 5         0.31 0 6         0.36 0 7         0.46 0 8         0.29 0 9         0.21 0 10        0.37 </pre>	2.4 ثانیه	محاسبه تقریب ضریب جاکارد

(د) در گام آخر می‌خواهیم از ایده MinHash و تکنیک Banding که در کلاس توضیح داده شده استفاده کنیم و زوج‌های  $r$  و  $b$  را بطور تجربی امتحان کرده و برای هر انتخاب، زمان اجرا، تعداد false positive ها و تعداد false negative ها را حساب کنید و گزارش کنید.

برای توابع درهم‌سازی، می‌توانید از توابع بصورت  $h(x) = ax + b \mod p$  وقتی  $p$  عدد اول و  $a$  و  $b$  عدد تصادفی کمتر از  $p$  هستند، استفاده کنید. دقت کنید در این حالت باید زیررشته‌های بطول  $k$  را تبدیل به اعداد صحیح کنید. برای این منظور می‌توانید از کد عددی کاراکترها بهره بجویید.

## MinHash and Banding Technique

Create as many Hash Functions as needed

```
num_of_hash=5
num_of_bands=2

hash_functions=[]
bands=[]
```

می‌خواهیم تعدادی باند و برای هر باند تعدادی تابع hash تصادفی بسازیم. البته عدد 2 و 5 مثال هستند. بهترین مقادیر را در انتهای این فایل word در جدول نوشته‌ام.

```

for k in range(0,num_of_bands):
    count=0
    hash_for_this_band=[]
    while True:

        a=random.randint(1,100)
        b=random.randint(1,100)
        tup=(a,b)

        check=tup in hash_functions

        if check==False:
            count+=1
            hash_functions.append(tup)
            hash_for_this_band.append(tup)
            #print(tup)
        if count==num_of_hash:
            bands.append(hash_for_this_band)
            break

```

در این قسمت توابع hash ایجاد می شوند و هر باند در یک لیست جداگانه قرار می گیرد. به عنوان مثال برای 2 باند و 5 تابع hash لیست bands بفرم زیر خواهد بود:

```

[[ (81, 20), (39, 45), (9, 2), (35, 97), (65, 29)],
 [ (70, 15), (47, 90), (86, 55), (80, 52), (78, 57)]]

```

calculating the results of all hash functions in bands on all 491 documents

```

hash_results_bands=[]
start = time.time()
for k in range (0,num_of_bands):
    hash_results=[]
    for i in range(0,491):
        temp=[]
        for j in range (0,num_of_hash):
            temp.append(\
                minHash(matrix_text[i],bands[k][j][0],bands[k][j][1],34231))
        hash_results.append(temp)
    hash_results_bands.append(hash_results)
end = time.time()

print('time used to compute hash results in bands \
on 491 texts: ', end-start)

```



در قسمت بالا مانند کاری که قبلاً هم کردیم با تابع minHash پاسخ ها را برای هر 491 متن روی تمامی باند ها حساب کرده ایم و در لیست hash\_results\_bands قرار داده ایم. به عنوان مثال برای 2 باند و 5 تابع hash لیست hash\_results\_bands دو عضوی است که عضو اول باند اول و عضو دوم باند دوم است. باند اول خود 491 عنصر دارد که هر کدام 5 عدد هستند و مشابه آن برای باند دوم هم داریم که عضو دوم لیست hash\_results\_bands است.

## Extract the pairs from the similar hash results in bands

```
def partition(hash_results):
    #natayej hash 491 doc ra besurate list migirad
    #va mige koduma ba ham moghayese beshan
    #(tedad hash ha mitavanad harchi bashad)

    spot=[0]*491
    partitions=[]
    for i in range (0,491):
        if spot[i]==0:
            gather=[]
            spot[i]=1
            gather.append(i)
            for j in range (i+1,491):
                if hash_results[j]==hash_results[i]:
                    spot[j]=1
                    gather.append(j)
            partitions.append(gather)
    return partitions
```

این قسمت می خواهیم bucket ها را مشخص کنیم پس تابع partition را تعریف کرده ایم تا یک لیست از نتایج hash های یک باند را بگیرد و نتایج hash هایی که یکسان شده اند را در یک لیست قرار دهد. به عنوان مثال می تواند بگوید:

[[0,3,4] , [2], [10,13,19], ....]

این به این معنی هست که متن های 0 3 4 تحت تابع های hash نتایج یکسانی داشته اند. پس در یک bucket (لیست) هستند.

```
def pair_producer(list):
    #yek listi as list haro migire (natijeye partition=input)
    #va zoj hayi ke bayad
    #bayad moghayese beshan vali hanuz tu hash_pairs nistand
    #ro vared hash_pairs mikone

    for i in range (0,len(list)):
        for j in range (0,len(list[i])):
            for k in range (j+1,len(list[i])):
                tup=(list[i][j],list[i][k])
                check=tup in hash_pairs
                if check==False:
                    hash_pairs.append(tup)
```

این تابع pair هم نتیجه ی partition را می گیرد و تمامی زوج های ممکن را در bucket ها می سازد. مثلاً:

[(0,3), (0,4), (3,4), (10,13), (10,19), (13,19) ... ]

```

start = time.time()
hash_pairs=[]
for i in range(0,num_of_bands):
    a= partition(hash_results_bands[i])
    pair_producer(a)
end = time.time()
print('It took', round((end-start),2), 'seconds \
to calculate pairs from minhash results')

```

این مرحله از دو تابع قبلی استفاده می کند و روی تک تک باند ها زوج های مشابه را پیدا می کند. (این دستور چون روی همه ی باند ها اجرا می شود پس دو متن حتی اگر در یک باند هم مشابه باشند مقایسه می شوند همان ایده ی banding) حال که نتایج را داریم می توانیم عملکرد این ایده را همراه با False Negative/Positive بررسی کنیم. ابتدا کل Positive ها و Negative های واقعی را پیدا می کنیم.

## Check False Positives and False Negatives

```

pos=0
neg=0
for i in range (0,491):
    for j in range (i+1,491):
        if jac_sim_matrix[i][j]>=0.3:
            pos+=1
        else:
            neg+=1

print(pos)
print(neg)

```

47760

72535

تعداد کل زوج ها	تعداد کل + های واقعی	تعداد کل - های واقعی
$\binom{491}{2} = 120,295$	47760	72535

```

false_positive=0
false_negative=0
checked_pairs_matrix=np.zeros((491, 491))

for i in range (0,len(hash_pairs)):
    checked_pairs_matrix[hash_pairs[i][0]][hash_pairs[i][1]]=-1
    if jac_sim_matrix[hash_pairs[i][0]][hash_pairs[i][1]]<0.3:
        false_positive+=1

for i in range (0,491):
    for j in range (i+1,491):
        if checked_pairs_matrix[i,j]==0:
            if jac_sim_matrix[i][j]>=0.3:
                false_negative+=1

print("#{False Positive}      ", false_positive)
print("#{All selected Pairs} ", len(hash_pairs))
print("#{False Negative}     ", false_negative)

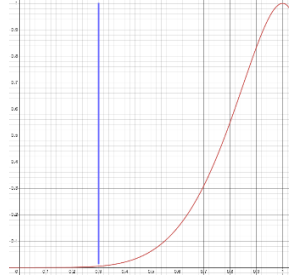
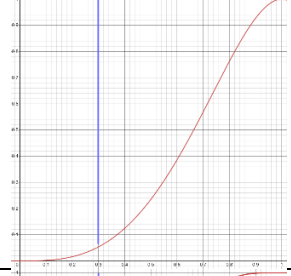
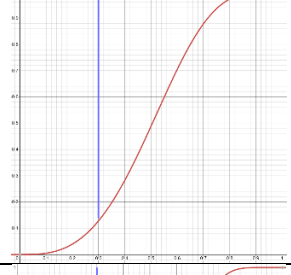
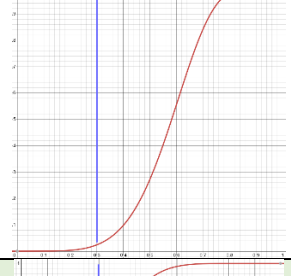
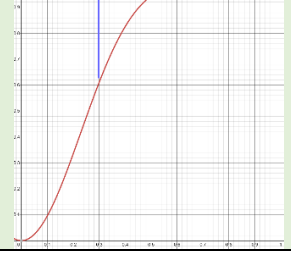
#{False Positive}      103
#{All selected Pairs}  420
#{False Negative}     47443

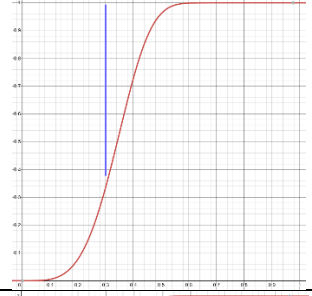
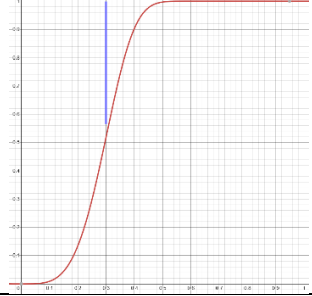
```

برای  $r=5$  و  $b=2$  نتایج خوب نیستند چون کلاً 420 زوج به عنوان مثبت (تشابه بیشتر از 0.3) انتخاب شدند و از این 420 مورد 103 تا هم اشتباهی مثبت شده اند پس درواقع از 47760 مثبت واقعی فقط 317 تا مثبت را توانسته تشخیص دهد. البته این مقادیر عوض می شوند چون hash ها تصادفی هستند. در جدول بعد یک اجرای دیگه از  $r=5$  و  $b=2$  را می بینیم که باز هم خوب نیست.

در ادامه برای مقادیر مختلف تست کرده ایم:

FP: false positive    FN: false negative    b: تعداد باند ها    r: تعداد توابع

r	b	$1 - (1 - x^r)^b$	Time(s)	FP	FN	All Pairs	F1-score
5	2		16.63	312	47,135	937	0.013
3	2		15.96	1,157	45,040	3,877	0.053
3	5		43.41	15,417	21,111	42,066	0.297
5	10		25.48	702	44,249	4,213	0.068
2	10		131.71	40,417	8,906	79,271	0.306

r	b	$1 - (1 - x^r)^b$	Time(s)	FP	FN	All Pairs	F1-score
4	50		90.94	14,356	19,264	42,852	0.314
4	90		157.18	19,702	13,341	54,121	0.338

(استاد من معیار F1-score رو از درس "نظریه یادگیری" استفاده کردم. چون هم Precision و هم Recall داره هم FN و FP رو در نظر می گیره بنظرم میتونه معیار خوبی باشه تو این مسئله).

بعد از تجربه های بالا  $r=2$ ,  $b=10$  بنظرم گزینه ی مناسبی است چون F1-score خوبی دارد و تقریباً 38K از 47K متنی که جاکارد بالای 0.3 دارند را پیدا کرده است همچنین زمان اجرا حدود 2 دقیقه است. البته این زمان تمامی زمان ها برای محاسبه ی مجموعه ی مادر و تشخیص عضویت شینگل ها در متن ها هم می شود. البته دو مورد آخر  $r=4$ ,  $b=50/90$  هم خوب هستند ولی بنظرم تو این مسئله Recall مهم تر هست که ما دوتا متنی که شبیه هم هستند رو miss نکنیم و این یعنی کم تر بودن FALSE NEGATIVE که در مورد  $r=2$ ,  $b=10$  کمینه هست.