Ch.12 k-Nearest Neighbors

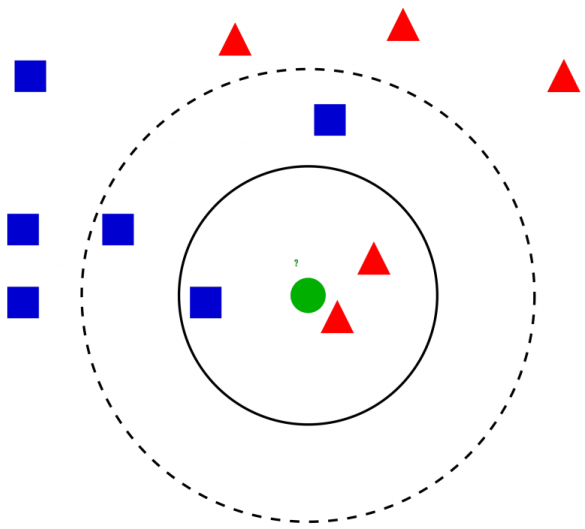
```
from collections import Counter
from linear_algebra import distance
from stats import mean
import math, random
import matplotlib.pyplot as plt
```

Nearest neighbors classification (최근접 이웃 분류)

- 다음 대통령 선거에서 제가 어떻게 투표할지 예측하려고 한다고 가정해봅시다.
  - 한 가지 합리적인 접근 방식은 **내 이웃들이 어떻게 투표할 계획인지 보는 것**입니다.
- 이제 단순히 지리적 정보만 아는 것이 아니라 **제 나이, 소득, 자녀 수 등**에 대해서도 알고 있다고 상상해보세요.
  - 모든 이러한 차원에서 저와 가까운 이웃들을 살펴보세요.

매우 간단한 모델

- **거리**에 대한 개념
- 서로 **가까운** 점들이 **유사하다**는 가정
- k개의 가장 가까운 **라벨이 있는** 점을 찾아 새로운 출력에 대해 투표하게 합니다.
  - 라벨은 True와 False일 수 있습니다 ("스팸인가?", "독성이 있는가?", "보기에 즐거울 것인가?"와 같은 질문에 대한 답).
  - 또는 카테고리일 수 있습니다, 예를 들어 영화 등급 (G, PG, PG-13, R, NC-17) 등.
  - 또는 대통령 후보자의 이름일 수 있습니다.
  - 또는 선호하는 프로그래밍 언어일 수 있습니다.



- 다음의 `raw_majority_vote`는 동점을 고려하지 않습니다.

```
from collections import Counter

def raw_majority_vote(labels):
    # 라벨들의 빈도를 세기 위해 Counter 객체 생성
    votes = Counter(labels)

    # 가장 많이 등장한 라벨과 그 빈도수를 반환 (most_common(1)은 빈도가 가장 높은 1개의 요소를 반환)
    winner, _ = votes.most_common(1)[0]

    # 가장 많이 등장한 라벨 반환
    return winner
```

동점 해결 방법

- 동점이 발생할 경우, 무작위로 한 명의 승자를 선택합니다.
- 거리로 투표에 가중치를 두고, 가중된 승자를 선택합니다.
- 고유한 승자를 찾을 때까지 k 값을 줄입니다.

```
from collections import Counter

def majority_vote(labels):
    """라벨들이 가장 가까운 것부터 가장 먼 순서로 정렬되어 있다고 가정합니다."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count for count in vote_counts.values() if count == winner_count])

    if num_winners == 1:
        return winner                     # 유일한 승자가 있으므로 반환
    else:
        return majority_vote(labels[:-1]) # 가장 먼 라벨을 제외하고 다시 시도
```

- 동점 해결 시 가장 먼 라벨이 여러 개 있을 경우, 이들을 한 번에 모두 제거해야 합니다. 이 경우 결정을 내리지 못할 수도 있습니다.
- 또한, `knn_classify` 구현이 부분적으로 잘못되어 있으며, 동점 상황에서 k-최근접 이웃을 고려하지 않습니다.

```
def knn_classify(k, labeled_points, new_point):
    """각 라벨이 지정된 점은 (점, 라벨) 쌍이어야 합니다"""

    # 라벨이 지정된 점들을 가장 가까운 것부터 먼 순서로 정렬
    by_distance = sorted(labeled_points,
                         key=lambda point_label: distance(point_label[0], new_point))

    # 가장 가까운 k개의 라벨을 찾기
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # 다수결로 최종 라벨 결정
    return majority_vote(k_nearest_labels)
```

Example: Favorite Languages

- 커뮤니티 참여 부사장은 이번 결과를 사용하여 설문 조사에 포함되지 않은 지역에서 선호하는 프로그래밍 언어를 예측할 수 있는지 알고 싶어합니다.
- 이 예시를 위해 k-최근접 이웃(KNN) 알고리즘을 사용하여 주어진 데이터로부터 새로운 지역의 선호하는 프로그래밍 언어를 예측해보겠습니다.

```
cities = [(-86.75,33.5666666666667,'Python'),(-88.25,30.6833333333333,'Python'),(-112.0166666666
cities = [([longitude, latitude], language) for longitude, latitude, language in cities]
```

용어 정의:

- k는 우리가 선택하고 실험하는 **하이퍼파라미터**입니다.
- "**Leave-one-out cross validation**"을 기반으로 정확성을 평가합니다 (하나의 데이터는 테스트 데이터로 사용하고, 나머지 데이터는 훈련 데이터로 사용합니다).

```
# 여러 다른 k 값을 시도하여 정확도 평가
for k in [1, 3, 5, 7]:
    num_correct = 0

    for location, actual_language in cities:

        # 현재 도시를 제외한 나머지 도시들을 훈련 데이터로 사용
        other_cities = [other_city
                        for other_city in cities
                        if other_city != (location, actual_language)]

        # 현재 도시의 프로그래밍 언어 예측
        predicted_language = knn_classify(k, other_cities, location)

        # 예측이 실제 언어와 일치하면 정확도 증가
        if predicted_language == actual_language:
            num_correct += 1

    # 각 k 값에 대한 결과 출력
    print(k, "neighbor[s]:", num_correct, "correct out of", len(cities))
```

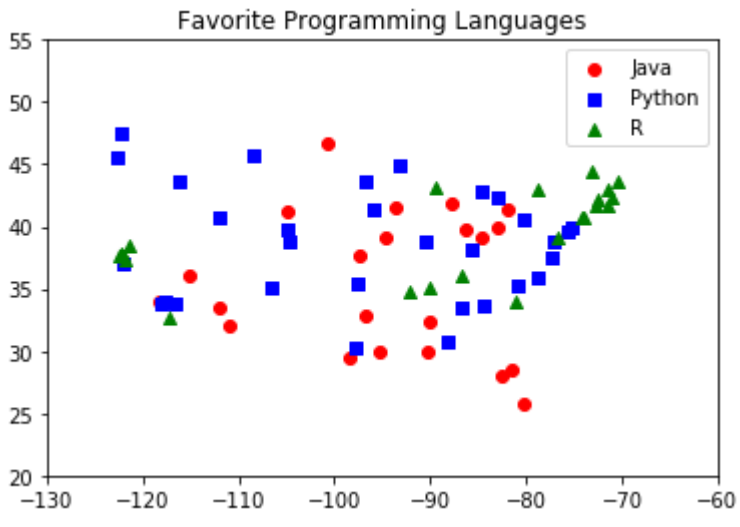
```
1 neighbor[s]: 40 correct out of 75
3 neighbor[s]: 44 correct out of 75
5 neighbor[s]: 41 correct out of 75
7 neighbor[s]: 35 correct out of 75
```

- 주어진 데이터를 사용하여 k-최근접 이웃(KNN) 알고리즘을 통해 Leave-one-out 교차 검증을 수행한 결과, 3-최근접 이웃이 약 59%의 정확도로 가장 좋은 성능을 보였습니다.

```
def plot_state_borders(plt, color='0.8'):
    # 주 경계를 그리는 함수. 현재는 구현되지 않음.
    pass

def plot_cities():
    # key는 언어, value는 (경도 리스트, 위도 리스트) 쌍
    plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

    # 각 언어마다 다른 마커와 색상을 설정
    markers = { "Java" : "o", "Python" : "s", "R" : "^" }
    colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

    # 도시 데이터를 반복하면서 경도와 위도를 해당 언어의 리스트에 추가
    for (longitude, latitude), language in cities:
        plots[language][0].append(longitude)
        plots[language][1].append(latitude)

    # 각 언어에 대해 산점도 시리즈를 생성
    for language, (x, y) in plots.items():
        plt.scatter(x, y, color=colors[language], marker=markers[language],
                          label=language, zorder=10)

    plot_state_borders(plt)    # assume we have a function that does this
    # 주 경계를 그리는 함수를 호출 (현재는 구현되지 않음)
```

```
    plt.legend(loc=0)           # matplotlib가 범례 위치를 선택하게 함
    plt.axis([-130, -60, 20, 55]) # 축의 범위를 설정
    plt.title("Favorite Programming Languages") # 제목 설정
    plt.show() # 그래프 표시
```

```
plot_cities()
```

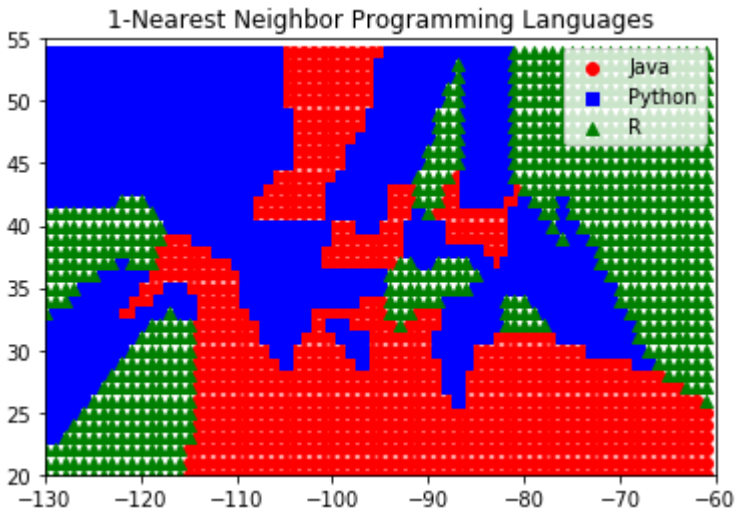


Classification regions (분류 영역)

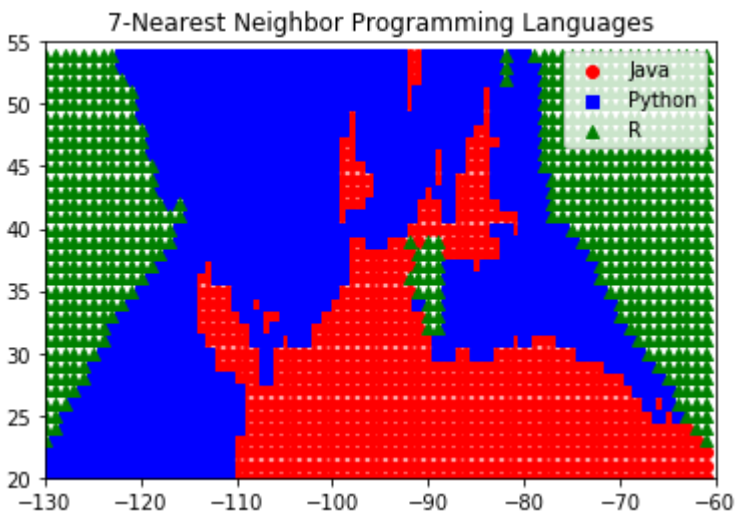
- 각 최근접 이웃 방식에 따라 어떤 영역이 어떤 언어로 분류될지를 시각화합니다.

```
def classify_and_plot_grid(k=1):
    plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }
    markers = { "Java" : "o", "Python" : "s", "R" : "^" }
    colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

    for longitude in range(-130, -60):
        for latitude in range(20, 55):
            predicted_language = knn_classify(k, cities, [longitude, latitude])
            plots[predicted_language][0].append(longitude)
            plots[predicted_language][1].append(latitude)

    # create a scatter series for each language
    for language, (x, y) in plots.items():
        plt.scatter(x, y, color=colors[language], marker=markers[language],
                    label=language, zorder=0)

    plot_state_borders(plt, color='black')  # 주 경계를 그리는 함수를 호출 (현재는 구현되지 않음)

    plt.legend(loc=0)           # matplotlib가 범례 위치를 선택하게 함
    plt.axis([-130, -60, 20, 55])  # 축의 범위를 설정
    plt.title(f"{k}-Nearest Neighbor Programming Languages")  # 제목 설정
    plt.show()
```

```
classify_and_plot_grid()
```




```
classify_and_plot_grid(7)
```



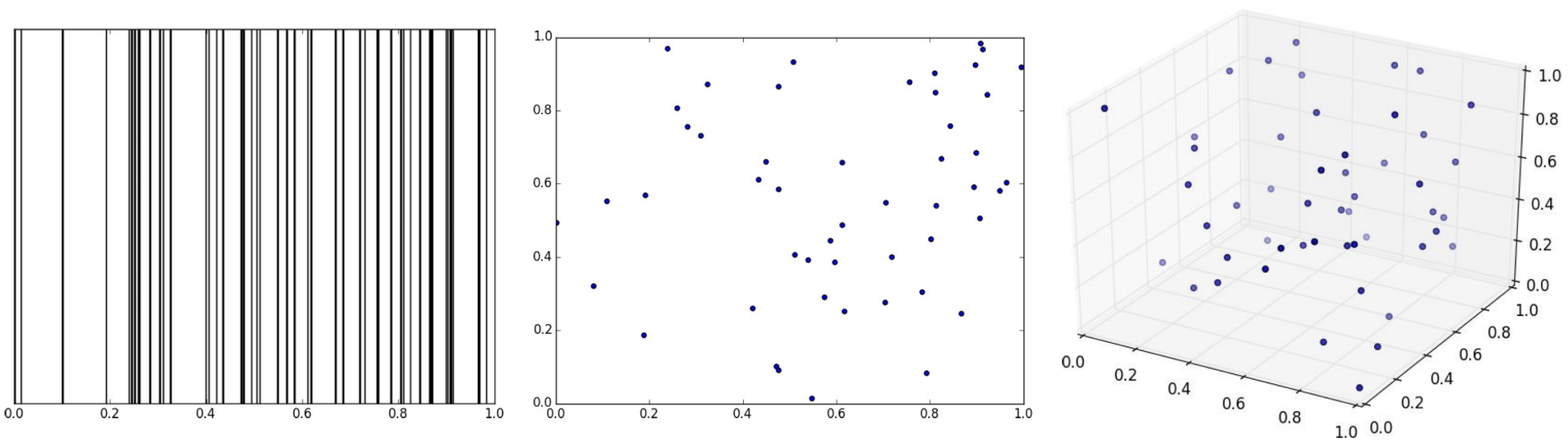
The Curse of Dimensionality (차원의 저주)

- k-최근접 이웃 알고리즘은 고차원 공간에서 문제가 발생합니다. 이는 "차원의 저주"로 불리며, 고차원 공간이 매우 넓다는 사실에 기인합니다.
- 고차원 공간의 점들은 서로 가깝지 않은 경향이 있습니다.
- 이를 확인하는 한 가지 방법은 다양한 차원에서 d-차원 "단위 큐브"에서 점 쌍을 무작위로 생성하고, 그들 사이의 거리를 계산하는 것입니다.

```
100*61

# 6100
```

```
import math

# 2의 250승의 자릿수를 계산
result = math.floor(math.log10(2**250)) + 1
print(result)

# 76
```

Throw 50 random numbers in 1, 2, 3 dimensions (50개의 랜덤 숫자를 1, 2, 3차원 공간에 던지는)



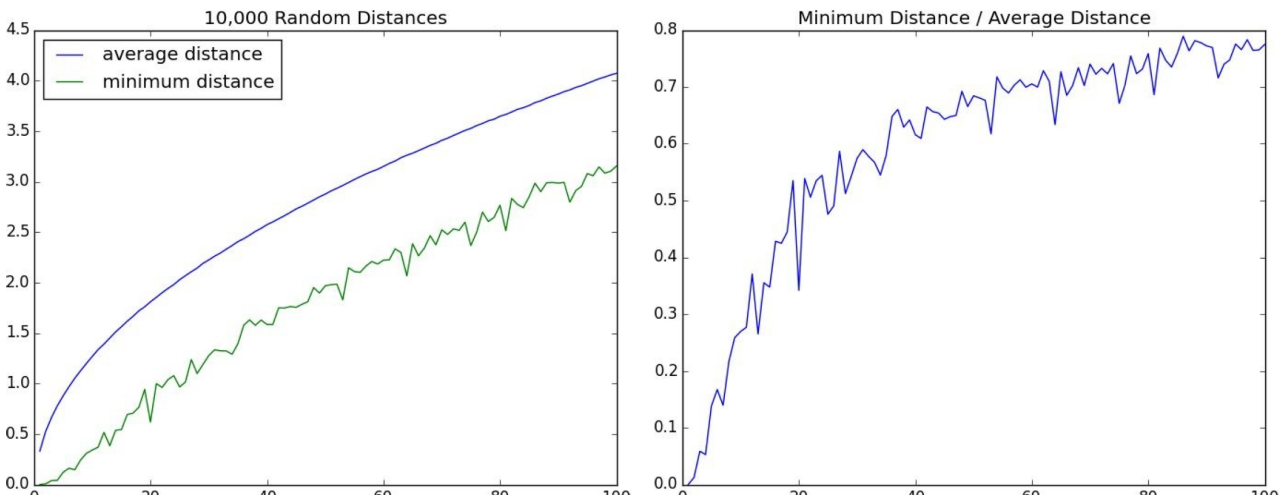

```
def random_point(dim):
    """dim 차원에서 랜덤한 점을 생성합니다."""
    return [random.random() for _ in range(dim)]

def random_distances(dim, num_pairs):
    """주어진 차원에서 num_pairs 만큼의 점 쌍을 무작위로 생성하고 거리를 계산합니다."""
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

```
# 차원을 1부터 100까지 5씩 증가시키면서 설정
dimensions = range(1, 101, 5)

avg_distances = []  # 평균 거리를 저장할 리스트
min_distances = []  # 최소 거리를 저장할 리스트

random.seed(0)  # 재현 가능한 결과를 위해 시드 설정

# 각 차원에 대해 10,000개의 랜덤 점 쌍 생성 및 거리 계산
for dim in dimensions:
    distances = random_distances(dim, 10000)  # 10,000 random pairs
    avg_distances.append(mean(distances))     # 평균 거리를 추적
    min_distances.append(min(distances))      # 최소 거리를 추적
    print(dim, min(distances), mean(distances), min(distances) / mean(distances))
```

```
1 7.947421226228712e-06 0.3310009902894413 2.4010264196729895e-05
6 0.18647467260473205 0.9677679968196268 0.19268530600055306
11 0.315888574043911 1.3334395796543002 0.23689755341281116
16 0.7209190490469604 1.6154152410436047 0.4462747600308797
21 0.9694045860570238 1.8574960773724116 0.5218878240800003
26 1.1698067560262715 2.0632214700056446 0.5669807013122402
31 1.2930748713962408 2.257299829279505 0.5728414340991512
36 1.5123637311959328 2.437670913316559 0.620413413038717
41 1.5514668006745476 2.6039686964057926 0.5958085451703037
46 1.6688006850159558 2.756796053135482 0.6053406392242623
51 2.0135369208019926 2.902997336534375 0.6936061895274667
56 2.1422705294432887 3.0461953095695335 0.7032610557548324
61 2.2891825062886793 3.1783717877656223 0.720237486092828
66 2.3805561409678484 3.305579571524835 0.7201630121006946
71 2.428355816745725 3.4329484139337785 0.7073674066552892
76 2.5356413086431617 3.558475062222762 0.7125640237195596
81 2.682272988673655 3.669873368578009 0.7308897935388364
86 2.8348947533212074 3.779672772114365 0.7500370863415659
```

```
91 3.015796748953059 3.888554628876585 0.7755572537306314
96 2.976216447967502 3.9912782735625743 0.7456800162698157
```

Throw 10,000 distances for every dimension from 1 to 100 (1차원부터 100차원까지 각 차원에서 10,000개의 점 쌍을 무작위로 생성)

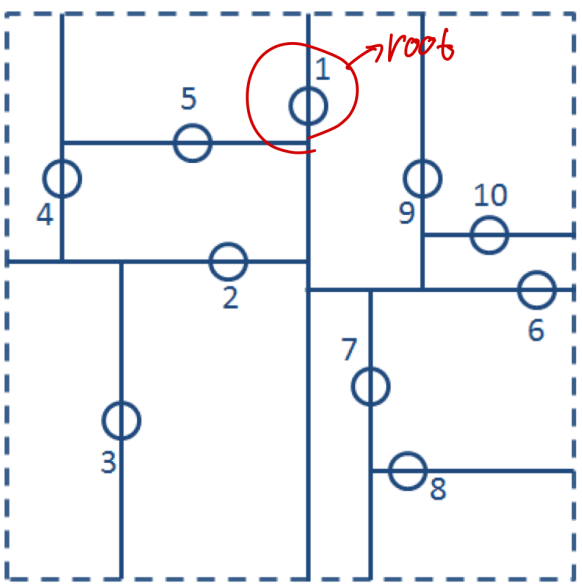


색인(Indexing)에 대한 주의 사항:

- 수백만 개의 데이터를 효율적으로 검색하기 위해서는 색인을 만들어야 합니다.
- 색인이 없으면 학습은 시간이 거의 걸리지 않지만, 예측은 많은 시간이 소요됩니다!

K-d Tree 색인을 위한 단계

- Step 1. 각 차원 d에 대해 {1, ..., k}:
  - 해당 차원의 중간 값을 찾습니다.
  - 중간 값을 기준으로 데이터를 분할합니다.
- Step 2. 데이터가 더 이상 없을 때까지 Step 1을 반복합니다.



- 균등 분할을 사용하여 K-d 트리를 구성할 수 있습니다.
- 이렇게 하면 모든 데이터 포인트를 리프 노드에 남길 수 있으며, 이는 데이터베이스 응용 프로그램에서 주로 사용됩니다.
- K-d 트리는 균형 잡힌 k-차원 이진 탐색 트리(BST)와 유사합니다.
- 우리는 범위 쿼리와 k-최근접 이웃 쿼리에 대해 생각해 볼 수 있습니다.
- 그러나 k가 커지면 차원의 저주를 경험하게 되며, 이는 큰 k에 대해 색인 생성이 거의 불가능함을 의미합니다.

help(neighbors.KNeighborsClassifier)

- Take a look at the parameters: algorithm, p, metric

```
help(neighbors.KNeighborsClassifier)
```

도움말: `neighbors.KDTree`

- `neighbors.KDTree`는 효율적인 K-최근접 이웃(KNN) 및 범위 쿼리(range query)를 수행하는 데 사용되는 데이터 구조입니다. 아래는 주요 메서드입니다:

```
tree.query(...)    # knn query
tree.query_radius(...)   # range query
```

KDTree 사용 방법

- KDTree를 실제 애플리케이션에서 사용하려면 이를 구성한 후 **pickle**로 저장해야 합니다.
- `pickle`은 Python 객체를 직렬화하여 파일로 저장하거나 네트워크를 통해 전송하는 데 사용됩니다.
- `unpickle`은 직렬화된 객체를 다시 역직렬화하여 Python 객체로 복원합니다.

예시 애플리케이션

- **knn query**: 내 위치에서 가장 가까운 주유소 k개 찾기
- **range query**: 도보로 5분 거리 이내의 레스토랑 찾기

```
from sklearn.neighbors import KDTree
import numpy as np
np.random.seed(0)
X = np.random.random((10000000, 2))  # 10 million points in 2 dimensions
%time tree = KDTree(X, leaf_size=1)

# Wall time: 45 s
```

```
from sklearn.neighbors import KDTree
import numpy as np
np.random.seed(0)
X = np.random.random((1000000000, 2))  # 1000 million points in 2 dimensions
%time tree = KDTree(X, leaf_size=1)
```

```
# 특정 점에 대해 k-최근접 이웃 쿼리 실행 및 시간 측정
%time dist, ind = tree.query([locations[0]], k=5)

# 가장 가까운 5개의 이웃의 인덱스 출력
print(ind)

# 가장 가까운 5개의 이웃까지의 거리 출력
print(dist)
```

```
Wall time: 997 µs
[[      0  498131 9686005  299237 8223653]]
[[0.         0.00014007 0.00014578 0.00016266 0.00019137]]
```

Compare to Homework #2's result

- CPU times: user 401 ms, sys: 264 ms, total: 665 ms

```
# 특정 점에 대해 반경 0.0005 내의 점 개수 계산
count = tree.query_radius([locations[0]], r=0.0005, count_only=True)
print(count)

# [9]
```

```
# 특정 점에 대해 반경 0.0005 내의 점들 검색
indices = tree.query_radius([locations[0]], r=0.0005)[0]
```

```
[[0.5486868  0.71490509]
 [0.54875985 0.71498733]
 [0.54850997 0.71539435]
 [0.5488135  0.71518937]
 [0.54892172 0.71510044]
 [0.54897222 0.71522494]
 [0.54902884 0.71487778]
 [0.54891016 0.7152985 ]
 [0.54892252 0.71534665]]
```

Use Scikit Learn KNN

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

# 학습 데이터
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]

# K-최근접 이웃 분류기 구성 및 학습
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X, y)

# 새로운 데이터 포인트 [1.1]에 대한 예측
prediction = neigh.predict([[1.1]])
print("Prediction for [1.1]:", prediction)

# 새로운 데이터 포인트 [0.9]에 대한 예측 확률
prediction_proba = neigh.predict_proba([[0.9]])
print("Prediction probabilities for [0.9]:", prediction_proba)
```

```
[0]
[[0.66666667 0.33333333]]
```

```
# cities 데이터를 X와 y로 분리
X, y = zip(*[(city[:2], city[2]) for city in cities])

# K-최근접 이웃 분류기 구성 및 학습
neigh = KNeighborsClassifier(n_neighbors=3)
```

```
neigh.fit(X, y)

# 새로운 데이터 포인트 [-86, 33]에 대한 예측
prediction = neigh.predict([[-86, 33]])

# 새로운 데이터 포인트 [-86, 33]에 대한 예측 확률
prediction_proba = neigh.predict_proba([[-86, 33]])

# 새로운 데이터 포인트 [-86, 33]에 대한 가장 가까운 5개의 이웃과 거리
distances, indices = neigh.kneighbors([[-86, 33]], 5, True)
```

```
['Python']
[[0.         0.66666667 0.33333333]]
(array([[0.94000591, 1.69615578, 3.19069829, 3.22946504, 4.1401154 ]]), array([[ 0, 24, 64,  1,
```