



دانشگاه تهران

دانشکده علوم و فنون نوین

تمرین شماره دو درس پردازش دادگان انبوه

فاطمه چیت ساز	نام و نام خانوادگی
830402092	شماره دانشجویی
1403 خرداد 1403	تاریخ ارسال گزارش

سوال یک

فیلتر بلوم یه ساختار داده‌ی احتمالیه که برای چک کردن عضویت یه عنصر در مجموعه استفاده میشه این ساختار خیلی بهینه‌ست از لحاظ حافظه و زمان پردازش ولی یه ویژگی خاص داره: ممکنه بعضی وقت‌ها نتیجه‌ی مثبت اشتباه بده: یعنی بگه یه عنصر در مجموعه هست در حالی که نیست اما هیچوقت نتیجه‌ی منفی اشتباه نمیده

ابدا از pandas استفاده کردیم تا داده‌های کد ملی رو از فایل CSV بخونیم و اوナ رو به یه لیست تبدیل کنیم اینطوری دسترسی به همه کدهای ملی داریم

```
df = pd.read_csv('unique_ids.csv')
national_ids = df['id'].tolist()

valid_ids = random.sample(national_ids, 1000)
```

valid_ids
[539431, 712008, 476475, 500606, 215353, 740443, 685958, 128827, 925001, 568460, 279651, 825013, 748046, 754836, 522141, 721770, 305695, 148748, 340729, 852810, 822852, 135483, 340601, 733551, 779864, ... 184857, 312565, 489596, 319447, 481432]

انتخاب کدهای مجاز و غیرمجاز:

```
invalid_ids = []
while len(invalid_ids) < 1000:
    fake_id = str(random.randint(1000000000, 9999999999))
    if fake_id not in national_ids:
        invalid_ids.append(fake_id)

valid_ids

[539431,
 712008,
 476475,
 500606,
 215353,
 740443,
 685958,
 128827,
 925001,
 568460,
 279651,
 825013,
 748046,
 754036,
 522141,
 721770,
 305695,
 148748,
```

برای این کار از `randomsample` استفاده کردیم تا هزار تا کد ملی مجاز رو از بین کدهای موجود انتخاب کنیم برای تولید کدهای ملی غیرمجاز به حلقه‌ی `while` نوشتیم که توی هر تکرار یه کد ملی تصادفی تولید می‌کرد و چک می‌کرد که این کد توی لیست اصلی نباشه اگه نبود اون رو به لیست کدهای غیرمجاز اضافه می‌کرد

پیاده‌سازی فیلتر بلوم

```

class BloomFilter:
    def __init__(self, size, hash_count):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = [0] * size

    def _hashes(self, item):
        result = []
        for i in range(self.hash_count):

            hash_result = int(hashlib.md5((str(item) + str(i)).encode('utf-8')).hexdigest(), 16) % self.size
            result.append(hash_result)
        return result

    def add(self, item):
        for hash_value in self._hashes(item):
            self.bit_array[hash_value] = 1

    def check(self, item):
        for hash_value in self._hashes(item):
            if self.bit_array[hash_value] == 0:
                return False
        return True

```

فیلتر بلوم از یه بیتمپ با اندازه ثابت و چندین تابع هش مستقل استفاده میکنه وقتی یه عنصر رو به فیلتر اضافه میکنی این مراحل انجام میشه:

عنصر رو با هر کدوم از توابع هش میکنی و یه سری اندیس به دست میاری بیت‌های متناظر با این اندیس‌ها رو توی بیتمپ روی ۱ تنظیم میکنی برای چک کردن اینکه یه عنصر در فیلتر هست یا نه:

عنصر رو با هر کدوم از توابع هش میکنی و اندیس‌ها رو به دست میاری چک میکنی که همه‌ی بیت‌های متناظر با این اندیس‌ها در بیتمپ برابر ۱ باشن اگه حتی یکی از بیت‌ها صفر باشه عنصر قطعاً در فیلتر نیست اگه همه بیت‌ها ۱ باشن عنصر احتمالاً در فیلتر هست

برای پیاده سازی ما یه کلاس BloomFilter درست کردیم که مسئول نگهداری و مدیریت فیلتر بلوم بود تو این کلاس سه تا تابع اصلی داشتیم:

init بیتمپ و تعداد توابع هش رو تنظیم میکنه و یه بیتمپ خالی میسازه

: hashes یه لیست از هش‌های مختلف برای یه آیتم رو برمی‌گردونه برای این کار از hashlibmd5 استفاده کردیم و هش‌ها رو با اضافه کردن یه مقدار متفاوت به رشتہ اصلی به دست آوردیم

: add یه آیتم رو به فیلتر اضافه می‌کنه برای این کار همه هش‌های آیتم رو به دست میاره و بیت‌های متناظر با این هش‌ها رو توی بیت‌مپ روی 1 تنظیم می‌کنه

: check این تابع چک می‌کنه که یه آیتم در فیلتر هست یا نه برای این کار همه هش‌های آیتم رو به دست میاره و چک می‌کنه که همه بیت‌های متناظر با این هش‌ها در بیت‌مپ برابر 1 باشن یا نه

راهاندازی سرور :

```
def server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind(('localhost', 1010))
    server_socket.listen(5)
    print("Running server on port 1010")

    while True:
        client_socket, addr = server_socket.accept()
        print(f"Connected from {addr}")
        client_handler = threading.Thread(target=handle_client, args=(client_socket,))
        client_handler.start()
```

سرور رو راهاندازی کردیم که توی یه ترد جداگانه ران میشه سرور منظر می‌مونه تا کلاینت‌ها وصل بشن و وقتی یه کلاینت وصل شد یه ترد جدید برای هندل کردن درخواست‌های اون کلاینت ایجاد می‌کنه سرور وظیفه داره که کد ملی رو از کلاینت بگیره اونو با فیلتر بلوم چک کنه و نتیجه رو به کلاینت برگردونه (در واقع میشد اینا رو ساده تر زد و سرور و یک تابع زد و کلاینت رو یک تابع جدا ولی خب اینطوری خوشگل تر مفهوم سرور و کلاینت رو میرسوند)

راهاندازی کلاینت

```

def client(national_ids):
    valid_count = 0
    invalid_count = 0
    false_positive_count=0
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 1010))

    for national_id in national_ids:

        client_socket.send(pickle.dumps(national_id))

        response = client_socket.recv(1024).decode('utf-8')

        if(response =='valid'):
            valid_count+=1
            if national_id in invalid_ids:
                false_positive_count += 1
        if(response =='invalid'):
            invalid_count+=1

    print(f"natinal code: {national_id}, is: {response}")

    print(f"Number of valid inputs: {valid_count}")
    print(f"Number of invalid inputs: {invalid_count}")
    print(f"Number of false positives: {false_positive_count}")
    client_socket.close()

```

کلاینت هم توی یه ترد جداگانه ران میشه این کلاینت کدهای ملی رو یکی یکی به سرور می فرسته و جواب رو دریافت می کنه در نهایت تعداد کدهای مجاز غیرمجاز و تعداد false positive ها رو چاپ می کنه

تستها

برای تست کردن کد چندین ترکیب مختلف از تعداد توابع هش و اندازه بیت مپ رو امتحان کردیم:

یه تابع هش و سایز بیت مپ 20000

```

Number of valid inputs: 1051
Number of invalid inputs: 949
Number of false positives: 51

```

```
all_ids = valid_ids + invalid_ids
random.shuffle(all_ids)
client(all_ids)

Connected from ('127.0.0.1', 62012)
natinal code: 173920, is: valid
natinal code: 739539, is: valid
natinal code: 575315, is: valid
natinal code: 675344, is: valid
natinal code: 4642103074, is: invalid
natinal code: 6618525012, is: invalid
natinal code: 4624114369, is: invalid
natinal code: 461589, is: valid
natinal code: 1904081763, is: invalid
natinal code: 9941302983, is: invalid
natinal code: 786010, is: valid
natinal code: 1706546114, is: invalid
natinal code: 5015224770, is: invalid
natinal code: 1363655284, is: valid
natinal code: 970384, is: valid
natinal code: 3798775362, is: invalid
natinal code: 801347, is: valid
natinal code: 3983725361, is: invalid
natinal code: 991571, is: valid
natinal code: 470902, is: valid
natinal code: 182803, is: valid
natinal code: 2289721813, is: valid
natinal code: 204402, is: valid
natinal code: 5154919011, is: invalid
...

```

سہ تابع ہش و سایز بیت مپ 20000

increase number of hash function

```
bloom_filter = BloomFilter(size=20000, hash_count=3)

for id in valid_ids:
    bloom_filter.add(id)
```

```
Number of valid inputs: 1004  
Number of invalid inputs: 996  
Number of false positives: 4
```

یه تابع هش و سایز بیت‌مپ 80000

increase size of bitmap

```
bloom_filter = BloomFilter(size=80000, hash_count=1)
```

```
for id in valid_ids:  
    bloom_filter.add(id)
```

```
Number of valid inputs: 1011  
Number of invalid inputs: 989  
Number of false positives: 11
```

نتیجه‌گیری:

تعداد توابع هش: با افزایش تعداد توابع هش دقت فیلتر بلوم بهتر می‌شود ولی زمان پردازش هم بیشتر می‌شود

اندازه بیت‌مپ: با افزایش اندازه بیت‌مپ احتمال برخورد هش‌ها (collision) کمتر می‌شود و در نتیجه احتمال false positive کمتر می‌شود

چگون طور کاهش احتمال false positive به ۱٪

برای کاهش احتمال false positive به ۱٪ باید نسبت مناسبی از تعداد توابع هش و اندازه بیت‌مپ را انتخاب کنیم می‌توانیم از فرمول زیر استفاده کنیم:

$$k \left(kn/m - e - 1 \right)$$

که این همون false positive rate ماست که k تعداد hash function ها و m سایز اون بیت مپه و n سایز اون ایتم هامونه حالا اگر اینو بذاریم مساوی 001 که سوال گفته میتوانی مقادیر دلخواهمون رو پیدا کنیم

سوال دو

آمده‌سازی متن:

```
import re

def preprocess_text(text):
    text = re.sub(r'^[^\w+]+$', '', text).lower()
    return text.split()

with open('Shakespeare.txt', 'r') as file:
    text = file.read()

words = preprocess_text(text)
words
```

```
['the',
 'project',
 'gutenberg',
 'ebook',
 'of',
 'the',
 'complete',
 'works',
 'of',
 'william',
 'shakespeare',
 'by',
 'william',
 'shakespeare',
 'this',
 'ebook',
 'is',
 'for',
 'the',
 'use',
 'of',
 'anyone',
 'anywhere',
 'at',
 'no',
 ...
 'project',
 'gutenberg',
 'is',
 'dedicated',
 ...]
```

اول از همه متن آثار شکسپیر رو از فایل خوندیم و یه کم تمیزش کردیم در واقع یعنی او مدیم کاراکترهای غیرحروفی و عددی مثل نقطه و ویرگول رو حذف کردیم و متن رو به حروف کوچک تبدیل کردیم اینجوری کلمات به شکلی استاندارد و ساده‌تر آماده پردازش می‌شن

تعريف توابع هش:

```
import random
import hashlib

def generate_hash_functions(num_funcs, max_bits):
    hash_funcs = []
    for _ in range(num_funcs):
        seed = random.randint(0, 100000)
        def hash_func(x, seed=seed):
            return int(hashlib.md5((str(x) + str(seed)).encode()).hexdigest(), 16) % (2**max_bits)
        hash_funcs.append(hash_func)
    return hash_funcs

hash_functions = generate_hash_functions(35, 24)
```

خب الگوریتم FlajoletMartin برای کار کردن نیاز به یه سری تابع هش داره اینجا ما 35 تا تابع هش درست کردیم که هر کدام با یه seed تصادفی مقداردهی می‌شه تابع هش ما از استفاده می‌کنه و خروجی رو به یه عدد 24 بیتی تبدیل می‌کنه

محاسبه تعداد صفرهای پایانی:

```
def trailing_zeros(n,max_bits):
    if n == 0:
        return max_bits
    return (n & -n).bit_length() - 1
```

تو الگوریتم FlajoletMartin ما دنبال تعداد صفرهای پایانی در نمایش باینری هش‌ها هستیم برای چی؟ چون تعداد این صفرها به ما یه نشونه از تعداد کلمات متمایز میده هر چقدر تعداد صفرهای پایانی بیشتر باشه احتمال بیشتری داره که کلمات بیشتری تو استریم داریم

اجرای الگوریتم FlajoletMartin

```

def flajolet_martin(stream, hash_functions,max_bits):
    max_zeros = [0] * len(hash_functions)
    for word in stream:
        for i, hash_func in enumerate(hash_functions):
            hash_value = hash_func(word)
            max_zeros[i] = max(max_zeros[i], trailing_zeros(hash_value,max_bits))
    return [2**z for z in max_zeros]

```

برای هر کلمه تو استریم:

1 اون رو با هر 35 تا تابع هش هش می‌کنیم

2 تعداد صفرهای پایانی هر هش رو محاسبه می‌کنیم

3 اگه تعداد صفرهای پایانی هش فعلی بیشتر از حداکثر قبلی بود اون رو جایگزین می‌کنیم

در نهایت با استفاده از 2 به توان تعداد صفرهای پایانی یه تخمینی از تعداد کلمات متمایز می‌زنیم

روش‌های مختلف تخمین:

```

def estimate_cardinality(estimates, method='median'):
    if method == 'mean':
        return np.mean(estimates)
    elif method == 'median':
        return np.median(estimates)

    elif method == 'categorized_median':
        num_categories = 5
        chunk_size = len(estimates) // num_categories
        means = []

        for i in range(num_categories):
            chunk = estimates[i*chunk_size:(i+1)*chunk_size]
            means.append(np.mean(chunk))

        return np.median(means)
    else:
        raise ValueError("Unknown method")

```

حالا ما 35 تا تخمین از تعداد کلمات متمایز داریم چطور می‌تونیم از اینا یه نتیجه دقیق‌تر بگیریم؟ سه روش رو امتحان کردیم:

میانگین (Mean): همه تخمین‌ها رو میانگین می‌گیریم

میانه (Median): میانه تخمین‌ها رو حساب می‌کنیم

35 تابع هش و طول رشته بیتی 32 بیت: طول رشته بیتی بیشتر

35 hash function max bit = 32

```
hash_functions = generate_hash_functions(35, 32)
```

```
len(estimate)
```

```
50
```

```
Mean Estimate: 96665.6  
Median Estimate: 32768.0  
categorized_median Mean Estimate: 50322.28571428572
```

نتایج واقعی :

true_count

```
unique_words = set(words)  
true_count = len(unique_words)  
print("True count of unique words:", true_count)
```

```
True count of unique words: 27523
```

تحلیل نتایج

تعداد توابع هش: با افزایش تعداد توابع هش دقیق تخمین افزایش پیدا می کنند اما زمان پردازش هم بیشتر می شود

طول رشته بیتی: با افزایش طول رشته بیتی دقیق تخمین بهتر می‌شود چون احتمال برخورد (collision) هش‌ها کمتر می‌شود ولی حافظه بیشتری هم مصرف می‌شود

نتیجه‌گیری:

بیشترین دقیق روش میانه دسته‌بندی شده گرفتیم این روش تعادلی بین میانگین و میانه برقرار می‌کنند و نوسانات رو کاهش میدهند همچنانی با تعداد توابع هش بیشتر و طول رشته بیتی بیشتر دقیق تخمین بهتر می‌شود ولی مصرف حافظه و زمان پردازش هم بیشتر می‌شود

سؤال سه

قسمت نوشتني :

دوستی

Confidence

$$\text{Confidence}(A \rightarrow B) = \frac{P(A \cap B)}{P(A)}$$

اگر خروجی احتمال $P(B)$ را داشته باشیم احتمال $P(A \cap B)$ را بمحض مسأله در مطابع مسأله

دراخواج مسأله کوچک را مخصوصاً نشون می‌نماییم $P(B)$ چیزی است که $P(A \cap B)$ را در آنها و مبنای ساختار $P(A \cap B)$ را در آنها داشته باشد $P(A)$ چیزی است که $P(A \cap B)$ را در آنها داشته باشد

اگر $P(B)$ را داشود و $P(A \cap B)$ را داشته باشد $P(A \cap B) / P(B)$ Convictions lift می‌گویند

پس اگر $P(A) = 100$ و $P(B) = 100$ باشند

$$P(A \cap B) = 100$$

$$P(\bar{A} \cap \bar{B}) = 100$$

$$\text{Conf}(A \rightarrow B) = \frac{100}{100} = 1$$

حرج نهادن حسنه احتمال مسأله را در آنها داشته باشد

$$\text{Conf}(B \rightarrow A) = \frac{100}{100} = 1$$

حرج نهادن حسنه احتمال مسأله را در آنها داشته باشد

$$\text{Conf}(A \rightarrow B) = \frac{P(A \cap B)}{P(A)} \neq \frac{P(A \cap B)}{P(B)} = P(B \rightarrow A)$$

Conf \leftarrow نسبة التحقق من صحة الـ #

$$lift(A \rightarrow B) = \frac{P(A \cap B)}{P(A) \cdot P(B)} = \frac{P(A \cap B)}{P(B) \cdot P(A)} = lift(B \rightarrow A)$$

نسبة التحقق من lift #

$$\text{Conv}(A \rightarrow B) = \frac{1 - P(B)}{1 - \text{Conf}(A \rightarrow B)} = \frac{1 - P(B)}{1 - \frac{P(A \cap B)}{P(A)}}$$

≠

$$\text{Conv}(B \rightarrow A) = \frac{1 - P(A)}{1 - \text{Conf}(B \rightarrow A)} = \frac{1 - P(A)}{1 - \frac{P(A \cap B)}{P(B)}}$$

نسبة التتحقق من صحة الـ ✓

$$A \rightarrow B \quad \text{معنی} \quad P(B|A) = 1$$

$$\underline{\text{Conf}}(A \rightarrow B) = \frac{P(A \cap B)}{P(A)} = P(B|A) = \underline{\underline{1}}$$

$$\underline{\text{lift}}(A \rightarrow B) = \frac{P(A \cap B)}{P(A) P(B)} \quad P(B|A) = 1 \Rightarrow P(A \cap B) = P(A)$$

$$= \frac{P(A)}{P(A) P(B)} \Rightarrow \frac{1}{P(B)} \quad \text{معنی} \quad \text{لطف}$$

$$\underline{\text{Conviction}}(A \rightarrow B) = \frac{1 - P(B)}{1 - \text{Conf}(A \rightarrow B)}$$

$$\frac{1 - P(B)}{1 - \frac{P(A \cap B)}{P(A)}} = \frac{1 - P(B)}{1 - 1} = \frac{1 - P(B)}{0} = \infty$$

قسمت کدی :

پیدا کردن دو تایی های پر تکرار

خوندن داده ها

```
with open('browsing.txt', 'r') as file:  
    transactions = [line.strip().split() for line in file]
```

اول از همه باید داده ها رو از فایل browsing.txt بخونیم این فایل شامل خطوطیه که هر کدوم نشون دهنده یه جلسه مرور کاربره و هر آیتم با یه فاصله از آیتم دیگه جدا شده:

```
transactions  
  
['FR011987', 'ELE17451', 'ELE89019', 'SNA90258', 'GR099222'],  
[  
    'GR099222',  
    'GR012298',  
    'FR012685',  
    'ELE91550',  
    'SNA11465',  
    'ELE26917',  
    'ELE52966',  
    'FR090334',  
    'SNA30755',  
    'ELE17451',  
    'FR084225',  
    'SNA80192'],  
[  
    'ELE17451', 'GR073461', 'DAI22896', 'SNA99873', 'FR086643'],  
    'ELE17451', 'ELE37798', 'FR086643', 'GR056989', 'ELE23393', 'SNA11465'],  
    'ELE17451',  
    'SNA69641',  
    'FR086643',  
    'FR078087',  
    'SNA11465',  
    'GR039357',  
    'ELE28573',  
    'ELE11375',  
    'DAI54444'],  
    'ELE17451',  
    .  
    'DAI85839',  
    'DAI70456',  
    'DAI83733',  
    'GR035122'],  
    ... ]
```

اینجا فایل رو باز می‌کنیم و هر خط رو به یه لیست از آیتم‌ها تبدیل می‌کنیم در نهایت یه لیست از لیست‌ها داریم که هر کدام نشونده‌ند یه جلسه مرور کاربره

شمارش آیتم‌ها

```
item_counts = defaultdict(int)
for transaction in transactions:
    for item in transaction:
        item_counts[item] += 1
```

حالا باید بفهمیم هر آیتم چند بار تکرار شده تا بتونیم آیتم‌های پرتکرار رو شناسایی کنیم:

```
item_counts
defaultdict(int,
{'FR011987': 104,
'ELE17451': 3875,
'ELE89019': 38,
'SNA90258': 550,
'GRO99222': 906,
'GRO12298': 385,
'FR012685': 23,
'ELE91550': 23,
'SNA11465': 142,
'ELE26917': 2292,
'ELE52966': 380,
'FR090334': 63,
'SNA30755': 456,
'FR084225': 74,
'SNA80192': 258,
'GRO73461': 3602,
'DAI22896': 1219,
'SNA99873': 2083,
'FR086643': 235,
'ELE37798': 534,
'GRO56989': 655,
'ELE23393': 66,
'SNA69641': 599,
'FR078087': 1531,
..
'FR094222': 6,
'ELE35887': 13,
'FR029984': 38,
'DAI82972': 17,
```

اینجا از defaultdict استفاده کردیم تا شمارش هر آیتم رو انجام بدمیم برای هر تراکنش هر آیتم رو شمردیم

شناسایی آیتم‌های پرتکرار

حالا آیتم‌ایی که بیش از 100 بار تکرار شدن رو شناسایی می‌کنیم:

```
min_support = 100
frequent_itemsets = {item for item, count in item_counts.items() if count >= min_support}
print(f"item with support > 100: \n {frequent_itemsets}")
print("\n count:\n", len(frequent_itemsets))
```

اینجا یه مجموعه از آیتم‌های پرتکرار درست کردیم که تعداد تکرار هر کدام حداقل 100 بار باشه

```
item with support > 100:
{'FR074121', 'GRO32524', 'FR098878', 'FR098729', 'SNA30579', 'ELE20347', 'DAI32836', 'ELE69630', 'ELE74009', 'ELE78169', 'FR04
count:
647
```

پیدا کردن مجموعه‌های پرتکرار:

برای پیدا کردن مجموعه‌های پرتکرار دو آیتمی و سه آیتمی از یه تابع کمک گرفتیم:

```
def find_frequent_itemsets(transactions, items, k):
    itemsets = defaultdict(int)
    for transaction in transactions:
        for itemset in combinations(set(transaction) & items, k):
            itemsets[itemset] += 1
    return {itemset: count for itemset, count in itemsets.items() if count >= min_support}
```

اینجا با استفاده از ترکیب‌ها (combinations) مجموعه‌های دو آیتمی پرتکرار رو پیدا کردیم همونطور که دیدی ترکیب هر دو آیتم از آیتم‌های پرتکرار رو محاسبه می‌کنیم و تعداد وقوع اونها رو می‌شمریم در نهایت فقط ترکیب‌هایی که تعداد وقوعشون بیشتر از 100 باشه رو نگه می‌داریم

دو تایی ها :

```
frequent_itemsets_2 = find_frequent_itemsets(transactions, frequent_itemsets, 2)
print("frequent_itemsets_2:\n", frequent_itemsets_2)
```

```
frequent_itemsets_2:
{('ELE17451', 'GRO99222'): 148, ('SNA30755', 'ELE17451'): 108, ('ELE17451', 'ELE26917'): 287, ('GRO99222', 'ELE26917'): 159, ('SNA99873', 'ELE17451'): 266, ('SNA99873', 'GRO99222'): 108}
```

تولید قوانین انجمنی

حالا باید قوانین انجمنی رو بر اساس مجموعه‌های پر تکرار تولید کنیم:

```
def generate_rules(frequent_itemsets):
    itemset_support = {itemset: support for itemset, support in frequent_itemsets}
    rules = []

    for itemset, support in frequent_itemsets.items():
        for i in range(1, len(itemset)):
            for antecedent in combinations(itemset, i):
                consequent = tuple(set(itemset) - set(antecedent))
                if consequent:
                    antecedent_support = itemset_support[antecedent] if len(antecedent) > 1 else item_counts[antecedent[0]]
                    confidence = support / antecedent_support
                    rules.append((antecedent, consequent, support, confidence))

    return rules

rules = generate_rules(frequent_itemsets_2)
print(f"Rules: {rules}")
```

اینجا برای هر مجموعه پر تکرار قوانین انجمنی رو تولید می‌کنیم به این صورت که برای هر زیر مجموعه (antecedent) از مجموعه اصلی (itemset) مجموعه باقی‌مانده (consequent) رو محاسبه می‌کنیم بعدش اعتقاد (confidence) قانون رو محاسبه می‌کنیم و قانون رو به لیست قوانین اضافه می‌کنیم

ی همچین ریختی :

```
Rules: [((('ELE17451',), ('GRO99222',), 148, 0.03819354838709677), (('GRO99222',), ('ELE17451',), 148, 0.16335540838852097),
```

(ELE17451,), (GRO99222,), 148, 003819354838709677) یعنی چی ؟

این خروجی نشون می‌ده که یه قانون انجمنی داریم که می‌گه اگه کسی ELE17451 رو مرور کنه احتمال زیادی داره که گروه GRO99222 رو هم مرور کنه یعنی:

ELE17451): این قسمت آیتم اول (antecedent) رو نشون می‌ده که ELE17451 هست

GRO99222): این قسمت آیتم دوم (consequent) رو نشون می‌ده که GRO99222 هست

148: این عدد تعداد دفعاتی رو نشون می‌ده که این دو آیتم با هم مرور شدن

003819354838709677: این عدد قانون رو نشون می‌ده یعنی حدود 38٪

از دفعاتی که ELE17451 مرور شده GRO99222 هم مرور شده

فیلتر و مرتب‌سازی قوانین

در نهایت باید قوانینی که اعتماد بالایی دارن رو فیلتر کنیم و پنج قانون برتر رو پیدا کنیم:

```
min_confidence=0.985  
high_confidence_rules = [rule for rule in rules if rule[3] > min_confidence]  
top_rules = sorted(high_confidence_rules, key=lambda x: (-x[3], x[0]))
```

اینجا قوانین با اعتماد بالاتر از 0985 رو فیلتر کردیم و بعدش اونها رو بر اساس اعتماد به صورت نزولی مرتب کردیم اگه اعتماد دو قانون برابر بود اونها رو به صورت الفبایی بر اساس سمت چپ قوانین (antecedent) مرتب کردیم

که خوب خروجی میشه :

```
top_rules[:5]
```

```
[('GR038636',), ('FR040251',), 106, 0.9906542056074766]
```

قسمت بعدی سه تایی های پر تکرار :

همانطور که در قسمت قبل دیدیم ابتدا آیتم های پر تکرار دو تایی رو پیدا کردیم حالا با استفاده ازتابع find_frequent_itemsets سه تایی پر تکرار رو پیدا می کنیم:

```
frequent_itemsets_2 = find_frequent_itemsets(transactions, frequent_itemsets, 2)  
frequent_itemsets_3 = find_frequent_itemsets(transactions, frequent_itemsets, 3)  
print("Frequent Itemsets of size 3:\n", sorted(frequent_itemsets_3.items(), key=lambda x: (-x[1], x[0])))
```

اینجا با استفاده از ترکیب ها (combinations) مجموعه های سه آیتمی پر تکرار رو پیدا کردیم همانند قبل ترکیب هر سه آیتم از آیتم های پر تکرار رو محاسبه می کنیم و تعداد وقوع اونها رو می شمریم در نهایت فقط ترکیب هایی که تعداد وقوعشون بیشتر از 100 باشند رو نگه می داریم

```
Frequent Itemsets of size 3:  
[('FR040251', 'DAI75645', 'SNA80324'), 374], (('FR040251', 'GRO85051', 'SNA80324'), 373), (('FR040251', 'DAI62779', 'SNA80324'), 359), (('ELE92920', 'SNA18336',
```

حالا تولید قوانین انجمنی برای مجموعه های سه تایی:

برای تولید قوانین انجمنی از مجموعه‌های سه تایی تابع generate_rules را به روزرسانی می‌کنیم تا قوانین را برای مجموعه‌های سه تایی بسازه:

```
def generate_rules(frequent_itemsets, item_counts, previous_frequent_itemsets):
    itemset_support = {itemset: support for itemset, support in frequent_itemsets.items()}
    previous_itemset_support = {itemset: support for itemset, support in previous_frequent_itemsets.items()}
    rules = []

    for itemset, support in frequent_itemsets.items():
        for i in range(1, len(itemset)):
            for antecedent in combinations(itemset, i):
                consequent = tuple(set(itemset) - set(antecedent))
                if consequent:
                    antecedent_support = previous_itemset_support.get(antecedent, item_counts[antecedent[0]])
                    confidence = support / antecedent_support
                    rules.append((antecedent, consequent, support, confidence))

    return rules

rules = generate_rules(frequent_itemsets_3, item_counts, frequent_itemsets_2)
print(f"Rules of size 3: {rules}")
```

در اینجا برای هر مجموعه سه تایی پر تکرار قوانین انجمنی را تولید می‌کنیم به این صورت که برای هر زیرمجموعه (antecedent) از مجموعه اصلی (itemset) مجموعه باقیمانده (consequent) را محاسبه می‌کنیم سپس اعتقاد (confidence) قانون را محاسبه می‌کنیم و قانون را به لیست قوانین اضافه می‌کنیم

```
Rules of size 3: [((('FRO040251',), ('ELE17451', 'DAI62779'), 181, 0.04663746457098686), (('ELE17451',), ('FRO040251', 'DAI62779'), 181, 0.04670967741935484), (('DAI62779',),
```

در نهایت باید قوانینی که اعتقاد بالایی دارن را فیلتر کنیم و پنج قانون برتر را پیدا کنیم: اینجا قوانین با اعتقاد بالاتر از 0985 را فیلتر کردیم و بعدش آونها را بر اساس اعتقاد به صورت نزولی مرتب کردیم اگر اعتقاد دو قانون برابر بود آونها را به صورت الفبایی بر اساس سمت چپ قوانین (antecedent) مرتب کردیم

```
top_rules[:5]
```

```
[('ELE20847', 'FR092469'), ('FR040251',), 101, 1.0),
 ('ELE20847', 'GR085051'), ('FR040251',), 116, 1.0),
 ('ELE92920', 'DAI23334'), ('DAI62779',), 117, 1.0),
 ('GR085051', 'SNA80324'), ('FR040251',), 373, 1.0)]
```

سوال چهار

: pcy قسمت اول

توضیحات کد:

اول همه‌به خوندن داده‌ها

ابتدا باید داده‌ها رو از فایل browsing.txt بخونیم:

```
def read_data(file_path):
    dataset = []
    with open(file_path, 'r') as file:
        for line in file:
            transaction = line.strip().split()
            dataset.append(transaction)
    return dataset

file_path = 'browsing.txt'

dataset = read_data(file_path)
print("Sample data:", dataset[:5])
```

اینجا فایل رو باز می‌کنیم و هر خط رو به یه لیست از آیتم‌ها تبدیل می‌کنیم و در نهایت یه لیست از لیست‌ها داریم که هر کدام نشون‌دهنده یه جلسه مرور کاربره

```
Sample data: [['FR011987', 'ELE17451', 'ELE89019', 'SNA90258', 'GR099222'], ['GR099222', 'GR012298', 'FR012685', 'ELE91550',
```

:Bitmap ایجاد

الگوریتم PCY از hash buckets استفاده می‌کنه تا تعداد جفت‌های آیتم‌ها رو به دست بیاره و بعدش از يه Bitmap برای فیلتر کردن جفت‌های پرتکرار استفاده می‌کنه:

```
def create_bitmap(hash_buckets, support_threshold):
    bitmap = np.zeros(len(hash_buckets))
    for i, count in enumerate(hash_buckets):
        if count >= support_threshold:
            bitmap[i] = 1
    return bitmap
```

اینجا يه بیت‌مپ درست می‌کنيم که برای bucket اگه تعدادش بالای حداقل support باشه مقدارش 1 می‌شه

شناسایی آیتم‌های پرتکرار:

برای شناسایی آیتم‌های پرتکرار از کد زیر استفاده می‌کنيم:

```
def get_frequent_items(dataset, min_support):
    item_counts = defaultdict(int)
    for transaction in dataset:
        for item in transaction:
            item_counts[item] += 1
    return {item for item, count in item_counts.items() if count >= min_support}
```

اینجا تعداد تکرار هر آیتم رو می‌شماریم و آیتم‌هایی که تعدادشون حداقل 100 بار باشه رو نگه می‌داریم (میشد اینجا از الگوریتمی دیگه هم استفاده کرد ولی دلمون نخواست)

:PCY اجرای الگوریتم

حالا الگوریتم PCY رو پیاده‌سازی می‌کنيم:

```

def pcy(dataset, min_support):

    hash_buckets = defaultdict(int)
    for transaction in dataset:
        for pair in itertools.combinations(transaction, 2):
            hash_buckets[hash(pair) % len(dataset)] += 1

    bitmap = create_bitmap(hash_buckets, min_support)

    frequent_items = get_frequent_items(dataset, min_support)
    candidate_pairs = set()
    for transaction in dataset:
        valid_items = [item for item in transaction if item in frequent_items]
        for pair in itertools.combinations(valid_items, 2):
            if bitmap[hash(pair) % len(dataset)] == 1:
                candidate_pairs.add(pair)

    pair_counts = defaultdict(int)
    for transaction in dataset:
        for pair in itertools.combinations(transaction, 2):
            if pair in candidate_pairs:
                pair_counts[pair] += 1

    frequent_pairs = {pair for pair, count in pair_counts.items() if count >= min_support}
    return frequent_pairs

```

اول جفت‌های آیتم‌ها رو تولید می‌کنیم و تعداد هر جفت رو توی hash bucket می‌ریزیم

بعدش بیت‌مپ رو بر اساس تعداد جفت‌ها و حداقل support درست می‌کنیم

بعد آیتم‌های پر تکرار رو شناسایی می‌کنیم

تو مرحله بعد candidate pairs رو پیدا می‌کنیم که توی بیت‌مپ مقدار 1 داشته باشن

در نهایت تعداد واقعی جفت‌های معتبر رو می‌شماریم و جفت‌هایی که تعدادشون حداقل

100 باشه رو نگه می‌داریم

خروجی:

تولید قوانین انجمنی:

و حال قوانین انجمنی رو تولید می‌کنیم:

```

def generate_association_rules(frequent_itemsets, dataset, min_confidence):
    item_counts = defaultdict(int)
    for transaction in dataset:
        for item in transaction:
            item_counts[item] += 1

    rules = []
    for itemset in frequent_itemsets:
        for item in itemset:
            antecedent = {item}
            consequent = set(itemset) - antecedent
            if consequent:
                antecedent_count = sum(1 for transaction in dataset if antecedent.issubset(transaction))
                rule_count = sum(1 for transaction in dataset if antecedent.issubset(transaction) and consequent.issubset(transaction))
                confidence = rule_count / antecedent_count
                if confidence >= min_confidence:
                    rules.append((antecedent, consequent, confidence))
    return rules

```

اینجا برای هر مجموعه پر تکرار قوانین انجمنی رو تولید می کنیم confidence قانون رو محاسبه می کنیم و قوانین با اعتماد بالاتر از ۵۰٪ رو نگه می داریم

Association Rules (PCY):

```

(['DAI55148'], {'DAI62779'}, 0.5394871794871795)
([{'ELE20847'}, {'FRO40251'}], 0.530562347188264)
([{'ELE21353'}, {'DAI62779'}], 0.5023255813953489)
([{'ELE20847'}, {'FRO40251'}], 0.530562347188264)
([{'DAI93865'}, {'FRO40251'}], 1.0)
([{'SNA30859'}, {'GRO24246'}], 0.53125)
([{'FR092469'}, {'FRO40251'}], 0.983510011778563)
([{'GR089004'}, {'ELE25077'}], 0.698051948051948)
([{'FR092469'}, {'FRO40251'}], 0.983510011778563)
([{'ELE92920'}, {'DAI62779'}], 0.7326649958228906)
([{'DAI46755'}, {'FR081176'}], 0.5803921568627451)
([{'SNA30533'}, {'SNA96271'}], 0.5090361445783133)
([{'FR019221'}, {'DAI62779'}], 0.5976714100905562)
([{'FR047962'}, {"DAI75645"}], 0.6176470588235294)
([{'GR085051'}, {"FRO40251"}], 0.999176276771005)
([{'ELE20847'}, {"SNA80324"}], 0.5012224938875306)
([{'DAI43223'}, {"ELE32164"}], 0.5511627906976744)
([{'DAI43223'}, {"ELE32164"}], 0.5511627906976744)
([{'GRO38636'}, {"FRO40251"}], 0.9906542056074766)
([{'GRO85051'}, {"FRO40251"}], 0.999176276771005)
([{'GR081647'}, {"GR073461"}], 0.6775510204081633)
([{'DAI53152'}, {"FRO40251"}], 0.717948717948718)
([{'SNA95666'}, {"FRO85978"}], 0.5758706467661692)
([{'ELE20847'}, {"SNA80324"}], 0.5012224938875306)
...
([{'ELE59028'}, {"DAI62779"}], 0.549777117384844)
([{'SNA18336'}, {"DAI62779"}], 0.7136812411847673)
([{'DAI23334'}, {"DAI62779"}], 0.9545454545454546)
([{'ELE32244'}, {"ELE66600"}], 0.6403508771929824)

```

:Apriori و PCY مقایسه

:Apriori PCY نسبت به

کارایی بهتر با استفاده از bitmap و hash buckets جفت‌های غیر ضروری کاهش پیدا می‌کنند

صرف حافظه کمتر به جای ذخیره تمام جفت‌های ممکن از بیت‌مپ استفاده می‌کنند که حافظه کمتری صرف می‌کنند

:Apriori PCY نسبت به

پیچیدگی بیشتر پیاده‌سازی و درک الگوریتم PCY پیچیده‌تره
حساسیت به کارایی الگوریتم به انتخاب تابع هش و تعداد bucket وابسته‌است
تحلیل ریز:

الگوریتم PCY نسبت به Apriori کارایی بهتری دارد و می‌توانه مجموعه آیتم‌های پرتکرار را با صرف حافظه کمتر پیدا کند اما درک و پیاده‌سازی اون پیچیده‌تره و به انتخاب تابع هش مناسب بستگی دارد

: Toivonen الگوریتم

الگوریتم Toivonen برای پیدا کردن مجموعه آیتم‌های پرتکرار از تکنیک نمونه‌برداری استفاده می‌کند هدف این استفاده از یک نمونه کوچک از داده‌ها، محاسبات اولیه رو انجام بد و بعد این نتایج رو روی کل داده‌ها بررسی کند اگه نیاز بود، دوباره نمونه‌برداری کند تا به نتیجه نهایی برسه

:Sampling

اولین مرحله این است که یک نمونه تصادفی از داده‌ها انتخاب کنیم:

```
def sampling(baskets, probability):
    size = len(baskets)
    sample_size = int(probability * size)
    print(f"Sampling {sample_size} out of {size} baskets...")
    return random.sample(baskets, sample_size)
```

اینجا، یک نمونه به اندازه 10 درصد از کل سبدها انتخاب می‌کنیم فرض کنید 1000 تا سبد داریم، از این تعداد 100 تا رو به صورت تصادفی انتخاب می‌کنیم

اجرای Apriori روی نمونه

حالا الگوریتم Apriori رو روی این نمونه اجرا می‌کنیم:

```
def apriori(baskets, support):
    k = 1
    result = []
    candidates = generate_singletons(baskets)
    while candidates:
        print(f"Generating frequent itemsets of size {k}")
        frequent_items = generate_frequent_itemsets(candidates, baskets, support)
        print('frequent_items', frequent_items)
        if frequent_items:
            result.append(frequent_items)
            candidates = generate_candidates(frequent_items, k + 1)
            k += 1
            if(k>3):
                break
        else:
            break
    return result

def generate_frequent_itemsets(candidates, baskets, support):
    item_counts = defaultdict(int)
    for basket in tqdm(baskets, desc="Generating frequent itemsets"):
        for candidate in candidates:
            if frozenset(candidate).issubset(basket):
                item_counts[frozenset(candidate)] += 1
    return [list(item) for item, count in item_counts.items() if count >= support]

def generate_candidates(frequent_items, k):
    candidates = []
    print(f"Generating candidates of size {k}")
    for i in tqdm(range(len(frequent_items)), desc="Generating candidates"):
        for j in range(i + 1, len(frequent_items)):
            candidate = frozenset(frequent_items[i]) | frozenset(frequent_items[j])
            if len(candidate) == k:
                candidates.append(candidate)
    return [list(candidate) for candidate in candidates]
```

اینجا با تنظیم حداقل support برای نمونه، الگوریتم Apriori رو اجرا می‌کنیم تا آیتم‌های پرتکرار رو پیدا کنیم مثلاً اگه حمایت حداقل 100 باشد، برای نمونه 10 درصدی، حداقل حمایت رو برابر 9 قرار می‌دیم همچنین negative border رو تولید می‌کنیم که شامل

آیتم‌هایی میشے که در نمونه پر تکرار نبودن ولی ممکنه در داده‌های کامل پر تکرار باشن
یعنی همون مجموعه‌هایی که تمامی زیر مجموعه‌هایشون پر تکرارن ولی خودشون نیستن

بررسی negative border در داده‌های کامل

در مرحله دوم، negative border را در کل داده‌ها بررسی می‌کنیم:

```
def check_negative_border_items(negative_border_items, baskets, support):
    for items in tqdm(negative_border_items, desc="Checking negative border items"):
        count = sum(1 for basket in baskets if frozenset(items).issubset(basket))
        if count >= support:
            return True
    return False
```

اینجا چک می‌کنیم که آیا آیتم‌های negative border در داده‌های اصلی support کافی دارن
یا نه اگه حتی یک آیتم مرز منفی support کافی داشته باشه، یعنی نتایج نمونه ما کامل
نیست و باید دوباره نمونه‌برداری و محاسبه کنیم اگر نه، مجموعه آیتم‌های پر تکرار واقعی را
فیلتر می‌کنیم و نگه می‌داریم

تکرار تا رسیدن به نتیجه

این فرآیند ممکنه چندین بار تکرار بشه تا مطمئن بشیم هیچ آیتمی در مرز منفی وجود
نداره که حمایت کافی داشته باشه:

هر بار که نمونه‌برداری و محاسبات رو تکرار می‌کنیم، تعداد iterations رو افزایش می‌دمیم تا
زمانی که به نتیجه نهایی برسیم

تولید قوانین انجمنی

```

def generate_association_rules(frequent_itemsets, baskets, min_confidence=0.5):
    rules = []
    for itemset_group in frequent_itemsets:
        for itemset in itemset_group:
            if len(itemset) > 1:
                subsets = list(combinations(itemset, len(itemset) - 1))
                for subset in subsets:
                    antecedent = frozenset(subset)
                    consequent = frozenset(itemset) - antecedent
                    support_antecedent = calculate_support(antecedent, baskets)
                    support_itemset = calculate_support(frozenset(itemset), baskets)
                    confidence = support_itemset / support_antecedent
                    if confidence >= min_confidence:
                        rules.append((antecedent, consequent, confidence))
    return rules

```

در نهایت، با استفاده از مجموعه آیتم‌های پر تکرار واقعی، قوانین انجمانی را تولید می‌کنیم:

خروجی مراحل مختلف :

```

Iteration 1 starting...
Starting pass one...
Generated 18929 baskets from input file.
Sampling 1892 out of 18929 baskets...
Adjusted support for sample: 9.000000000000002
Generating frequent itemsets of size 1
Generating frequent itemsets: 100%[██████] 1892/1892 [00:01<00:00, 1066.42it/s]
frequent_items[['SNA45677'], ['DAI22896'], ['ELE99737'], ['GRO38814'], ['DAI85309'], ['SNA49107'], ['ELE78169'], ['GRO46854'], ['ELE42696'], ['GR094758'], ['SNA93730'], ['EL
Generating candidates of size 2
Generating candidates: 100%[██████] 451/451 [00:00<00:00, 1520.77it/s]
Generating frequent itemsets of size 2
Generating frequent itemsets: 100%[██████] 1892/1892 [00:48<00:00, 38.61it/s]
frequent_items[['ELE99737'], ['SNA45677'], ['GRO38814'], ['SNA45677'], ['DAI85309'], ['ELE78169'], ['SNA45677'], ['GRO46854'], ['SNA45677'], ['GR094758'], ['SNA45677'], ['GR094758'], [
Generating candidates of size 3
Generating candidates: 100%[██████] 771/771 [00:00<00:00, 3680.16it/s]
Generating frequent itemsets of size 3
Generating frequent itemsets: 100%[██████] 1892/1892 [00:15<00:00, 124.70it/s]
frequent_items[['ELE99737'], ['SNA45677'], ['DAI85309'], ['ELE99737'], ['SNA45677'], ['GR094758'], ['SNA45677'], ['DAI85309'], ['GR046854'], ['SNA45677'], ['GR094758'], ['DAI85309'], ['EL
Generating candidates of size 4
Generating candidates: 100%[██████] 1043/1043 [00:00<00:00, 2175.89it/s]
Frequent itemsets in sample: [[[['SNA45677'], ['DAI22896'], ['ELE99737'], ['GRO38814'], ['DAI85309'], ['SNA49107'], ['ELE78169'], ['GRO46854'], ['ELE42696'], ['GR094758'], [
Generating negative border: 100%[██████] 451/451 [00:00<00:00, 12397.15it/s]
Negative border items: []
Starting pass two...
Generated 18929 baskets from input file.
Checking negative border items: 0it [00:00, ?it/s]
Filtering frequent items: 100%[██████] 451/451 [00:03<00:00, 145.91it/s]
Filtering frequent items: 100%[██████] 771/771 [00:05<00:00, 136.20it/s]
Filtering frequent items: 100%[██████] 1043/1043 [00:08<00:00, 118.74it/s]
Frequent itemsets after pass two: [[[['SNA45677'], ['DAI22896'], ['ELE99737'], ['GRO38814'], ['DAI85309'], ['SNA49107'], ['ELE78169'], ['GRO46854'], ['ELE42696'], ['GR094758'],

```

ایتم های پر تکرار:

```

frequent_itemsets = results[2]
frequent_itemsets

[[['SNA45677'],
  ['DAI22896'],
  ['ELE99737'],
  ['GRO38814'],
  ['DAI85309'],
  ['SNA49107'],
  ['ELE78169'],
  ['GRO46854'],
  ['ELE42696'],
  ['GRO94758'],
  ['SNA93730'],
  ['SNA12663'],
  ['SNA90094'],
  ['FRO40251'],
  ['FR078994'],
  ['DAI75645'],
  ['SNA80324'],
  ['ELE22970'],
  ['ELE85027'],
  ['DAI62779'],
  ['FR094523'],
  ['GRO38983'],
  ['DAI89320'],
  ['SNA96271'],
  ['DAI64292'],
  ...
  ['DAI62779', 'GRO73461', 'DAI85309'],
  ['DAI62779', 'ELE17451', 'GRO59710'],
  ['ELE32164', 'ELE17451', 'DAI43223'],
  ['ELE32164', 'DAI62779', 'ELE17451'],
  ['ELE17451', 'DAI43223', 'GRO59710]]]
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```

قوانين انجمنى :

```

Generated 18929 baskets from input file.

Association Rules with confidence higher than 50%:
{'GRO85051'} -> {'FRO40251'} (confidence: 1.00)
{'ELE59028'} -> {'DAI62779'} (confidence: 0.59)
{'DAI55148'} -> {'DAI62779'} (confidence: 0.57)
{'FRO47962'} -> {'DAI75645'} (confidence: 0.80)
{'FRO19221'} -> {'DAI62779'} (confidence: 0.65)
{'DAI83031'} -> {'DAI94679'} (confidence: 0.52)
{'DAI88079'} -> {'FRO40251'} (confidence: 0.99)
{'DAI43868'} -> {'SNA82528'} (confidence: 0.95)
{'SNA82528'} -> {'DAI43868'} (confidence: 0.51)
{'FRO92469'} -> {'FRO40251'} (confidence: 0.99)
{'DAI93865'} -> {'FRO40251'} (confidence: 1.00)
{'ELE20847'} -> {'FRO40251'} (confidence: 0.53)
{'SNA53220'} -> {'DAI62779'} (confidence: 0.57)
{'SNA53220'} -> {'SNA93860'} (confidence: 0.56)
{'SNA30533'} -> {'SNA96271'} (confidence: 0.50)
{'ELE88583'} -> {'SNA24799'} (confidence: 0.59)
{'DAI43223'} -> {'ELE32164'} (confidence: 0.58)
{'GRO59710'} -> {'ELE32164'} (confidence: 0.51)
{'ELE32244'} -> {'ELE66600'} (confidence: 0.74)
{'GRO81647'} -> {'GRO73461'} (confidence: 0.70)
{'FRO73056'} -> {'GRO44993'} (confidence: 0.58)
{'ELE21353'} -> {'DAI62779'} (confidence: 0.62)
...
{'ELE17451', 'DAI43223'} -> {'ELE32164'} (confidence: 0.67)
{'ELE32164', 'ELE17451'} -> {'DAI62779'} (confidence: 0.74)
{'ELE17451', 'DAI43223'} -> {'GRO59710'} (confidence: 0.53)
{'ELE17451', 'GRO59710'} -> {'DAI43223'} (confidence: 0.60)

```

مقایسه Toivonen با Apriori

مزایا:

کارایی بهتر: با استفاده از نمونهبرداری، حجم داده‌های مورد پردازش کاهش می‌یابد که سرعت الگوریتم را افزایش می‌دهد

کاهش مصرف حافظه: به دلیل پردازش نمونه‌ای از داده‌ها، حافظه کمتری مصرف می‌شود

معایب:

پیچیدگی بیشتر: پیاده‌سازی و درک الگوریتم Toivonen نسبت به Apriori پیچیده‌تر است
نیاز به تنظیم دقیق پارامترها: پارامترهایی مانند اندازه نمونه و حمایت تنظیم شده باید به دقیق تنظیم شوند تا الگوریتم به درستی کار کند

نسبت به pcy این الگوریتم اگر یکبار اجرا بشه باعث می‌شود فقط یکبار به داده‌های اصلی رجموع کنیم اما pcy دو بار مجبور بود

همچنین این الگوریتم false negative و false positive هم نداره که عالیه تحلیل ریز

الگوریتم Toivonen با استفاده از تکنیک نمونهبرداری و بررسی negative border ، بهبودهایی در کارایی و مصرف حافظه نسبت به الگوریتم Apriori داره اما پیچیدگی و نیاز به تنظیم دقیق پارامترها از معایب اون محسوب می‌شه با این حال، این الگوریتم به ما کمک می‌کنه تا مجموعه آیتم‌های پرتکرار و قوانین انجمنی رو به شکلی کاراتر و با مصرف حافظه کمتر پیدا کنیم

: eclat الگوریتم

این الگوریتم بر خلاف Apriori و Toivonen، از روش vertical برای پردازش تراکنش‌ها استفاده می‌کنه

توضیح کد:

تبدیل به فرمت عمودی

```
def vertical_format(transactions):
    vertical = {}
    for tid, transaction in enumerate(transactions):
        for item in transaction:
            if item not in vertical:
                vertical[item] = set()
            vertical[item].add(tid)
    return vertical
```

بعد اینکه داده ها رو خوندیم تراکنشها رو به فرمت عمودی تبدیل می کنیم:

اینجا، هر آیتم رو به کلید دیکشنری تبدیل می کنیم و مقدار هر کلید، مجموعه ای از transaction IDs است که اون آیتم رو شامل میشن سپس آیتمها رو بر اساس تعداد تراکنشها مرتب می کنیم

اجرای الگوریتم Eclat

```
def eclat(prefix, items, min_support, freq_itemsets):
    while items:
        i, itids = items.pop()
        support = len(itids)
        if support >= min_support:
            freq_itemsets.append(prefix + [i])
            suffix = []
            for j, jtids in items:
                intersection = itids & jtids
                if len(intersection) >= min_support:
                    suffix.append((j, intersection))
            eclat(prefix + [i], sorted(suffix, key=lambda item: len(item[1]), reverse=True), min_support, freq_itemsets)
```

الان می رسمیم به بخش اصلی الگوریتم Eclat که به صورت بازگشتی مجموعه های پرتکرار رو پیدا می کنه:

خب:

با آیتم های تکی شروع می کنیم و اگه تعداد تراکنش های شامل اون آیتم بیشتر از حداقل حمایت باشه، اون رو به لیست آیتم های پرتکرار اضافه می کنیم

برای هر آیتم، با بقیه آیتم‌ها ترکیب می‌کنیم و تقاطع مجموعه تراکنش‌ها را پیدا می‌کنیم
اگه تعداد تراکنش‌های تقاطع هم بیشتر از حداقل حمایت باشه، اون ترکیب رو به عنوان
کاندید جدید در نظر می‌گیریم

این کار رو به صورت بازگشتی انجام می‌دیم تا تمام مجموعه‌های پر تکرار پیدا بشن

حالا قوانین انجمنی رو تولید می‌کنیم:

```
[({frozenset({'DAI93865'}), frozenset({'FR040251'}), 1.0},  
 ({frozenset({'DAI43868'}), frozenset({'SNA82528'}), 0.9520958083832335},  
 ({frozenset({'SNA82528'}), frozenset({'DAI43868'}), 0.5112540192926045},  
 ({frozenset({'ELE81534'}), frozenset({'DAI62779'}), 0.611764705882353},  
 ({frozenset({'GR089004'}), frozenset({'ELE25077'}), 0.6613756613756613},  
 ({frozenset({'DAI46755'}), frozenset({'FR081176'}), 0.5863874345549738},  
 ({frozenset({'GR023573'}), frozenset({'DAI62779'}), 0.5252525252525252},  
 ({frozenset({'ELE32244'}), frozenset({'ELE66600'}), 0.7378640776699028},  
 ({frozenset({'FR047962'}), frozenset({'GR073461'}), 0.5555555555555556},  
 ({frozenset({'FR047962'}), frozenset({'DAI75645'}), 0.7962962962962962},  
 ({frozenset({'FR017734'}), frozenset({'ELE28189'}), 0.5701357466063348},  
 ({frozenset({'SNA30533'}), frozenset({'SNA96271'}), 0.5},  
 ({frozenset({'SNA93730'}), frozenset({'DAI62779'}), 0.5947136563876652},  
 ({frozenset({'ELE92920'}), frozenset({'SNA18336'}), 0.5565217391304348},  
 ({frozenset({'ELE92920'}), frozenset({'DAI62779'}), 0.708695652173913},  
 ({frozenset({'GR081647'}), frozenset({'GR073461'}), 0.6958333333333333},  
 ({frozenset({'DAI88079'}), frozenset({'FR040251'}), 0.9925093632958801},  
 ({frozenset({'DAI83031'}), frozenset({'DAI94679'}), 0.5222222222222223},  
 ({frozenset({'ELE59028'}), frozenset({'DAI62779'}), 0.5904059040590406},  
 ({frozenset({'SNA18336'}), frozenset({'DAI62779'}), 0.5583941605839416},  
 ({frozenset({'ELE88583'}), frozenset({'SNA24799'}), 0.5858585858585859},  
 ({frozenset({'ELE21353'}), frozenset({'DAI62779'}), 0.6246056782334386},  
 ({frozenset({'DAI88088'}), frozenset({'ELE38289'}), 0.5062111801242236},  
 ({frozenset({'SNA53220'}), frozenset({'SNA93860'}), 0.5608465608465608},  
 ({frozenset({'SNA53220'}), frozenset({'DAI62779'}), 0.5687830687830687},  
 ...  
 ({frozenset({'DAI55148'}), frozenset({'DAI62779'}), 0.567651632970451},  
 ({frozenset({'DAI43223'}), frozenset({'ELE32164'}), 0.5782208588957056},  
 ({frozenset({'FR019221'}), frozenset({'DAI62779'}), 0.6517341040462429},  
 ({frozenset({'GR059710'}), frozenset({'ELE32164'}), 0.510443864229765},  
 ({frozenset({'GR085051'}), frozenset({'FR040251'}), 1.0})]
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings...](#)

اینجا برای هر مجموعه آیتم پر تکرار که طولش بیشتر از یکه، زیرمجموعه‌های مختلف رو
بررسی می‌کنیم و confidence هر قانون رو محاسبه می‌کنیم قوانینی که اعتماد بالاتر از
50% دارن رو نگه می‌داریم

مزایا و معایب Eclat نسبت به Apriori

مزایا:

کارایی بهتر: به دلیل استفاده از روش عمودی، الگوریتم Eclat می‌توانه سریع‌تر از Apriori عمل کنه، مخصوصاً وقتی که تعداد آیتم‌ها زیاد باشه

صرف حافظه کمتر: به جای ذخیره تمام ترکیب‌های ممکن، فقط ترکیب‌های مورد نیاز رو نگه می‌داره

معایب:

پیچیدگی پیاده‌سازی: پیاده‌سازی Eclat نسبت به Apriori پیچیده‌تره
نیاز به مرتب‌سازی مکرر: مرتب‌سازی و تقاطع مجموعه‌ها می‌توانه زمان بر باشه
تحلیل ریز

الگوریتم Eclat با استفاده از روش عمودی و پردازش بازگشتی، بهبودهایی در کارایی و مصرف حافظه نسبت به Apriori داره این الگوریتم به ما کمک می‌کنه تا مجموعه آیتم‌های پر تکرار و قوانین انجمنی رو به شکلی کاراتر و با صرف حافظه کمتر پیدا کنیم با این حال، پیاده‌سازی و درک اون پیچیده‌تره و نیاز به مرتب‌سازی مکرر داره

الگوریتم : fp growth

توضیح کد :

پیدا کردن آیتم‌های یکتا

```
def find_uniItems(transactions):
    unique_items = []
    for i in transactions:
        for j in i:
            if j not in unique_items:
                unique_items.append(j)
    return unique_items

unique_items = find_uniItems(transactions)
print("Unique_items in the transactional data are:\n{}".format(unique_items))
```

```
Unique_items in the transactional data are:  
['FRO11987', 'ELE17451', 'ELE89019', 'SNA90258', 'GRO99222', 'GR012298', 'FR012685', 'ELE91550', 'SNA11465', 'ELE26  
[<]
```

توی این قسمت، تمام آیتم‌های یکتا رو از تراکنش‌ها پیدا می‌کنیم و توی یه لیست به نام unique_items ذخیره می‌کنیم هر آیتمی که قبلاً توی لیست نبوده رو بهش اضافه می‌کنیم پیدا کردن فرکانس آیتم‌ها

اینجا از Counter استفاده می‌کنیم تا تعداد تکرار هر آیتم رو توی تراکنش‌ها بشمریم و نتایج رو توی یه دیکشنری به نام frequent_item_sets ذخیره کنیم

حذف آیتم‌های غیرپرترکار و مرتب‌سازی

```
min_support = 100  
print("The minimum support of the dataset is: {}".format(min_support))  
  
def remove_infrequent_and_sort(frequent_item_sets, min_support):  
    temp_itemset = frequent_item_sets.copy()  
    for key, values in frequent_item_sets.items():  
        if values < min_support:  
            temp_itemset.pop(key)  
    frequent_item_sets = temp_itemset  
    keys = list(frequent_item_sets.keys())  
    values = list(frequent_item_sets.values())  
    sorted_value_index = np.argsort(values)  
    sorted_value_index = np.flip(sorted_value_index)  
    frequent_item_sets = {keys[i]: values[i] for i in sorted_value_index}  
    return frequent_item_sets  
  
frequent_item_sets = remove_infrequent_and_sort(frequent_item_sets, min_support)  
print("The frequent item sets that satisfy the support count are: {}".format(frequent_item_sets))
```

این قسمت اول آیتم‌هایی که فرکانس‌شون کمتر از min_support هست رو حذف می‌کنه و بعدش آیتم‌های باقی‌مانده رو بر اساس فرکانس مرتب می‌کنه در نهایت، دیکشنری frequent_item_sets فقط آیتم‌های پرترکار و مرتب شده رو نگه می‌داره

خروجی:

```
The minimum support of the dataset is: 100  
The frequent item sets that satisfy the support count are: {'DAI62779': 6667, 'FRO40251': 3881, 'ELE17451': 3875, 'GRO73461': 3602, 'SNA80324': 3044, 'ELE32164': 2851, 'D
```

ساخت مجموعه آیتم‌های مرتب شده

```

def build_ordered_itemset(transactions, frequent_item_sets):
    keys = list(frequent_item_sets.keys())
    temp_transactions = []
    for transaction in transactions:
        temp_items = []
        for item in transaction:
            if item in keys:
                temp_items.append(item)
        temp_transactions.append(temp_items)

    transactions = []
    for temp_transaction in temp_transactions:
        new_transaction = []
        for key in keys:
            if key in temp_transaction:
                new_transaction.append(key)
        transactions.append(new_transaction)

    return transactions

transactions = build_ordered_itemset(transactions, frequent_item_sets)
print(transactions)

Generating Rules:  0%|          | 82/89535 [01:37<29:09:22,  1.19s/it]
[['ELE17451', 'GR099222', 'SNA90258', 'FR011987'], ['ELE17451', 'ELE26917', 'GR099222', 'SNA30755', 'GR012298', 'ELE52966', 'SNA80192', 'SNA11465'], ['ELE17451', 'GR073461']

```

این قسمت تراکنش‌ها را بازسازی می‌کنه تا فقط آیتم‌های پرتکرار و مرتب شده را شامل بشن اینطوری، تراکنش‌ها آماده‌ی استفاده در درخت FP می‌شن

ساخت درخت FP

ابتدا کلاس Node را تعریف می‌کنیم که نودهای درخت را می‌سازه

```

class Node:
    def __init__(self, item, count, parent):
        self.item = item
        self.count = count
        self.parent = parent
        self.children = {}
        self.link = None

    def add_child(self, child):
        if child.item not in self.children:
            self.children[child.item] = child

    def increment_count(self, count):
        self.count += count

```

کلاس Node شامل یه آیتم، تعداد تکرار اون آیتم، والدش، فرزندانش و یه لینک به نود بعدی با همون آیتم هست این لینکها برای conditional trees استفاده می‌شن

ساخت درخت FP

کلاس FPTree یه درخت FP می‌سازه هر تراکنش به صورت یه مسیر توی درخت اضافه می‌شه header_table هم برای نگهداری لینک‌های نودها با همون آیتم استفاده می‌شه

ساخت الگوهای پرتکرار

توی این قسمت، ابتدا یه درخت FP ساخته می‌شه سپس، مجموعه آیتم‌های پرتکرار از درخت استخراج می‌شن توابع mine_frequent_itemsets و generate_frequent_itemsets برای استخراج این الگوها استفاده می‌شن

ایتم‌های پرتکرار :

```
[{(frozenset({'ELE17451'}), 3875), (frozenset({'GR099222'}), 906), (frozenset({'SNA90258'}), 550), (frozenset({'FRO11987'}), 104), (frozenset({'ELE26917'}), 2292), (frozenset({'GRO11987'}), 104), (frozenset({'SNA90258'}), 550), (frozenset({'GR099222'}), 906), (frozenset({'ELE17451'}), 3875)}, {(frozenset({'ELE17451'}), 3875), (frozenset({'GR099222'}), 906), (frozenset({'SNA90258'}), 550), (frozenset({'FRO11987'}), 104), (frozenset({'ELE26917'}), 2292), (frozenset({'GRO11987'}), 104), (frozenset({'SNA90258'}), 550), (frozenset({'GR099222'}), 906), (frozenset({'ELE17451'}), 3875)}, {(frozenset({'ELE17451'}), 3875), (frozenset({'GR099222'}), 906), (frozenset({'SNA90258'}), 550), (frozenset({'FRO11987'}), 104), (frozenset({'ELE26917'}), 2292), (frozenset({'GRO11987'}), 104), (frozenset({'SNA90258'}), 550), (frozenset({'GR099222'}), 906), (frozenset({'ELE17451'}), 3875)}]
```

:FPGrowth نقاط قوت

FPGrowth نسبت به الگوریتم‌هایی مثل Apriori سریع‌تره چون نیاز به تولید کاندیداهای متعدد نداره

ساختار درختی کمک می‌کنه تا عملیات‌های پرهزینه کاهش پیدا کنن

FPTree فضای کمتری نسبت به جدول‌های متعدد Apriori نیاز داره
تراکنش‌ها فشرده می‌شن و فقط آیتم‌های پرتکرار ذخیره می‌شن
این الگوریتم می‌تونه با داده‌های بزرگ بهتر کار کنه چون ساختار درختی بهینه‌ای داره

:FPGrowth نقاط ضعف

پیاده‌سازی FP-Growth نسبت به الگوریتم‌های ساده‌تر مثل Apriori پیچیده‌تره و نیاز به دانش عمیق‌تری از ساختار داده‌ها دارد
در بعضی موارد، ساخت درخت‌های شرطی می‌توانه به اندازه داده‌های اصلی بزرگ باشد
که باعث مشکلات حافظه می‌شود

ترتیب آیتم‌ها توی تراکنش‌ها می‌توانه تأثیر زیادی روی ساختار درخت داشته باشد
انتخاب ترتیب مناسب می‌توانه بهینه‌سازی بیشتری رو به همراه داشته باشد

الگوریتم : opus

توی این الگوریتم، هدف اینه که مجموعه آیتم‌های پر تکرار رو پیدا کنیم برای این کار از یک روش درختی استفاده می‌کنیم که نودهای درخت نماینده مجموعه‌ای از آیتم‌ها هستند و هر نود می‌توانه چندتا فرزند داشته باشد که هر فرزند آیتم جدیدی رو به مجموعه نود والد اضافه می‌کنه

چطور OPUS کار می‌کنه؟

شروع با یک نود خالی:

اول از همه، با یه نود ریشه که وضعیتش خالیه و همه آیتم‌ها به عنوان عملگر فعل داره،
شروع می‌کنیم

تولید نودهای فرزند:

برای هر نود، بچه‌هایش رو با اضافه کردن یکی از عملگرهای فعل به وضعیت فعلی می‌سازیم
مثالاً اگه وضعیت فعلی یه نود باشه $[A]$ و عملگرهای فعلش $[B, C]$ باشن، دو نود فرزند
ساخته می‌شوند: $[A, B]$ و $[A, C]$

بررسی هدف:

بعد از تولید هر نود فرزند، چک می‌کنیم که آیا این نود یه مجموعه آیتم پر تکرار هست یا نه
یعنی تعداد تراکنش‌هایی که شامل همه آیتم‌های وضعیت جدید هستن، باید بیشتر یا
مساوی با min_support باشد
هرس کردن نودها:

اگه یه نود نتونه به حداقل حمایت برسه، اون رو هرس می‌کنیم و عملگر مرتبط با اون رو از
لیست عملگرهای فعل حذف می‌کنیم این کار باعث می‌شه که زمان و حافظه کمتری
صرف بشه و الگوریتم سریع‌تر بشه
تکرار مراحل:

این مراحل رو برای هر نود تکرار می‌کنیم تا زمانی که همه نودها بررسی بشن و نودهای
جدیدی برای تولید باقی نمون کهنه تا صب طول میکشه
نقاط قوت این عزیز:

بهینه‌سازی از طریق Pruning
یکی از بزرگ‌ترین مزایای OPUS، استفاده از هرس کردن نودهایی است که نمی‌توانند به
حداقل حمایت برسند این کار باعث می‌شود که فضای جستجو به صورت قابل توجهی
کاهش یابد و الگوریتم سریع‌تر شود

هرس کردن نودهای بی‌فایده باعث می‌شود که حافظه کمتری استفاده شود و محاسبات
کمتری انجام گیرد

جستجوی موثر:
الگوریتم OPUS از جستجوی درختی استفاده می‌کند که بهینه‌تر از روش‌های دیگر مانند
جستجوی کامل است این روش باعث می‌شود که الگوریتم بتواند به صورت موثرتر و با
سرعت بیشتری مجموعه آیتم‌های پر تکرار را پیدا کند

مقیاس‌پذیری:

این الگوریتم می‌تواند با داده‌های بزرگ‌تر و پیچیده‌تر به خوبی مقیاس شود با افزایش تعداد تراکنش‌ها و آیتم‌ها، الگوریتم همچنان قادر است با استفاده از هرس کردن، کارایی خود را حفظ کند

نقاط ضعف این عزیز

پیچیدگی پیاده‌سازی:

نسبت به الگوریتم‌های ساده‌تری مثل Apriori، پیاده‌سازی OPUS پیچیده‌تر است و نیاز به درک دقیق‌تر و کدنویسی پیچیده‌تری دارد

نیاز به مدیریت دقیق لیست‌های عملگرهای فعال و هرس کردن نودها می‌تواند چالش‌برانگیز باشد

نیاز به حافظه زیاد در برخی مواقع:

در صورتی که تعداد عملگرهای فعال زیاد باشد و هرس کردن به خوبی انجام نشود، ممکن است الگوریتم به حافظه زیادی نیاز داشته باشد

توضیح کد :

کلاس Node

کلاس Node برای تعریف نودهای درخت استفاده می‌شود هر نود شامل state و active و most_recent_operator است

```
class Node:
    def __init__(self, state, active, most_recent_operator=None):
        self.state = state
        self.active = active
        self.most_recent_operator = most_recent_operator

    def __repr__(self):
        return f"Node(state={self.state}, active={self.active}, most_recent_operator={self.most_recent_operator})"
```

: مجموعه آیتم‌هایی که تا اینجا انتخاب شده‌اند state

لیستی از آیتم‌هایی که هنوز می‌توان به مجموعه اضافه بشن: active

آخرین آیتمی که به مجموعه اضافه شده: most_recent_operator

تابع is_goal_state

این تابع بررسی می‌کنه که آیا نود فعلی به min_support می‌رسه یا نه

```
def is_goal_state(state, transactions, min_support):
    count = sum(1 for transaction in transactions if all(item in transaction for item in state))
    return count >= min_support
```

کهههه

وضعیت فعلی نود: state

لیست تراکنش‌ها: transactions

حداقل حمایت مورد نظر: min_support

این تابع تعداد تراکنش‌هایی که شامل همه آیتم‌های وضعیت فعلی هستند را می‌شمره و

اگه این تعداد بیشتر یا مساوی با min_support باشه، True برمی‌گردونه

تابع apply_operator

این تابع یه عملگر (آیتم) رو به وضعیت فعلی اضافه می‌کنه و وضعیت جدید رو برمی‌گردونه

```
def apply_operator(state, operator):
    new_state = state + [operator]
    return new_state
```

: که

وضعیت فعلی: state

آیتمی که می‌خوایم به وضعیت فعلی اضافه کنیم: operator

: وضعیت جدید که شامل آیتم اضافه شده است new_state

تابع generate_children

```
def generate_children(node, transactions, min_support):
    new_nodes = []
    for operator in node.active:
        new_state = apply_operator(node.state, operator)
        new_node = Node(new_state, None, operator)
        if is_goal_state(new_state, transactions, min_support):
            return [new_node], True
        new_nodes.append(new_node)
    return new_nodes, False
```

این تابع برای هر نود، بچههاش را با اضافه کردن عملگرهای فعال به وضعیت فعلی تولید می‌کنه

برای هر عملگر، وضعیت جدید را با اضافه کردن عملگر به وضعیت فعلی تولید می‌کنه و نود جدید رو می‌سازه اگه وضعیت جدید به حداقل حمایت برسه، این نود رو به عنوان نود هدف برمی‌گردونه

تابع can_prune

این تابع بررسی می‌کنه که آیا یه نود می‌تونه هرس بشه یا نه اگه تعداد تراکنش‌هایی که شامل همه آیتم‌های وضعیت فعلی هستند کمتر از min_support باشه، نود هرس می‌شه

تابع prune_nodes

```

def prune_nodes(new_nodes, transactions, min_support, remaining_operators):
    pruned_nodes = []
    for node in new_nodes:
        if can_prune(node, transactions, min_support, remaining_operators):
            remaining_operators.remove(node.most_recent_operator)
        else:
            pruned_nodes.append(node)
    return pruned_nodes

```

نودهایی که به حداقل حمایت نمی‌رسن رو هرس می‌کنه و از لیست عملگرهای فعال حذف می‌کنه

تابع opus

تابع اصلی الگوریتم OPUS که تمامی مراحل رو اجرا می‌کنه

```

def opus(transactions, operators, min_support):
    open_list = [Node([], operators)]
    frequent_itemsets = []
    progress_bar = tqdm(total=len(open_list), desc="Processing Nodes")

    while open_list:
        current_node = open_list.pop(0)
        progress_bar.update(1)
        progress_bar.set_postfix({"Current Node": current_node.state})
        remaining_operators = current_node.active.copy()
        new_nodes, found_goal = generate_children(current_node, transactions, min_support)
        if found_goal:
            frequent_itemsets.append(new_nodes[0].state)
            print(f"\n Appending frequent itemset: {new_nodes[0].state}")
        new_nodes = prune_nodes(new_nodes, transactions, min_support, remaining_operators)
        for node in new_nodes:
            node.active = remaining_operators.copy()
        open_list.extend(new_nodes)
        progress_bar.total = len(open_list) + progress_bar.n # Update total to account for new nodes

    progress_bar.close()
    return frequent_itemsets

```

با یه نود ریشه که وضعیتش خالیه و همه عملگرها رو داره شروع می‌کنیم توی هر تکرار، نود فعلی رو از لیست باز حذف می‌کنیم و بچههای جدید رو تولید می‌کنیم نودهای جدید رو هرس می‌کنیم و نودهایی که به حداقل حمایت نمی‌رسن رو نگه می‌داریم

نودهای باقی‌مانده رو به لیست باز اضافه می‌کنیم و این کار رو تا وقتی که همه نودها بررسی بشن، تکرار می‌کنیم

```
Processing Nodes: 0% | 0/1 [00:00<?, ?it/s]
Processing Nodes: 100% | 1/1 [00:00<00:00, 478.86it/s, Current Node=[]]
Processing Nodes: 100% | 2/2 [00:01<00:00, 1.87it/s, Current Node=[]]
Processing Nodes: 100% | 2/2 [00:01<00:00, 1.87it/s, Current Node=['FR043226']]

Appending frequent itemset: ['FR043226']

Processing Nodes: 100% | 3/3 [00:02<00:00, 1.38it/s, Current Node=['FR043226']]
Processing Nodes: 100% | 3/3 [00:02<00:00, 1.38it/s, Current Node=['FR043226', 'FR043226']]

Appending frequent itemset: ['FR043226', 'FR043226']

Processing Nodes: 100% | 4/4 [00:02<00:00, 1.26it/s, Current Node=['FR043226', 'FR043226']]
Processing Nodes: 100% | 4/4 [00:02<00:00, 1.26it/s, Current Node=['FR043226', 'FR043226', 'FR043226']]

Appending frequent itemset: ['FR043226', 'FR043226', 'FR043226']

Processing Nodes: 100% | 5/5 [00:03<00:00, 1.18it/s, Current Node=['FR043226', 'FR043226', 'FR043226']]
Processing Nodes: 100% | 5/5 [00:03<00:00, 1.18it/s, Current Node=['FR043226', 'FR043226', 'FR043226', 'FR043226']]

Appending frequent itemset: ['FR043226', 'FR043226', 'FR043226', 'FR043226']

Processing Nodes: 100% | 6/6 [00:04<00:00, 1.14it/s, Current Node=['FR043226', 'FR043226', 'FR043226', 'FR043226']]
Processing Nodes: 100% | 6/6 [00:04<00:00, 1.14it/s, Current Node=['FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226']]

Appending frequent itemset: ['FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226']

Processing Nodes: 100% | 7/7 [00:05<00:00, 1.12it/s, Current Node=['FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226']]
Processing Nodes: 100% | 7/7 [00:05<00:00, 1.12it/s, Current Node=['FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226', 'FR043226']]
```