محمد امین رامی

۹۸۱۰۱۵۸۸

پروژه نهایی درس تحلیل داده های حجیم

دکتر ایمان غلامپور

پاییز ۱۴۰۱

رسم هیستوگرام تعداد ترددها در مسیر دوربین های مجاور

در این مسئله از ما خواسته شده است که هیستوگرام تعداد ترددها در مسیر دوربین های مجاور را رسم کنیم. به این منظور نیاز است تا ابتدا دوربین های مجاور را شناسایی کنیم. الگوریتم به دست آوردن دوربین های مجاور به شرح زیر است:

دیتای ما شامل سطر هایی است که حاوی اطلاعات پلاک ماشین و آیدی دوربین میباشد. حال فرض کنید یک ماشین در سطح شهر در حال تردد است. این ماشین توسط دوربین های مختلف detect میشود و دیتای آن ثبت میشود. حال توجه کنید که اگر توالی زمانی را درنظر بگیریم، هر ماشین در حال حرکت، از دوربین های مجاور، پشت سر هم رد میشود.

به عبارت ساده تر، اگر دیتای ذخیره شده هر ماشین را بر اساس پلاک ماشین groupby کنیم و سپس دیتای را بر اساس تاریخ ثبت sort بکنیم، دیتای دوربین های مجاور پشت سر هم قرار میگیرند. پس برای پیدا کردن دوربین های مجاور به صورت زیر عمل میکنیم:

۱- دیتا را بر اساس هر ماشین (پلاک ماشین) groupby کنید.
۲- در هر گروه حاصل از groupby، دیتای ذخیره شده هر ماشین را بر اساس تاریخ ثبت آن sort کنید.
۳- در لیست sort شده هر ماشین، هر دو دوربینی که پشت سر هم قرار گرفته اند(توالی زمانی دارند) را به عنوان یک جفت دوربین مجاور در نظر بگیرید.
۴- اشتراک این دوربین ها را محاسبه کنید و آنهارا به عنوان جفت دوربین های مجاور معرفی کنید.

Group data by car

In [5]:
```
grouped_data = raw_data_rdd.groupBy(lambda x: x[1]).mapValues(list)
```

In [6]:
```
def create_adjucent_camera(row):
    records = row[1]
    records = [(x[0], x[2]) for x in records]
    records = [(x[0], datetime.datetime.strptime(x[1], '%Y-%m-%d %H:%M:%S')) for x in records]
    records.sort(key=lambda x: x[1])   # sort by time
    adjucents = []
    if len(records) > 1:
        for i in range(len(records)-1):
            if len(set(tuple(sorted([records[i][0], records[i+1][0]])))) != 1:
                adjucents.append(tuple(sorted([records[i][0], records[i+1][0]])))
    return list(set(adjucents))
```

قطعه کد بالا، الگوریتم یاد شده را روی دیتا انجام میدهد.

بعد از groupby کردن، بر اساس تاریخ ثبت sort میکند و از لیست دیتای هر ماشین، دوربین های مجاور را استخراج میکند.

حال ما تعداد جفت دوربین های مجاور را میتوانیم داشته باشیم:

```
Finding adjucent cameras

In [7]: adjucent_cameras = grouped_data.flatMap(create_adjucent_camera).distinct()

In [8]: adjucent_count = adjucent_cameras.count()
        print(f'number of adjucent cameras: {adjucent_count}')

        [Stage 4:==================================================>     (50 + 7) / 57]

        number of adjucent cameras: 27927
```

همانطور که مشاهده میشود، به تعداد ۲۷۹۲۷ جفت دوربین مجاور پیدا شده است. توجه کنید که با تاوجه به اینکه حدود ۹۰۰ دوربین متمایز داریم، این عدد عدد معقولی است.

حال برای تست کد خود، تعدادی از این دوربین های مجاور را چاپ میکنیم:

```
In [9]: print('some examples of adjucent cameras:')
        for pair in adjucent_cameras.top(10):
            print(f'{pair[0]} and {pair[1]} are adjucent')

        some examples of adjucent cameras:

        [Stage 7:==================================================>    (52 + 5) / 57]

        900277 and 900278 are adjucent
        900275 and 900278 are adjucent
        900275 and 900277 are adjucent
        900274 and 900278 are adjucent
        900273 and 900277 are adjucent
        900273 and 900274 are adjucent
        900272 and 900277 are adjucent
        900272 and 900273 are adjucent
        900269 and 900278 are adjucent
        900269 and 900277 are adjucent
```

همانطور که مشاهده میشود دوربین های ذکر شده در بالا به عنوان دوربین مجاور معرفی شده اند.

حال تعداد ترددهارا محاسبه میکنیم.

الگوریتم محاسبه تعداد ترددها نیز مشابه دوربین های مجاور است فقط تعدادی گام اضافه دارد. این الگوریتم به شرح زیر است:

۱- دیتا را بر اساس هر ماشین (پلاک ماشین) groupby کنید.

۲- در هر گروه حاصل از groupby، دیتای ذخیره شده هر ماشین را بر اساس تاریخ ثبت آن sort کنید.

۳- در لیست sort شده هر ماشین، هر دو دوربینی که پشت سر هم قرار گرفته اند(توالی زمانی دارند) را به عنوان یک جفت دوربین مجاور در نظر بگیرید.

۴- حال به ازای هر جفت دور بین مجاور، یک key/value به صورت زیر را emit میکنیم:

<key = camera_pair, value = 1>

۵- داده هایی که توسط بخش قبل emit میشوند را بر اساس key آنها reduce کرده و جمع بزنید.

۶- باانجام مرحله ۵، در نهایت یک لیست خواهیم داشت که شامل تعداد ترددها در هر جفت دوربین مجاور (camera_pair) خواهد بود که همان چیزی است که میخواستیم. در حقیقت خواهیم داشت:

< key = camera_pair, value = number_of_passings>

Counting the number of car passings between adjucent cameras ¶

In [10]:
```
def count_passing(row):
    records = row[1]
    records = [(x[0], x[2]) for x in records]
    records = [(x[0], datetime.datetime.strptime(x[1], '%Y-%m-%d %H:%M:%S')) for x in records]
    records.sort(key=lambda x: x[1])   # sort by time
    counts = []
    if len(records) > 1:
        for i in range(len(records)-1):
            if len(set(tuple(sorted([records[i][0], records[i+1][0]])))) != 1:
                r1 = records[i]
                r2 = records[i+1]
                if (r2[1] - r1[1]) > datetime.timedelta(minutes=15):
                    counts.append((tuple(sorted([r1[0], r2[0]])), 1))
    return counts
```

قطعه کد بالا مراحل گفته شده را پیاده سازی میکند.

حال یک سمپل از لیست جفت دوربین های مجاور را انتخاب میکنیم و تعداد ترددهای آنهارا به دست می آوریم و سپس هیستوگرام تعداد تردد هارا رسم میکنیم.

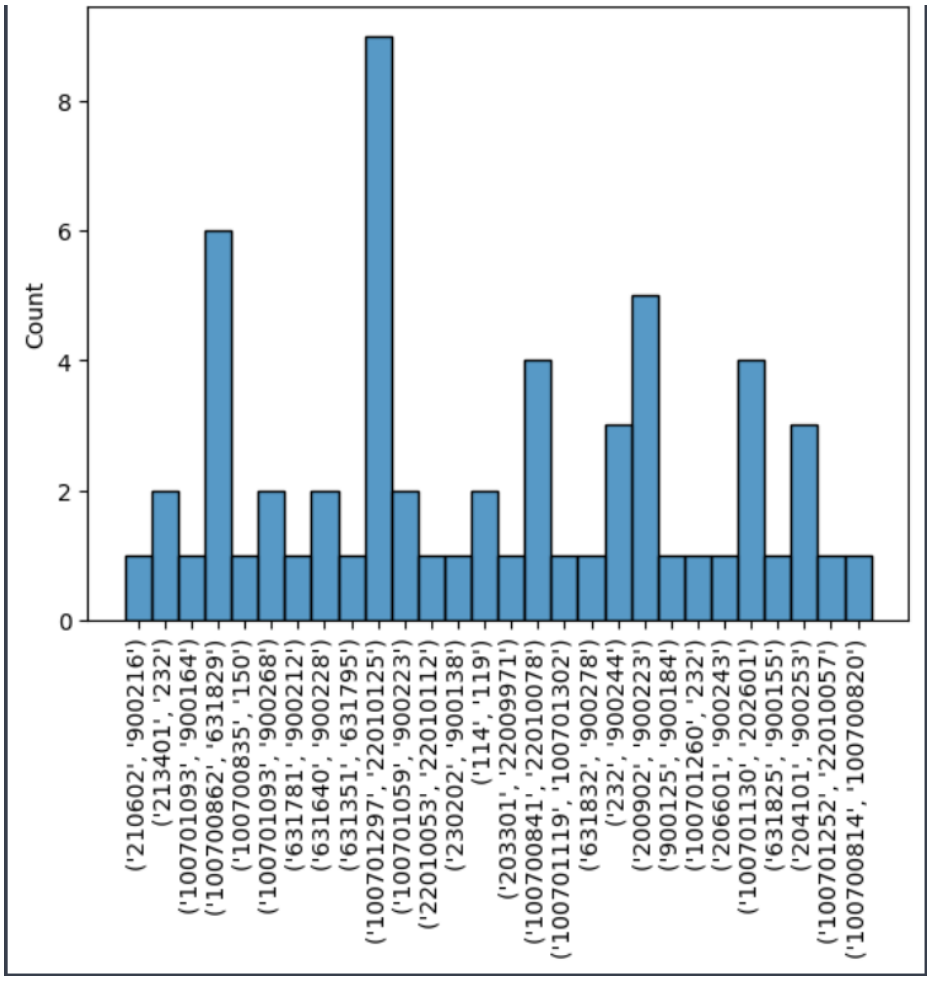
```
In [11]: passing_counts = grouped_data.flatMap(count_passing).reduceByKey(add)

In [15]: passings = passing_counts.sample(fraction=1.4e-3, withReplacement=False).collect()
```

Drawing the histogram

```
In [16]: vis_data = []
         for data in passings:
             vis_data.extend([str(data[0]) for _ in range(data[1])])
```

هستوگرام حاصل شده:



لیبل های هیستوگرام، کدهای جفت دوربین های مجاور هستند و ارتفاع میله ها نیز تعداد تردد است.

جفت دوربین ها نیز به صورت رندوم انتخاب شده اند.

یافتن مسیرهای مکرر با استفاده از frequent-items و مدل item-basket

در این مسئله قصد داریم تا مسیرهای پرتردد را بیابیم. برای این منظور از مدل frequent-items و item-basket استفاده میکنیم.

ابتدا باید مسئله به طرز مناسبی به یک مسئله frequent-items تبدیل کنیم. برای اینکار مدل زیر را نظر میگیریم:

هر ماشین را یک basket و هر دوربین را یک item درنظر میگیریم. در این صورت دوربین هایی که هر ماشین visit کرده است را به عنوان یک item در داخل basket متناظر آن ماشین قرار میدهیم. در این صورت تعدادی basket (ماشین)خواهیم داشت که در آنها تعدادی item (دوربین) قرار دارند.

توابع زیر مراحل بالا را انجام میدهند:

```
In [5]:
# deleting outliers
data_df = data_df.filter(data_df['ORIGINE_CAR_KEY'] == data_df['FINAL_CAR_KEY'])
data_df = data_df.select(['DEVICE_CODE', 'ORIGINE_CAR_KEY', 'PASS_DAY_TIME'])
data = data_df.rdd

In [6]:
# a function to generate the required rdd
def create_key_value(row):
    key = (row['ORIGINE_CAR_KEY'], row['PASS_DAY_TIME'])
    value = row['DEVICE_CODE']
    return (key, value)
```

تابع create_key_value، پلاک ماشین و تاریخ ثبت آن را به عنوان کلید و کد دوربین را به عنوان مقدار درنظر میگیرد و یک tuple را emit میکند. جلوتر این tuple ها را reduceByKey میکنیم تا سبد های ما تکمیل شوند:

```
In [8]:
# cleaning the data: for some cameras, a car had been submited mutiple time in a single moment
# which is not rational. so, we will remove this redundancy
def day_time(basket):
    plate = basket[0][0]
    day_time = basket[0][1]
    camera_code = basket[1]
    day, time = day_time.split()
    time = time[:-3]
    return ((plate, day + " " + time), camera_code)

data_modified = data.map(day_time)
clean_data = data_modified.groupByKey().mapValues(lambda x: tuple(set(tuple(x))))

In [9]:
# removing time, leaving only day
def day(basket):
    plate = basket[0][0]
    day_time = basket[0][1]
    camera_codes = basket[1]
    day = day_time.split()[0]
    return ((plate, day), camera_codes)

def flatten(x):
    if isinstance(x, str):
        yield x
    else:
        for item in x:
            yield from flatten(item)

clean_data = clean_data.map(day)
clean_data = clean_data.groupByKey().mapValues(tuple).mapValues(lambda x: tuple(flatten(x)))
camera_baskets = clean_data.values()
```

حال که سبد ها و آیتم های ما آماده هستند، میتوانیم از الگوریتم های frequent-items استفاده کنیم.

ادعا میکنیم که frequent-item های ما، همان مسیر های پر تردد هستند. درک آن ساده است، زیرا بیان میکند که مثلا یک جفت دوربین وجود دارد که دیتای بسیاری از ماشین ها ثبت شده است.

برای پیدا کردن frequent-items ها از دو روش استفاده میکنیم:

روش اول: الگوریتم A-Priori

همانطور که در کلاس گفته شد، الگوریتم A-Priori به صورت زیر است:

۱- ابتدا دوربین های frequent را پیدا میکنیم. تعریف میکنیم یک دوربین frequent است اگر به تنهایی بیشتر از مقدار مشخصی توسط ماشین ها visit شده باشد. این مقدار مشخص را نیز برابر میانگین تعداد visit های هر دوربین به علاوه 0.4 انحراف معیار آنها در نظر میگیریم:

```
SUPPORT_S =  mean_camera_count +  0.4 * std_camera_count
```

۲- حال لیست دوربین های frequent را با معیار ذکر شده داریم. حال تمام دوتایی های ممکن از این لیست را تولید میکنیم و به عنوان کاندید برای frequent آیتم های دوتایی در نظر میگیریم.

۳- حال که کاندیدهای frequent آیتم های دوتایی را داریم، در درون سبد ها میگردیم و تعداد آنهارا میشماریم. سپس بررسی میکنیم که آیا تعداد آنها از مقدار آستانه ای که در بخش اول مشخص کردیم بیشتر است یا خیر.

۴- دوتایی هایی که تعدادشان از آستانه بیشتر است را به عنوان دوتایی های frequent معرفی میکنیم.

توجه کنید اگر دوربینی به تنهایی frequent نباشد، نمیتواند به صئرت دوتایی نیز در لیست دوتایی های frequent قرار داشته باشد. ما از این موضوع برای تولید کاندیدهای خود استفاده کردیم.

```
In [10]:
# counting the visits to each camera
def extract_cameras(basket):
    items = tuple((camera, 1) for camera in basket)
    return items

camera_count = camera_baskets.flatMap(extract_cameras).reduceByKey(add)
```

```
In [11]:
# calculating support threshold. we set it equal to mean + 0.4 * std
mean_camera_count = camera_count.values().mean()
std_camera_count = camera_count.values().stdev()
SUPPORT_S =  mean_camera_count +  0.4 * std_camera_count
print(f'mean camera count:    {mean_camera_count:.2f}')
print(f'std of camera count:  {std_camera_count:.2f}')
print(f'support threshold:    {SUPPORT_S:0.2f}')
```

```
[Stage 10:=====================================================>(56 + 1) / 57]


mean camera count:    3858.59
std of camera count:  8506.71
support threshold:    7261.27
```

کد بالا تعداد visit های هر دوربین را به تنهایی حساب میکند. همچنین میانگین و انحراف معیار تعداد visitهای دوربین هارا برای محاسبه آستانه حساب میکند.

```
In [14]:
# remove cameras which are not frequents
def remove_infrequent_cameras(basket):
    return tuple(item for item in basket if item in frequent_cameras_list.value)

baskets_with_frequent_cameras = camera_baskets.map(remove_infrequent_cameras)
baskets_with_frequent_cameras = baskets_with_frequent_cameras.filter(lambda x: len(x)<350)
```

```
In [15]:
# creating two tuples
def create_two_tuple(basket):
    n = len(basket)
    two_tuples = []

    for i in range(n):
        for j in range(i+1, n):
            two_tuple = tuple(sorted([basket[i], basket[j]]))
            if len(set(two_tuple)) == 2:
                two_tuples.append((two_tuple, 1))
    return two_tuples

frequent_two_tuples = baskets_with_frequent_cameras.flatMap(create_two_tuple).reduceByKey(add).filter(lambd
```

در کد بالا ما ابتدا آیتم هایی که مکرر نیستند را از سبد ها حذف میکنیم و سپس از آیتم های باقی مانده دوتایی میسازیم به عنوان کاندید.

پس از شمارش دوتایی ها و بررسی کاندیدها، مسیرهای پر تردد به صورت زیر در می آیند:

```
In [16]:
# results
top_two_tuples = frequent_two_tuples.sortBy(lambda x: x[1], ascending=False).take(10)
print(f'number of two tuples is: {frequent_two_tuples.count()}')
print('====================================')
print('===== top frequent two tuples =====')
for two_tuple in top_two_tuples:
    print(two_tuple)
```

```
[Stage 41:==================================================>      (49 + 8) / 57]


number of two tuples is: 33
====================================
===== top frequent two tuples =====
(('22010039', '22010061'), 42802)
(('22010040', '22010061'), 38219)
(('22010053', '22010061'), 33258)
(('22010047', '22010061'), 29518)
(('22010039', '22010040'), 29389)
(('22010039', '22010053'), 24383)
(('22010044', '22010061'), 22361)
(('22010040', '22010053'), 22024)
(('22010039', '22010047'), 21202)
(('22010048', '22010061'), 20349)
```

همانطور که مشاهده میشود، ۳۳ مسیر پرتردد یافت شده است که ۱۰ تا از پرتردد ترین آنها در بالا ذکر شده اند.

Tuple های نمایش داده شده، حاوی کد دوربین ها میباشد.

برای مسیرهای سه دوربینه نیز نتایج زیر حاصل شده است:

```
number of three tuples is: 267
================================
===== top frequent three tuples =====
(('22010039', '22010040', '22010061'), 1312097)
(('22010039', '22010053', '22010061'), 1027128)
(('22010040', '22010053', '22010061'), 950212)
(('22010039', '22010047', '22010061'), 917659)
(('22010040', '22010047', '22010061'), 854912)
(('22010039', '22010040', '22010053'), 772502)
(('22010047', '22010053', '22010061'), 760622)
(('22010039', '22010044', '22010061'), 714274)
(('22010039', '22010040', '22010047'), 672416)
(('22010039', '22010048', '22010061'), 666648)
```

۲۶۷ مسیر پر تردد سه دوربینه یافت شده است.

روش دوم: الگوریتم SON

الگوریتم SON نیز به صورت زیر است:

۱- ابتدا دیتا را به سه دسته تقسیم میکنیم. اینکار را میشود با استفاده یک hash مناسب انجام داد.

۲- سپس برای هر کدام از این دسته ها، frequent آیتم هارا با استفاده از A-Priori پیدا میکنیم و به عنوان کاندید معرفی میکنیم.

۳- از کاندیدا های هر سه دسته اجتماع میگیریم.

۴- سپس تعداد این کاندید هارا در دیتای اصلی میشماریم و چک میکنیم که آیا تعداد آنها از آستانه بیشتر است یا خیر

۵- کاندیدهایی که تعدادی بیشتر از آستانه دارند را به عنوان frequent آیتم معرفی میکنیم.

خروجی که از الگوریتم SON میگیریم به صورت زیر است که دقیقا مانند الگوریتم A-Priori میباشد:

```
In [19]:
# results
top_two_tuples = son_final_frequent_two_tuples.sortBy(lambda x: x[1], ascending=False).take(10)
print(f'number of two tuples is: {son_final_frequent_two_tuples.count()}')
print('==================================')
print('===== top frequent two tuples =====')
for two_tuple in top_two_tuples:
    print(two_tuple)
```

```
number of two tuples is: 33
==================================
===== top frequent two tuples =====
(('22010039', '22010061'), 42802)
(('22010040', '22010061'), 38219)
(('22010053', '22010061'), 33258)
(('22010047', '22010061'), 29518)
(('22010039', '22010040'), 29389)
(('22010039', '22010053'), 24383)
(('22010044', '22010061'), 22361)
(('22010040', '22010053'), 22024)
(('22010039', '22010047'), 21202)
(('22010048', '22010061'), 20349)
```

مسیرهای پرتردد دو دوربینه که توسط الگوریتم SON به دست آمده اند.

```python
top_three_tuples = son_final_frequent_three_tuples.sortBy(lambda x: x[1], ascending=False).take(10)
print(f'number of three tuples is: {son_final_frequent_three_tuples.count()}')
print('================================')
print('===== top frequent three tuples =====')
for three_tuple in top_three_tuples:
    print(three_tuple)
```

```
number of three tuples is: 267
================================
===== top frequent three tuples =====
(('22010039', '22010040', '22010061'), 1312097)
(('22010039', '22010053', '22010061'), 1027128)
(('22010040', '22010053', '22010061'), 950212)
(('22010039', '22010047', '22010061'), 917659)
(('22010040', '22010047', '22010061'), 854912)
(('22010039', '22010040', '22010053'), 772502)
(('22010047', '22010053', '22010061'), 760622)
(('22010039', '22010044', '22010061'), 714274)
(('22010039', '22010040', '22010047'), 672416)
(('22010039', '22010048', '22010061'), 666648)
```

مسیرهای پرتردد سه مسیر که توسط الگوریتم SON به دست آمده است.

یافتن دوربین های مشابه با استفاده از Collaborative Filtering

در این مسئله ما میخواهیم برای دیتایی که در دست داریم، ابتدا latent variable هارا پیدا کنیم و سپس با استفاده از آن، روی دیتا collaborative filtering انجام دهیم. ابتدا نیاز است تا ماتریس مناسب را برای دیتا تشکیل دهیم.

ماتریس ما به این صورت است که سطرهای آن دوربین های ما هستند و ستون ها آن ساعات روز در روزهای مختف هستند و مقدار هر درایه نیز تعداد ثبتی های هر دوربین در آن روز و ساعت خاص میباشد.

ابتدا مانند کد زیر، key هایی بر اساس دوربین و روز و ساعت ثبت میسازیم و همچنین به هر دوربین یک index نسبت میدهیم.

```
In [5]:
# find day and hour
def to_day_hour(row):
    camera_code, date_time = row
    date_time = datetime.strptime(date_time, '%Y-%m-%d %H:%M:%S')
    return (camera_code, (date_time.weekday(), date_time.hour))

In [6]:
data_rdd = raw_data_rdd.map(to_day_hour)
camera_list = data_rdd.map(lambda x: x[0]).distinct().collect()
camera2index = {camera: index for index, camera in enumerate(camera_list)}
index2camera = {index: camera for index, camera in enumerate(camera_list)}
```

حال طبق کد زیر ابتدا تعداد ثبتی هارا با استفاده از key ها و یک عملیات reduce میشماریم و با استفاده از index هر دربین، ماتریس یادشده را تشکیل میدهیم.

```
In [7]:
def count(row):
    return (row, 1)

data_values = data_rdd.map(count)

In [8]:
M = np.zeros((len(camera_list), 24*7))

In [9]:
records = data_values.reduceByKey(add).collect()

for record in records:
    key, value = record
    camera = key[0]
    day, hour = key[1]
    M[camera2index[camera], 24*day + hour] = value

mean = M - np.mean(M, axis=1).reshape((-1, 1))
M = M - mean
```

حال بخشی از ماتریس را به عنوان دیتای train برای collaborative filter و باقی آن را برای test فیلتر خود در نظر میگیریم و سپس فیلتر را آموزش میدهیم.

به یاد داریم که بردار p مربوط به بردار یژگی های پنهان دوربین ها، و بردار q بردار ویژگی های پنهان ساعات روز است.

به عنوان hyperparameter مسئله، بعد فضای پنهان (latent space) خود را برابر ۳ درنظر میگیرم.

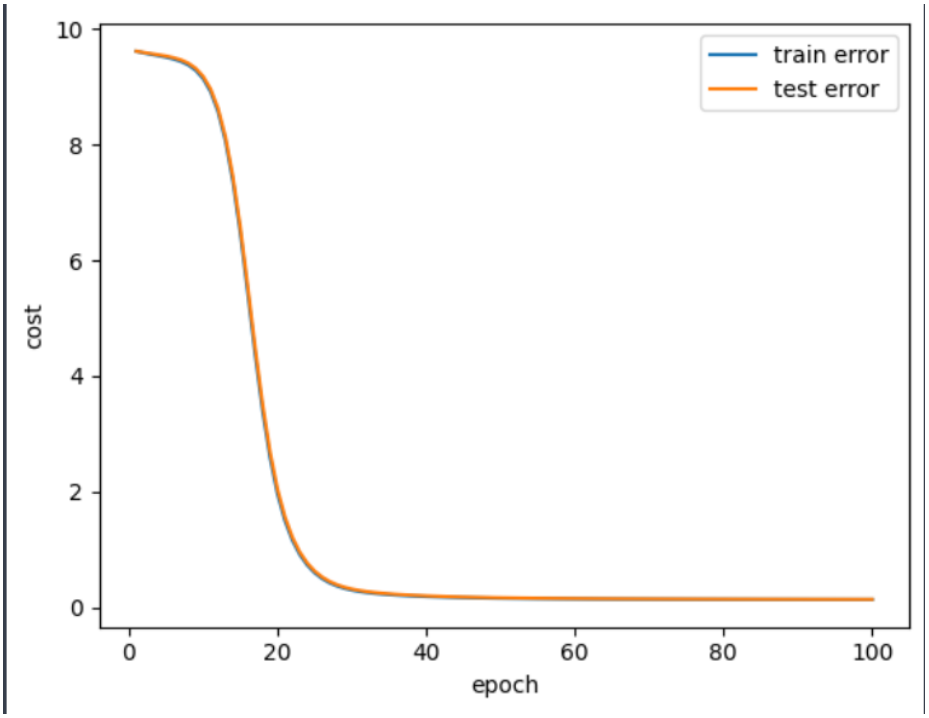
تابع هزینه ما به صورت زیر است:

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - q_i^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

که منظور از r ، مقدار واقعی ماتریس اصلی میباشد. قاعده آپدیت نیر باتوجه به تابع هزینه به صورت زیر است:

- $q_i \leftarrow q_i + \gamma \cdot (e_{ui} \cdot p_u - \lambda \cdot q_i)$
- $p_u \leftarrow p_u + \gamma \cdot (e_{ui} \cdot q_i - \lambda \cdot p_u)$

حال فیلتر را آموزش میدهیم و loss آن را در حین آموزش رسم میکنیم:



مشاهده میشود که تابع هزینه در هر مرحله آموزش، کاهش میابد و در آخر نیز تا حد بسیار مطلوبی کم شده است.

یافتن دوربین های مشابه با استفاده از Collaborative Filtering

هم اکنون که فیلتر ما آموزش داده شده است، این توانایی را دارد که هر دوربین را به سه متغییر پنهان map کند. یعنی این توانایی را دارد که دوربین های مارا توسط سه ویژگی پنهان که یادگیری شده اند، توضیف کند. حال که دوربین ها به فضای کوچک تری map شده اند، دوربین هایی که latent variable هایی مشابهی دارنده را به عنوان دوربین های مشابه معرفی میکنیم.

حال برای مثال یک دوربین رندوم انتخا کرده و با ضرب داخلی بردار متغییر های مخفی این دوربین با بردار متغییر های مخفی بقیه دوربین ها و ماکسیمم گیری از مقادیر حاصل، دوربین های مشابه آن دوربین را پیدا میکنیم:

```
In [16]:
query_index = random.choice(range(len(camera_list)))
query_latent_vector = cf.P[:, [query_index]]

similarities = query_latent_vector.T @ cf.P
indices = list(np.argsort(similarities.reshape((-1, )), )[::-1][:10])
```

```
In [17]:
query_code = index2camera[query_index]
print(f'query camera code:  {query_code}')
print('------- Top 10 similar cameras -------')
for i, index in enumerate(indices):
    print(f'{i+1} - ')
    print(f'camera code:    {index2camera[index]}')
    print(f'similarity:      {similarities[0, index]:0.2f}')
```

```
query camera code:  801510
------- Top 10 similar cameras -------
1 -
camera code:    100700853
similarity:      3.59
2 -
camera code:    631353
similarity:      3.08
3 -
camera code:    900225
similarity:      2.93
4 -
camera code:    900212
similarity:      2.66
5 -
camera code:    900269
similarity:      2.44
6 -
camera code:    900273
similarity:      2.41
7 -
camera code:    200101
similarity:      2.38
8 -
camera code:    900223
similarity:      2.37
9 -
camera code:    631634
similarity:      2.32
10 -
camera code:    10015402
similarity:      2.28
```

در کد بالا یک دوربین را به صورت رندوم انتخاب کردیم و سپس دوربین هایی که ضرب داخلی بردار ویژگی های پنهان آنها با این دوربین، بیشتر مقدار را دارد، به عنوان دوربین مشابه معرفی کردیم:

یافتن دوربین های مشابه با استفاده از Pixie

در این مسئله ما قصد داریم تا دوربین های مشابه با یک دوربین خاص را در بین بقیه دوربین ها پیدا کنیم.

برای این منظور از الگوریتم Pixie استفاده میکنیم. این الگوریتم را به صورت زیر پیاده سازی کرده ایم:

۱- ابتدا باید دیتا را به صورت یک گراف bi-partite دربیاوریم. یک طرف این گراف دوربین ها و در طرف دیگر، ماشین ها قرار دارند. یک دوربین به یک ماشین وصل است اگر حداقل یک بار آن ماشین توسط آن دوربین دیده شده باشد.

۲- گراف دیتای را تشکیل میدهیم. یعنی به ازای هر نود، همسایه های آن را پیدا میکنیم و آنها را به عنوان همسایه ذخیره میکنیم.

۳- از نود query شده به عنوان نود آغازین شروع میکنیم.

۴- سپس یک همسایه رندوم از این نود انتخاب کرده (که از نوع ماشین است) و به آن میرویم. حال دوباره در نود جدید یکبار دیگر یک همسایه رندوم انتخاب میکنیم(که اینبار از نوع دوربین است).

۵- سپس در نود جدید، یک عدد به تعداد دفعات visit شدن آن اضافه میکنیم.

۶- با یک احتمال مشخص، به صورت تصافی به مرحله ۳ میرویم.

۷- به مرحله ۴ میرویم. اینکار را (که عملا random walk است) به تعداد کافی انجام میدهیم تا الگوریتم خاطمه بیابد.

۸- سپس نود هایی که بیشترین دفعات visit شدن را دارند به عنوان نود های شبیه معرفی میکنیم.

برای پیاده سازی الگوریتم فوق، نیاز داریم تا یک کلاس پایتونی برای نود پیاده سازی کنیم که امکان ذخیره سازی همسایه هارا داشته باشد. کلاس پایتونی زیر امکانات مورد نیاز برای الگوریتم فوق را فراهم میکند:

```
In [5]: # Node class for creating our bipartite graph

class Node:
    def __init__(self, code):
        self.neighbours = []
        self.count = 0
        self.code = code

    def add_neighbour(self, neighbour):
        self.neighbours.append(neighbour)

    def get_neighbours(self):
        return self.neighbours

    def add_count(self):
        self.count = self.count + 1

    def get_random_neighbour(self):
        return random.choice(self.get_neighbours())
```

طبق مراحل گفته شده، دیتای داده شده به یکی یکی میخوانیم و به ازای هر دوربین و ماشینی که میخوانیم یک یال وصل میکنیم:

Constructing the grapgh

```
In [8]: data_list = data_rdd.collect()
        for record in tqdm(data_list):
            camera_name = record[0]
            car_name = record[1]
            pin_node = camera_node_list[camera_name2index[camera_name]]
            board_node = car_node_list[car_name2index[car_name]]
            pin_node.add_neighbour(board_node)
            board_node.add_neighbour(pin_node)
```

به این ترتیب گراف دلخواه ما construct میشود.

تابع الگوریتم pixie نیز، طبق مراحلی که بیان شده به صورت زیر است:

```
def pixie_algorithm(query_node, steps, alpha):
    pin_node = query_node
    for i in tqdm(range(steps)):
        pin_node.add_count()
        board_node = pin_node.get_random_neighbour()
        pin_node = board_node.get_random_neighbour()
        if np.random.rand() <= alpha:
            pin_node = query_node
```

در تابع بالا، query_node همان نود اولیه یا نود کوئری شده است و steps تعداد گام های random walk است و alpha نیز احتمال برگشتن به خانه اول یا همان query_node است.

حال برای اینکه الگوریتم خود را تست کنیم، یک دوربین رندوم را به الگوریتم میدهیم و دوربین های مشابه کشف شده را چاپ میکنیم.

Querying a sample camera

Here a choose a random camera and find similar cameras to it using pixie's algorithm

In [10]:
```
query_camera = random.choice(camera_node_list)
pixie_algorithm(query_camera, 500000, 0.008)
```

```
100%|██████████| 500000/500000 [00:01<00:00, 457519.74it/s]
```

In [11]:
```
proximity_list = sorted(camera_node_list, key= lambda x: x.count, reverse=True)
```

In [13]:
```
print(f'code of query camera:   {query_camera.code}')
print('------- Top 10 similar cameras -------')
for i in range(10):
    camera = proximity_list[i]
    print(f'{i+1}- camera code: {camera.code}         visited: {camera.count}')
```

```
code of query camera:   22009928
------- Top 10 similar cameras -------
1- camera code: 22009928         visited: 36005
2- camera code: 900212         visited: 9593
3- camera code: 900269         visited: 9363
4- camera code: 900244         visited: 9128
5- camera code: 100700853         visited: 8560
6- camera code: 900142         visited: 7983
7- camera code: 631634         visited: 7874
8- camera code: 900222         visited: 6548
9- camera code: 900101         visited: 5890
10- camera code: 900218         visited: 5726
```

۱۰ دوربینی که بیشترین تعداد visit را داشته اند، چاپ کرده ایم. مشاهده میشود که شبیه ترین دوربین به دوربین کوئری، خود دوربین است که منطقی نیز هست. بعد از دوربین های دیگری نیز شبیه شناخته شده اند، که حدود ۱۰۰۰۰ visit شده اند.

پس موفق شدیم با استفاده از pixie، دوربین های مشابه یک نود query را پیدا کنیم.

Classify کردن و تفکیک دوربین ها با استفاده از PCA و Clustering (ایده جدید)

در این مسئله ما قصد داریم تا دوربین هارا به ۵ دسته متفاوت تقسیم کنیم و همچنین هر دور بین را بر اساس ویژگی هایش، visualize کنیم.

برای اینکار ابتدا باید یک سری ویژگی برای هر دوربین استخراج کنیم. برای اینکار، ویژگی هر دوربین را به صورت تعداد ثبتی های هر دوربین در هر ساعت روز در نظر میگیریم. یعنی هر دوربین دارای یک بردار ویژگی ۲۴ بعدی است که که هر عنصر این بردار، تعداد ثبتی های آن دور بین را در ۲۴ ساعت روز به تفکیک، را در بر دارد.

پس بردار ویژگی هر دوربین ۲۴ بعدی است.

درحقیقت کد زیر، برای هر دوربین، بردار ویژگی را بر اساس دیتای داده شده، درمی آورد:

```
In [8]:
# now we create an rdd which contaons a feature vector for each camera
# the features are hours in a day and their value is the number of detected cars
def create_feature_vec(row):
    device_code = row[0]
    feature_vec = np.zeros((24, ))
    for record in row[1]:
        index = int(record[0][1])
        value = record[1]
        feature_vec[index] = value
    feature_vec = Vectors.dense(feature_vec)
    return (device_code, feature_vec)

camera_feature_vec = camera_hour_count.groupBy(lambda x: x[0][0]).mapValues(list).map(create_feature_vec)
```

```
In [9]:
camera_feature_vec.toDF(schema=['DEVICE_CODE', 'FEATURE_VEC']).show(20)
```

```
+-----------+--------------------+
|DEVICE_CODE|         FEATURE_VEC|
+-----------+--------------------+
|   22010054|[1302.0,758.0,416...|
|     107301|[435.0,307.0,170....|
|     200301|[397.0,212.0,96.0...|
|   22010135|[167.0,110.0,87.0...|
|    1001031|[8.0,6.0,3.0,3.0,...|
|     210110|[6.0,4.0,6.0,7.0,...|
|   22010139|[270.0,324.0,249....|
|     710101|[2.0,1.0,1.0,2.0,...|
|     631929|[5.0,1.0,1.0,4.0,...|
|     900212|[6844.0,3753.0,19...|
|    1001051|[2.0,2.0,1.0,3.0,...|
|     207701|[6.0,1.0,3.0,1.0,...|
|        128|[0.0,0.0,0.0,0.0,...|
|   22009802|[1.0,3.0,4.0,2.0,...|
|        103|[160.0,92.0,34.0,...|
|     900126|[267.0,145.0,87.0...|
|     203902|[1675.0,892.0,474...|
|     631610|[9.0,3.0,3.0,0.0,...|
|     631891|[0.0,0.0,0.0,0.0,...|
|   22009834|[33.0,6.0,10.0,6....|
+-----------+--------------------+
only showing top 20 rows
```

برای مثال برای تعدادی از دوربین ها، بردار ویژگی به صورت رو به رو است:

حال ما برای اینکه بتوان دوربین هارا visualize کرد، با استفاده از PCA، بردار ویژگی را به فضای دوبعدی میبریم. یعنی هر دوربین تنها با دو مولفه توصیف میشود.

برای این کاهش بعد از PCA روی دیتا استفاده میکنیم.

```
In [10]:
# perform pca on the data
camera_feature_df = spark.createDataFrame(camera_feature_vec, ['DEVICE_CODE', 'feature_vec'])
dim = 2   # good for visualization
pca = PCA(k=dim, inputCol='feature_vec')
pca.setOutputCol('reduced_feature_vec')
model = pca.fit(camera_feature_df)
```

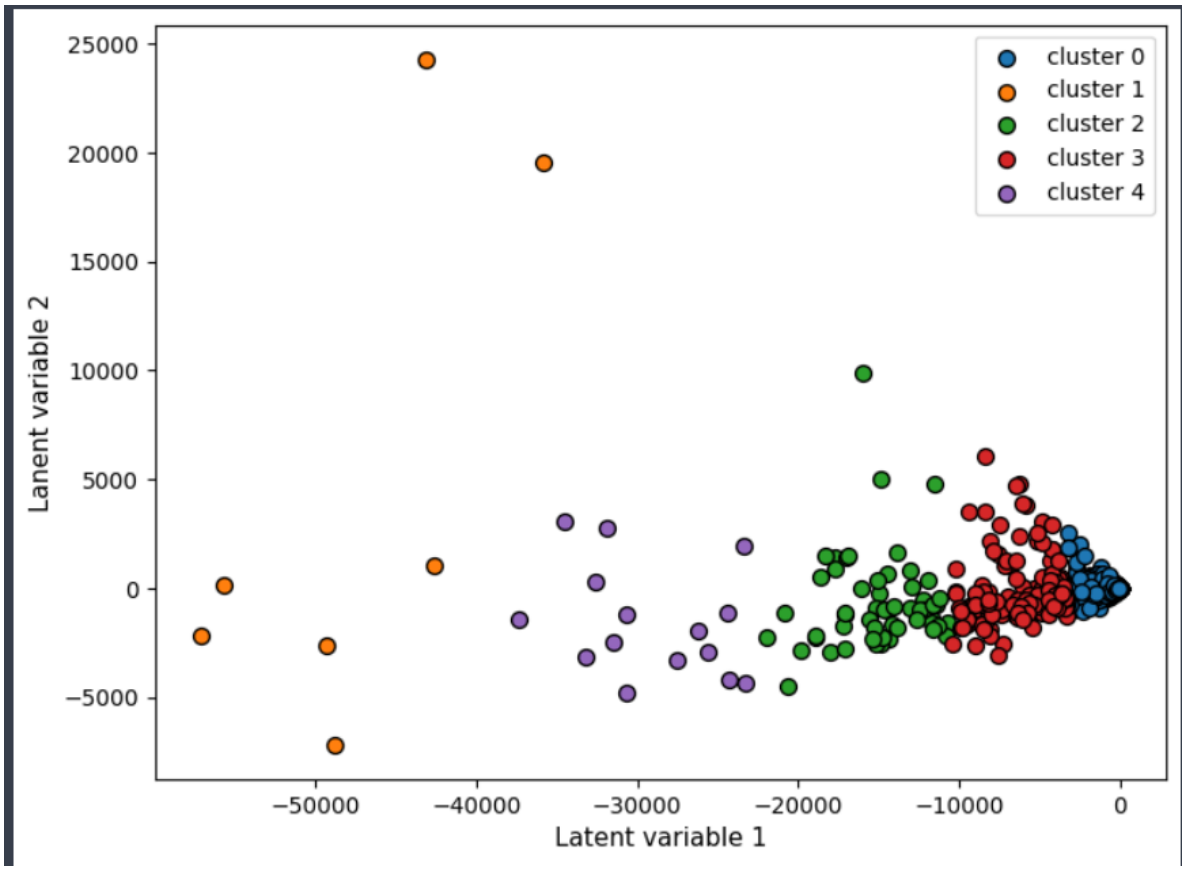
بردار ویژگی جدید که دوبعدی است، به صورت زیر است:

```
In [11]:
# show the pca results
camera_pca = model.transform(camera_feature_df)
camera_pca.show(20)
```

```
+-----------+--------------------+--------------------+
|DEVICE_CODE|         feature_vec| reduced_feature_vec|
+-----------+--------------------+--------------------+
|   22010054|[1302.0,758.0,416...|[-13060.650312009...|
|     107301|[435.0,307.0,170....|[-2100.3610658532...|
|     200301|[397.0,212.0,96.0...|[-4585.2098928953...|
|   22010135|[167.0,110.0,87.0...|[-1286.9896280502...|
|    1001031|[8.0,6.0,3.0,3.0,...|[-29.845326107283...|
|     210110|[6.0,4.0,6.0,7.0,...|[-3.4161668013355...|
|   22010139|[270.0,324.0,249....|[-1879.5721151562...|
|     710101|[2.0,1.0,1.0,2.0,...|[-6.8623194327505...|
|     631929|[5.0,1.0,1.0,4.0,...|[-34.060699659742...|
|     900212|[6844.0,3753.0,19...|[-57092.806501184...|
|    1001051|[2.0,2.0,1.0,3.0,...|[-22.272090893339...|
|     207701|[6.0,1.0,3.0,1.0,...|[-54.222304040916...|
|        128|[0.0,0.0,0.0,0.0,...|[-2716.5988239707...|
|   22009802|[1.0,3.0,4.0,2.0,...|[-10.366629946907...|
|        103|[160.0,92.0,34.0,...|[-1756.8509591644...|
|     900126|[267.0,145.0,87.0...|[-6938.1364774889...|
|     203902|[1675.0,892.0,474...|[-14341.437728655...|
|     631610|[9.0,3.0,3.0,0.0,...|[-1581.5495743922...|
|     631891|[0.0,0.0,0.0,0.0,...|[-18.868896518997...|
|   22009834|[33.0,6.0,10.0,6....|[-1749.6928249832...|
+-----------+--------------------+--------------------+
only showing top 20 rows
```

حال که بردار ویژگی دوبعدی شده است، روی دیتا الگوریتم K-means را اجرا میکنیم تا دوربین هارا به خوشه های متفاوت تقسیم کند.

چون بردار ویژگی دوبعدی است، میتوانیم دوربین هارا بر اساس latent variable های دوتایی شان رسم کنیم و بر اساس خوشه ای که در آن قرار دارند، رنگ آمیزی کنیم:



همانطور که مشاهده میشود، دوربین ها به ۵ دسته متفاوت تقسیم شده اند که از لحاظ مختصاتی نیز به طرز خوبی باهم اختلاف دارند. یعنی داده هایی که در خوشه های متفاوتی هستند تا حد خوبی در latent variable های خود اختلاف دارند و discrimination تاحد خوبی اتفاق افتاده است.

پس موفق شدیم که دوربین هارا به ۵ دسته متفاوت تقسیم کنیم.

یافتن تجزیه SVD ماتریس Utility خودرو ها و دوربین های پرتردد

در این مسئله قصد داریم تا تجزیه SVD ماتریس Utility خودرو و دوربین های پر تردد را به دست بیاوریم. برای این به دست آوردن تجزیه SVD از الگوریتم CUR استفاده خواهیم کرد.

اما قبل از آن باید دوربین ها و خودرو های پرتردد را به دست بیاوریم.

معیار پرتردد بودن را به صورت زیر است:

اگر camera_count را برای هر دوربین برابر تعداد دفعات ثبت ماشین درنظر بگیریم، یک دوربین پرتردد است اگر تعداد دفعات ثبتی آن بیشتر از mean(camera_count) + std(camera_count) باشد.

طبق این آستانه، دوربین های پرتردد را به دست می آوریم.

اگر car_count را برای هر ماشین برابر تعداد دفعات ثبت ماشین توسط دوربین ها درنظر بگیریم، یک ماشین پرتردد است اگر تعداد دفعات ثبتی آن بیشتر از mean(car_count) + std(car_count) باشد.

طبق این آستانه، ماشین های پرتردد را به دست می آوریم.

حال که ماشین ها و دوربین های پرتردد را تشکیل دادیم، ماتریس utility را به این گونه تشکیل میدهیم که مقدار آن در سطر i و ستون j ، برابر با تعداد ثبتی های ماشین پرتردد i توسط دوربین پرتردد j باشد.

برای انجام اینکار کافی است یکی یکی دیتا را بخوانیم و اگر ماشین پرتردد و دوربین پرتردد ظاهر شده، یک key/value به صورت زیر emit کنیم:

<key = (car_code, camera_code), value = 1>

سپس روی دیتای تولید شده یک reduce با عملیات جمع بزنیم تا درایه های ماتریس utility به دست بیاید:

Creating Utility Matrix

In [19]:
```
def utility_count(row):
    car = row[1]
    camera = row[0]
    car_index = car2index.value[car]
    camera_index = camera2index.value[camera]
    return ((car_index, camera_index), 1)

car_camera_counts = data_rdd.filter(lambda x: x[0] in frequent_camera_list and x[1] in frequent_cars_list).
```

In [20]:
```
utility_matrix = car_camera_counts.reduceByKey(add)
```

حال که ماتریس Utility را داریم، می آییم و تجزیه SVD آن را با استفاده از CUR انجام میدهیم.

برای این منظور مراحل زیر را طی میکنیم:

۱- ابتدا hyperparameter تعداد سطر و ستون تصادفی که آن را r مینامیم را انتخاب میکنیم. برای مثال r = 10

۲- حال به اندازه r تا سطر و ستون به طور تصادفی از ماتریس اصلی انتخاب میکنیم. البته توزیع انتخاب uniform نیست و برای هر سطر یا ستون، متناسب با اندازه آن سطر یا ستون است.

۳- حال ماتریس های R و C را تشکیل میدهیم که به ترتیب سطر و ستون های انتخابی رندوم هستند.

۴- حال ماتریس تلاقی آنها در ماتریس اصلی آنها را W مینامیم.

۵- داریم:

$$W = X\Sigma Y^T$$

$$U \coloneqq V(\Sigma^+)^2 X^T$$

۶- به این ترتیب، تجزیه ما کامل شده است و داریم:

$$M = CUR$$

محاسبه اندازه هر ستون:

```
In [29]:
def norm_of_column(item):
    return (item[0][1], item[1]**2)

column_norm = utility_matrix.map(norm_of_column).reduceByKey(add)
```

```
In [30]:
column_norms = [0 for _ in range(len(frequent_camera_list))]
for index, norm in column_norm.collect():
    column_norms[index] = norm
f_norm = sum(column_norms)
column_probs = [x/f_norm for x in column_norms]
```

محاسبه اندازه هر سطر:

Calculating norm of each row

In [32]:
```
def norm_of_row(item):
    return (item[0][0], item[1]**2)

row_norm = utility_matrix.map(norm_of_row).reduceByKey(add)
```

In [33]:
```
row_norms = [0 for _ in range(len(frequent_cars_list))]
for index, norm in row_norm.collect():
    row_norms[index] = norm
f_norm = sum(row_norms)
row_probs = [x/f_norm for x in row_norms]
```

In [35]:
```
print(f'Frobenius norm by summing row norms:  {f_norm}')
```

```
Frobenius norm by summing row norms:  10354
```

حال در کد زیر سطر و ستون رندوم را انتخاب میکنیم:

Selecting random rows

In [36]:
```
r = 10
```

In [37]:
```
# selecting each row and calculating the scaling factor for each selected row
# the scaling factor is calculated as defined by the book
random_row = random.choices(range(len(frequent_cars_list)), weights=row_probs, k=r)
unique_row = [*set(random_row)]
row_count = [random_row.count(row) for row in unique_row]
row_factor = [np.sqrt(k/(r*row_probs[index])) for k, index in zip(row_count, unique_row)]
```

Selecting random columns

In [39]:
```
# selecting each row and calculating the scaling factor for each selected row
# the scaling factor is calculated as defined by the book
random_column = random.choices(range(len(frequent_camera_list)), weights=column_probs, k=r)
unique_column = [*set(random_column)]
column_count = [random_column.count(column) for column in unique_column]
column_factor = [np.sqrt(k/(r*row_probs[index])) for k, index in zip(column_count, unique_row)]
```

درباره کد بالا به این نکته باید توجه داشته باشیم که وقتی ستون یا سطر تکراری داریم، آنهارا merge میکنیم.

یعنی اینکه ستون یا سطر های یکسان را باهم یکی میکنیم اما طبق فرمول کتاب باید هر سطر یا ستون را به عدد زیر نرمالایز کنیم:

$$\sqrt{\frac{k}{rp_i}}$$

که k برابر تعداد دفعات ستون یا سطر تکراری و p_i احتمال انتخاب آن ستون است. این مقادیر در لیست row_factor و column_factor محاسبه میشوند.

سپس طبق روال گفتع شده، ماتریس W و به تبع آن ماتریس U که نحوه به دست آوردن و فرمول آن ذکر شد را به دست می آوریم:

Forming W matrix

```
In [42]: print(f'shape of W:  ({len(unique_row), len(unique_column)})')

shape of W:  ((6, 7))

In [55]: W_shape = (len(unique_row), len(unique_column))
         W = np.zeros(W_shape)

         for i, row in enumerate(unique_row):
             for j, column in enumerate(unique_column):
                 query = utility_matrix.filter(lambda x: x[0] == (row, column)).collect()
                 if len(query):
                     W[i, j] = query[0][1]
                 else:
                     W[i, j] = 0
         print('the calculated W matrix')
         print(W)

the calculated W matrix
[[ 0.  0.  0.  0.  2.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 8.  0.  0.  0.  0. 29.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]]
```

به دست آوردن ماتریس U :

Constructing U matrix

In [62]:
```
X, S, YT = np.linalg.svd(W)
```

In [64]:
```
Sigma = np.zeros(W.shape)
for i, value in enumerate(S):
    Sigma[i, i] = S[i]
```

In [77]:
```
# Constructing the U matrix
U = YT.T @ np.linalg.pinv(Sigma) **2 @ X.T

print('The calculated U Matrix:')
print(f'{U}')
```

```
The calculated U Matrix:
[[0.         0.         0.         0.         0.00029384 0.        ]
 [0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.        ]
 [0.25       0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.00106519 0.        ]
 [0.         0.         0.         0.         0.         0.        ]]
```

پس U نیز محاسبه شد. R و C نیز قبل تر محاسبه شدند. پس داریم:

$$M = CUR$$

پس تجزیه SVD ماتریس M که ماتریس $utility$ هست را به دست آوردیم.