배열 정렬¶

NumPy 배열의 값을 정렬하는 알고리즘을 다루겠다. 이 알고림즘은 캄퓨터 과학 입문 과정의 단골 주체로, 강의를 들어본 독자라면 아마 꿈에서 삽입 정렬(insertion sorts), 선택정렬(selection sort), 병렬 정렬(merge sorts), 퀵 정렬(quick sorts), 버블 정렬(bubble sorts)등을 수행해 본 적이 있을 것이다. 이 알고리즘은 모두 리스트나 배열의 값을 정렬하는 비슷한 작업을 수행한다.

예를 들어 간단한 선택 정렬은 리스트의 최소값을 반복적으로 찾아서 리스트가 정렬될 때까지 값을 교환한다. 파이썬으로는 몇 줄만 코딩하면 된다.

```
In [1]:import numpy as np
        def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x

In [2]:x = np.array([2, 1, 4, 3, 5])
        selection_sort(x)
Out[2]:array([1, 2, 3, 4, 5])
```

컴퓨터 과학 전공 1 년차 학생이라면 선택 정렬은 로직이 단순하다는 점에서 유용하지만 배열이 큰 경우에는 너무 느려서 쓸모가 없다고 말할 것이다. N 개의 값을 가진 리스트의 경우 N 번 반복해야하며: 매회 교환한 값을 찾기 위해 대략 ~N 회 비교를 수행한다. 이 알고리즘의 특징을 기술할 때 자주 사용되는 '대문자 O' 표기법 면에서 보면 선택 정렬은 평균의 복잡도를 갖는다. 리스트의 항목 개수가 2 배가 되면 실행시간은 4 개로 늘어날 것이다.
그러나 이러한 선택 정렬도 고전적인 정렬 알고림즘인 보고정렬(bogosort)보다는 훨씬 낫다.

```
In [3]:def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x
In [4]:x = np.array([2, 1, 4, 3, 5])
        bogosort(x)
Out[4]:array([1, 2, 3, 4, 5])
```

이 정렬운 순전히 운에 의존한다. 이 방식은 결과가 정렬될 때까지 반복적으로 배열을 임의로 섞는다. 평균 복잡도가 O[N x N](N 팩토리알의 N 배)라서 이 알고림즘은 당연히 실제 계산에서는 절대 사용하지 않는다.

다행이도 파이썬에는 방금 보여주 ㄴ 간단한 알고리즘보다 훨씬 더 효율적인 정렬 알고리즘이 내장돼 있다.

NumPy 의 빠른 정렬: np.sort 와 np.argsort

파이썬에도 리스트를 정렬하는 내장함수인 sortd 와 sorted 가 있지만, NumPy 의 np.sort 함수가 훨씬 더 효율적이고 유용하다. np.sort 는 기본적으로 퀵 정렬 알고리즘을 사용하지만 병렬 정렬(mergesort)과 힙 정렬(heapsort)도 사용할 수 있 다. 대부분의 애플리케이션에서는 기본 퀵 정렬로도 충분하다.

```
In [5]:x = np.array([2, 1, 4, 3, 5])
        np.sort(x)
```

```
Out[5]:array([1, 2, 3, 4, 5])
```

배열을 그 자리에서 바로 정렬하는 게 좋으면 배열의 sort 메서드를 사용하면 된다.:

```
In [6]:x.sort()
       print(x)
[1 2 3 4 5]
```

이와 관련된 함수로는 정렬된 요소의 인덱스를 반환하는 argsort 가 있다.

```
In [7]:x = np.array([2, 1, 4, 3, 5])
       i = np.argsort(x)
       print(i)
[1 0 3 2 4]
```

이 결과의 첫 번째 요소는 가장 작은 요소의 인덱스고 두 번째로 작은 요소의 인덱스를 제공하는 식이다. 이 인덱스는 원하다면 정렬된 배열을 구성하는 데(팬시 인덱싱을 통해) 사용될 수 있다.

```
In [8]:x[i]
Out[8]:array([1, 2, 3, 4, 5])
```

행이나 열 기준으로 정렬하기

NumPy 의정렬 알고리즘의 유용한 기능은 axis 인수를 사용해 다차원 배열의 특정 행이나 열에 따라 정렬할 수 있다는 것이다.:

```
In [9]:rand = np.random.RandomState(42)
       X = rand.randint(0, 10, (4, 6))
       print(X)
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
In [10]:# x 의 각 열을 정렬
np.sort(X, axis=0)
Out[10]:array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
In [11]:# X 의 각 행을 정렬
       np.sort(X, axis=1)
Out[11]:array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

이 코드는 각 행이나 열을 독립적인 배열로 취급하므로 행 또는 열 값 사이의 관계는 잃어버린다는 점을 명심하자.

부분 정렬: 파티션 나누기

때때로 전체 배열을 정렬할 필요는 없고 단순히 배열에서 K 개의 가장 작은 값을 찾고 싶을 때가 있다. NumPy 에서는 np.partition 함수에서 이 기능을 제공한다. np.partition 은 배열과 숫자 K 를 취해 새로운

배열을 반환하는데, 반환된 파티션의 왼쪽에는 K개의 가장 작은 값이 있고 오른쪽에는 나머지 값이 임의의 순서로 채워져 있다.

```
In [12]:x = np.array([7, 2, 3, 1, 6, 5, 4])
         np.partition(x, 3)
Out[12]:array([2, 1, 3, 4, 6, 5, 7])
```

결과로 얻은 배열의 처음 세 개의 값은 배열의 가장 작은 값 세개에 해당하며, 배열의 나머지 위치에는 나머지 값이 들어 있다. 두 파티션 내의 요소는 임의의 순서를 가진다.

```
In [13]:np.partition(X, 2, axis=1)
Out[13]:array([[3, 4, 6, 7, 6, 9],
         [2, 3, 4, 7, 6, 7],
         [1, 2, 4, 5, 7, 7],
         [0, 1, 4, 5, 9, 5]])
```

그 결과, 첫 두개의 슬롯에는 해당 행의 가장 작은 값이 채워지고 나머지 슬롯에는 그 밖의 값이 채워진 배열을 얻는다.

마지막으로 정렬의 인덱스를 계산하는 np.argpartition 이 있다.

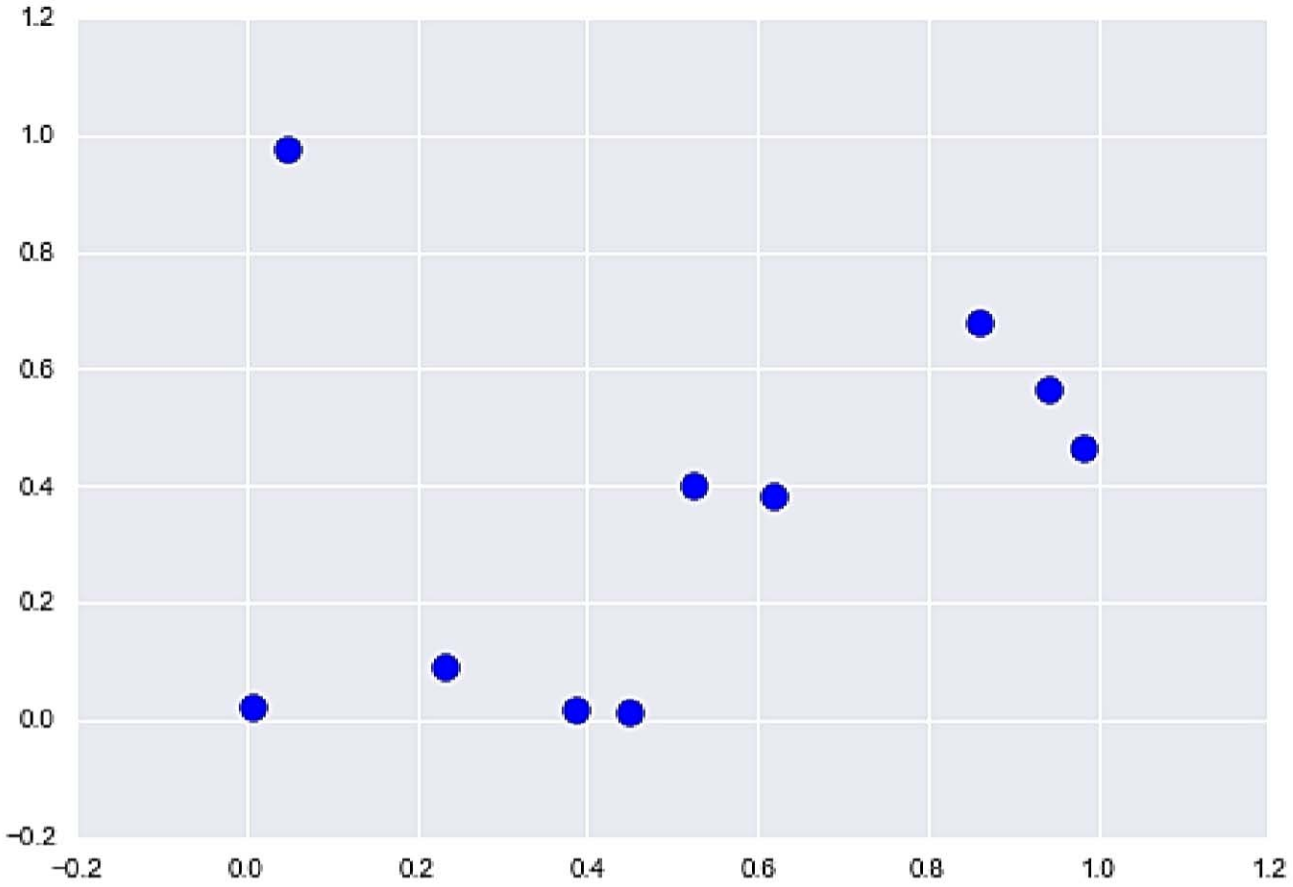
예제: k 최근접 이웃 알고리즘

집합에서 각 점의 가장 가까운 이웃들을 찾기 위해 여러 축을 따라 argsort 함수를 어떻게 사용하는지 간단히 살펴본다. 우선 2 차원 평면에 임의의 점 10 개를 가지는 집합을 만들자. 표준 규약에 따라 이것들을 10x2 배열에 배치할 것이다

```
In [14]:X = rand.rand(10, 2)
```

이 점들이 어떤 모습일지 떠올릴 수 있도록 간단하게 산포도로 표시해 보자

```
In [15]:%matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn; seaborn.set() # Plot styling
         plt.scatter(X[:, 0], X[:, 1], s=100);
```



이제 각 쌍의 점 사이의 거리를 계산할 것이다, 두 점 사이의 제곱은 각 차원 차이를 제곱해서 더한 값과 같다는 사실을 기억하자. NumPy 에서 제공하는 효율적인 브로드캐스팅과 집계 루틴을 사용해 코드 한 줄로 제곱 거리 행렬을 계산할 수 있다.

```
In [16]:dist_sq = np.sum((X[:, np.newaxis, :] - X[np.newaxis, :, :]) ** 2, axis=-1)
```

이 연산에는 많은 것이 포함되어 있으며, NumPy 의 브로드캐스팅 규칙이 익숙하지 않은 독자라면 다소 헷갈릴 수도 있다. 이런 코드를 만나면 코드를 구성 단계로 나누어 보는 것이 유용할 수 있다.

```
In [17]:# 각 쌍의 점 사이에좌표 차이를 계산함
         differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
         differences.shape
Out[17]:(10, 10, 2)
In [18]:# 좌표 차이를 제곱합
         sq_differences = differences ** 2
         sq_differences.shape
Out[18]:(10, 10, 2)
In [19]:# 제곱 거리를 구하기 위해 좌표 차이를 더함.
         dist_sq = sq_differences.sum(-1)
         dist_sq.shape
Out[19]:(10, 10)
```

작업한 내용을 확인 하는 차원에서 이 행렬의 대각선이(즉, 각 점과 그 점 사이의 거리 집합) 모두 0 인지 확인해야 한다.

```
In [20]:dist_sq.diagonal()
```

```
Out[20]:array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

확인이 끝났다! 이 제곱 거리를 변환하면 이제 np.argsort 를 이용해 행별로 정렬할 수 있다.

```
In [21]:nearest = np.argsort(dist_sq, axis=1)
        print(nearest)
[[0 3 9 7 1 4 2 5 6 8]
 [1 4 7 9 3 6 8 5 0 2]
 [2 1 4 6 3 0 8 9 7 5]
 [3 9 7 0 1 4 5 8 6 2]
 [4 1 8 5 6 7 9 3 0 2]
 [5 8 6 4 1 7 9 3 2 0]
 [6 8 5 4 1 7 9 3 2 0]
 [7 9 3 1 4 0 5 8 6 2]
 [8 5 6 4 1 7 9 3 2 0]
 [9 7 3 0 1 4 5 8 6 2]]
```

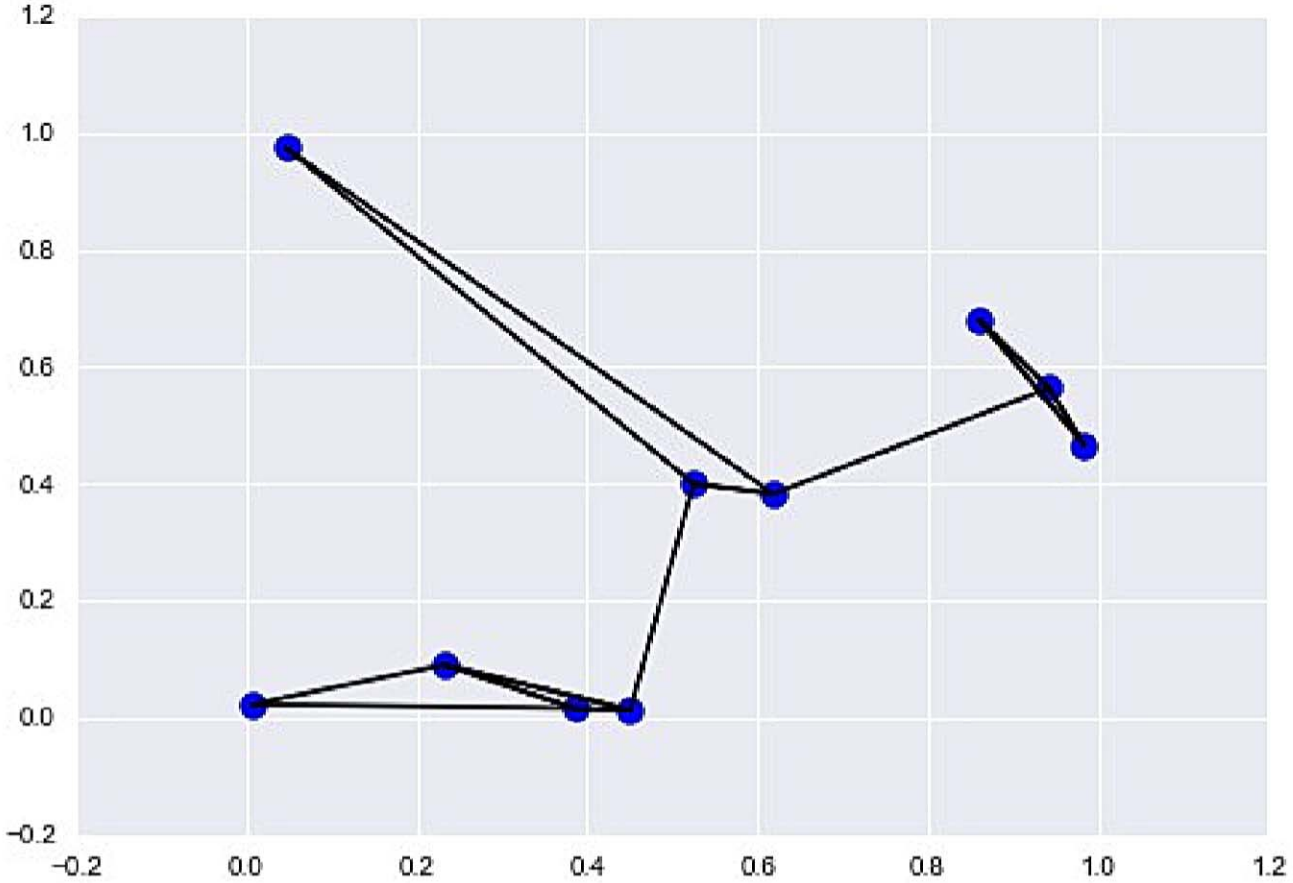
첫 번째 열은 숫자 0 부터 9 까지 순서대로 나타나는 것을 알 수 있다. 이미 예상했겠지만 각 점의 가장 가까운 이웃은 자기 자신이기 때문이다.

여기서는 전체 정렬을 사용해 실제로 이 경우에 필요한 더 많은 일을 한셈이다, 가장 가까운 k 이웃을 구하기만 하면 된다면 각 행을 파티션으로 나눠 가장 작은 k+1 개의 제곱 거리가 먼저 오고 그 보다 큰 거리의 요소를 배열의 나머지 위치에 채워지게만 하면된다. 이 작업은 np.partition 함수로 할 수 있다.

```
In [22]:K = 2
        nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

이 이웃의 네트워크를 시각화하기 위해 각 점을 가장 가까운 두개의 이웃과 연결한 선과 함께 프로팅해 보자

```
In [23]:plt.scatter(X[:, 0], X[:, 1], s=100)
# 각 점을 두 개의 가장 가까운 이웃과 선으로 이음
K = 2
for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```



도표의 각 점은 두 개의 가장 가까운 이웃과 선으로 연결돼 있다. 언뜻 보면 일부 점에서 두 개 이상의 선이 나와서 이상해 보일 수 있다. 그 이유는 점 A가 점 B의 가장 가까운 이웃 중 하나라는 사실이 반드시 점 B가 점 A의 가장 가까운 이웃 중 하나라는 것을 의미하지는 않기 때문이다.

이 방식에서 사용한 브로드캐스팅과 행 단위의 정렬이 루프를 작성하는 것보다 더 복잡해 보이지만, 파이썬에서는 그것이 이 데이터를 조작하는 매우 효율적인 방식이다. 같은 유형의 작업을 직접 데이터를 반복해 각 이웃의 집합을 정렬해서 하고 싶을 수도 있겠지만 그것이 위에서 보여준 벡터화 방식보다 더 느린 알고리즘을 만들어낼 것이 거의 확실하다. 이 방식의 장점은 입력 데이터의 크기에 구애받지 않는 방식으로 작성됐다는 점이다. 임의의 차원에서 100개든 100만 개든 그사이의 이웃을 쉽게 계산할 수 있고 코드도 동일하다.

마지막으로 매우 많은 개수의 최근접 이웃을 검색하는 경우에는 복잡도가 O[N log N]인 트리 기반의 and/or 근사 알고리즘(and/or approximate algorithm)을 사용할 수 있는데, 이 알고리즘이 복잡도가 $O[N^2]$인 무차별 대입 알고리즘[brute-force-alorithm]보다 더 낫다. 이 알고리즘의 대표적인 예로 Scikit-Leam에 구현된 KD 트리를 들 수 있다.