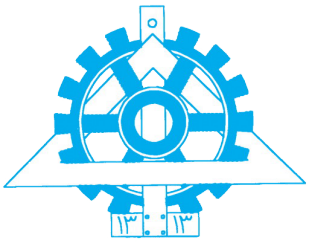

ضرب ماتریس-آرایه‌ای خلوت
با استفاده از تکنولوژی اسپارک

تهیه کننده گزارش

مصطفی محمدی قراسویی

۸۱۰۸۹۷۰۳۰

عنوان درس

الگوریتم های پردازش موازی

استاد راهنما

دکتر ترابی

رشته

الگوریتم و محاسبات

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده علوم مهندسی

فهرست مطالب

ضرب ماتریسی-آرایه‌ای خلوت با استفاده از تکنولوژی Spark

ماتریس خلوت

ماتریس خلوت به عنوان ماتریسی تعریف می‌شود که بیشتر المان‌های آن صفر است. به طور ویژه می‌توان ماتریس‌های خلوت را به صورت زیر تعریف کرد:

(تعداد کل المان‌های ماتریس) / (تعداد المان‌های صفر)

ماتریس‌های اسپارد به صورت سیستم‌های خطی ارتباط ضعیف معرفی می‌شوند. در برخی موارد بهتر است که ماتریس‌های خلوت را در قالب لیست‌های مختصات[1] ذخیره شوند. این کار به ما این اجازه را می‌دهد که تنها المان‌های غیر صفر ماتریس را به صورت لیست‌های سه ستونه ذخیره کنیم به طوری که ستون اول نماینده سطر، ستون دوم نماینده ستون و ستون سوم عدد غیر صفر در درایه سطر و ستون ماتریس خلوت است. به عنوان مثال کدهای پایتونی وجود دارد که با استفاده از ماژول NumPy عداد تصادف تولید می‌کند و می‌توان با استفاده از آن ماتریسی به اندازه $\mathbf{R}^{10,000\times10,000}$ با 20,000 عدد غیر صفر بین صفر و یک تولید کرد. ما دراین گزارش از ماتریس‌های مختصات با استفاده از کلاس scipy.sparse استفاده می‌کنیم که اجازه می‌دهد به سرعت از قالب‌های پیچیده و عمیق برای آزمایشات خود استفاده کنیم.

```
import numpy as np
from scipy.sparse import coo_matrix
from pyspark import SparkConf, SparkContext
```

```
n = 10000
```

```
indices = np.random.randint(0, n, size=(2*n, 2))
values = np.random.random(size=2*n)
```

```
sparse_representation = np.c_[indices, values[:, None]]
```

```
sparse_representation[:5]
```

```
array([[6.86000000e+02, 3.38500000e+03, 7.94401577e-01],
       [5.86500000e+03, 5.35100000e+03, 7.74288349e-01],
       [1.59000000e+03, 5.72300000e+03, 3.41039090e-01],
       [1.31100000e+03, 9.25600000e+03, 3.44232609e-01],
       [9.03100000e+03, 4.97900000e+03, 9.57372493e-01]])
```

این مقادیر را براس استفاده در آینده در هارد ذخیره می‌کنیم:

```
np.savetxt('sparse_matrix.txt', sparse_representation, delimiter=' ')
```

---

[1]. coordinate list (coo-matrix)

ماتریس‌های مختصات برای ساخت ماتریس‌های خلوت با استفاده از قالب ((data, (i,j)) به طوری که i و j آرایه ما هستند استفاده می‌کند.

قالب ماتریسی scipy فوق العاده مورد استفاده قرار دارد به طوری که با الگوریتم‌های sklearn کاملا منطبق می‌باشد.

در اینجا ما فقط به منظور تبدیل بازنمایش ماتریس خلوتمان به آرایه‌های عمیق برای مقایسه و تست استفاده می‌کنیم.

```
M_sparse = coo_matrix((values, (indices.T[0], indices.T[1])), shape=(n, n))
M_sparse
```

```
<10000x10000 sparse matrix of type '<type 'numpy.float64'>'
    with 20000 stored elements in COOrdinate format>
```

```
M = M_sparse.toarray()
M.shape
```

```
(10000, 10000)
```

```
type(M)
```

```
numpy.ndarray
```

مجموعه داده توزیع شده انعطاف‌پذیر اسپارک[۲] و انتقالات

ساختار پایه داده اسپارک بر اساس مجموعه داده[۳] توزیع شده انعطاف‌پذیر یا RDD است که در آن مجموعه المان‌ها در برابر خطا تحمل‌پذیر هستند که می‌توانند به صورت موازی با استفاده از تکنولوژی اسپارک کار کنند. تابع استاندارد برای نمونه‌برداری RDD با استفاده از ارجاع دهی به مجموعه‌داده در یک سیستم ذخیره‌سازی خارجی مانند سیستم فایل توزیع شده[۴]، HDFS، HBase، و یا هر ساختاری که بتواند ورودی قالب سیستم Hadoop را پیشنهاد دهد، کار می‌کند. در زیر ما با استفاده از RDD یک فایل متنی، با استفاده از PySpark نمونه‌برداری‌ها را انجام می‌دهیم. این کار با تفسیر فایل متنی به صورت دنباله‌ای از string ها می‌باشد که هر خط در فایل به صورت یک string تنها نمایش داده شده است.

---

2. Spark RDD (resilliant distributed dataset)

3. dataset

4. distributed file system(DFS)

```
M_rdd = lines.map(lambda l: map(float, l.strip().split(' ')))
M_rdd.take(10)
```

```
[[686.0, 3385.0, 0.7944015774384875],
 [5865.0, 5351.0, 0.7742883485561377],
 [1590.0, 5723.0, 0.3410390904855993],
 [1311.0, 9256.0, 0.3442326085505081],
 [9031.0, 4979.0, 0.957372493292332],
 [3627.0, 3573.0, 0.6118458463822919],
 [9061.0, 6866.0, 0.5300661428327066],
 [1471.0, 7093.0, 0.0834423431861061],
 [6158.0, 5673.0, 0.13409163529952728],
 [7761.0, 3392.0, 0.2583474112696168]]
```

با استفاده از تابع take(۱۰) می‌توانیم ۱۰ آیتم اول از RDD را بدست بیاوریم که معادل ۱۰ سطر اول فایلی است که قبلا در آن ماتریسمان را ذخیره کرده‌ایم. می‌خواهیم این خطوط را از حالت رشته‌ای به رکوردهای ۳ مقداره تبدیل کنیم. برای این کار از انتقال[۵] بر روی RDD استفاده می‌کنیم. ساده‌ترین روش برای این کار استفاده از تابع map() است که بر روی هر یک از آیتم‌های گرفته شده در RDD عمل می‌کند.

```
conf = SparkConf()
sc = SparkContext(conf=conf)
```

```
lines = sc.textFile('sparse_matrix.txt')
lines.take(10)
```

```
[u'6.860000000000000e+02 3.385000000000000e+03 7.944015774384874939e-01',
 u'5.865000000000000e+03 5.351000000000000e+03 7.742883485561377066e-01',
 u'1.590000000000000e+03 5.723000000000000e+03 3.410390904855993277e-01',
 u'1.311000000000000e+03 9.256000000000000e+03 3.442326085505080790e-01',
 u'9.031000000000000e+03 4.979000000000000e+03 9.573724932923319830e-01',
 u'3.627000000000000e+03 3.573000000000000e+03 6.118458463822918914e-01',
 u'9.061000000000000e+03 6.866000000000000e+03 5.300661428327065883e-01',
 u'1.471000000000000e+03 7.093000000000000e+03 8.344234318610610490e-02',
 u'6.158000000000000e+03 5.673000000000000e+03 1.340916352995272787e-01',
 u'7.761000000000000e+03 3.392000000000000e+03 2.583474112696168001e-01']
```

بنابراین با این کار می‌توانیم RDDای که شامل بازنمایش ماتریس مختصات است را بدست بیاوریم.

ضرب ماتریس-آرایه‌ای با استفاده از اسپارک RDD:

یکی از انتقالات ساده روی RDD ها، نگاشت و کاهش است که این کار را با استفاده از دو تابع map و reduceByKey انجام می‌پذیرد که دقیقا مشابه کاری است که در روش MapReduce در موازی سازی استفاده می‌شود. به طور خلاصه می‌توان این روش را به سه مرحله تقسیم کرد که در زیر به آن اشاره مختصری خواهیم داشت:

---

[5]. transformation

۱- **map**: تابع را بر روی هر یک از المان‌های ورودی مجموعه‌داده اعمال می‌کند و نتیجه حاصل از این کار ترتیب زوج مقدار-کلید[۶] به صورت زیر است:

$$[(k_1, v_1), (k_2, v_2), (k_3, v_3), \ldots].$$

۲- **Group**: زوج‌های کلید-مقدار را به صورت مرتب شده و سازمان‌یافته بر روی کلیدها مرتب می‌کند، بنابراین هر یک از کلیدهای منحصر به فرد به لیستی از مقادیر منتسب خواهد شد:

$$[(k_1, [v_1, v_3, \ldots]), (k_2, [v_2, \ldots]), \ldots].$$

۳- **Reduce**: مقادیر هر یک از کلیدها را متناسب با لیست‌شان با استفاده از برخی از توابع ترکیب می‌کند. این تابع بر روی دو مقدار در هر زمان تعریف می‌شود و باید این قابل ارتباط[۷] و قابل ترکیب[۸] باشند.

برای مثال، در زیر می‌توان تابع کاهش استفاده شده به منظور به دست آوردن جمع تمام المان‌های اجتماع شده را در یک کلید دید:

```
def summation(v1, v2):
    return v1 + v2
```

که می‌تواند به صورت دقیق‌تر و کوتاه‌تر با استفاده از تابع **lambda** آن را نوشت، به صورت زیر:

```
lambda v1, v2: v1 + v2
```

همان‌طور مشخص است، بخش نگاشت-کاهش برای ضرب ماتریس‌های خلوت و آرایه‌ها خوش‌تعریف هستند. به منظور اثبات غیر رسمی این موضوع اجازه دهید که مثالی را بیان کنیم.

معادله ماتریس زیر را در نظر بگیرید:

$$y = A\,x$$

با $A \in \mathbf{R}^{m\times n}$، هر المان از y به صورت زیر تعریف می‌شود.

$$y_i = \sum_{j=1}^{n} A_{ij} x_j\,.$$

بنابراین، بازنمایشی از **RDD** ماتریس و آرایه x که در حافظه نگه‌داری شده است را داریم. حال حاصل ضرب را به صورت زیر انجام می‌دهیم:

۱- **Map**: رکورد (i, j, A_{ij}) را دریافت و خروجی را به صورت رکورد $(i, A_{ij} * x[j])$ را تولید می‌کند.

۲- **Group**: کلیه مدخل‌ها را بر اساس اندیس سری گروه‌بندی می‌کند.

---

6. key-value

7. communitive

8. associative

۳-Reduce: مقادیر هر یک از اندیس‌های سطری را جمع می‌کند.

reduceByKey اسپارک مرحله ۲ و ۳ را به صورت یک‌جا انجام می‌دهد. تمام آن‌چه باقی می‌ماند دسته‌بندی کردن نتیجه‌ها است. ما باید نتایج را بر اساس کلیدها مرتب کرده و سپس کلیدهای خطادار را مدیریت کنیم که ممکن است زمانی رخ دهد که مثلا هیچ یک از المان‌های یک سطر در ماتریس ما غیر صفر نباشد.

اجازه دهید که این مطالب را به صورت عملی انجام دهیم:

ابتدا یک آرایه تصادفی برای ضرب با ماتریس خودمان که قبلا ساختیم و آن را ذخیره کرده‌ایم بسازیم.

```
v_in = np.random.random(size=n)
```

در مرحله بعد عملیات نگاشت-کاهش را با استفاده از اسپارک انجام می‌دهیم. توجه داشته باشید که چطور انتقالات می‌تواند به صورت زنجیره‌ای با یکدیگر اتفاق بیافتد.

در مرحله آخر با استفاده از تابع collect نتایج RDD را به یک لیست پایتون تبدیل می‌کنیم. این عمل باید با دقت انجام شود. اگر نتیجه بسیار بزرگ باشد، ممکن است که در برخی از مقادیر حقیقی مشکل به وجود بیاورد. در این مورد، می‌دانیم که نتیجه آرایه به اندازه آرایه ورودی است، بنابراین می‌توانیم RDD را به صورت امن در حافظه فعال ذخیره کنیم.

```
v_out_spark_raw = np.array(
    M_rdd\
        .map(lambda x: (x[0], v_in[int(x[1])] * x[2]))\
        .reduceByKey(lambda v1, v2: v1 + v2)\
        .sortByKey()\
        .collect()
)
```

```
len(v_out_spark_raw)
```

```
    8620
```

اما نتیجه‌ای انتظار داشتیم یک آرایه $\mathbf{R}^{10,000}$ است! همان‌طور که قبل‌تر نیز اشاره شد هنگامی این عمل اتفاق می‌افتد که ماتریس ما مقادیر غیر صفر در برخی از سطرهایش نداشته باشد. می‌توانیم این مشکل را به سادگی با استفاده از چند روش ساده ماژول NumPy در پایتون مدیریت کنیم. به کد زیر نگاهی بیاندازید:

```
v_out_spark = np.zeros(n)
v_out_spark[map(int, v_out_spark_raw.T[0])] = v_out_spark_raw.T[1]
```

نهایتا، می‌توانیم آن چه را که با استفاده از تکنیک نگاشت-کاهش و ضرب معمولی انجام می‌پذیرد را مشاهده کنیم:

```
v_out_numpy = M.dot(v_in)
```

```
np.allclose(v_out_spark, v_out_numpy)
```

```
    True
```

```
v_out_numpy[:20]
```

```
    array([0.20550791, 0.24228745, 1.88363129, 0.66752008, 0.01382379,
           0.28009837, 0.52376888, 0.10529744, 0.        , 0.62103075,
           1.07149336, 0.06488723, 0.        , 1.02896754, 0.63032014,
           0.30943638, 0.41731815, 1.30066203, 0.29911015, 0.01944877])
```

```
v_out_spark[:20]
```

```
    array([0.20550791, 0.24228745, 1.88363129, 0.66752008, 0.01382379,
           0.28009837, 0.52376888, 0.10529744, 0.        , 0.62103075,
           1.07149336, 0.06488723, 0.        , 1.02896754, 0.63032014,
           0.30943638, 0.41731815, 1.30066203, 0.29911015, 0.01944877])
```

اتمام گزارش.

———

لازم به ذکر است که کد گزارش ضرب ماتریس-آرایه به صورت iPython به همراه همین نسخه و فایل‌های ضرب ماتریس و ماتریس‌های مختصات و غیره ارسال می‌گردد.