Nearest Neighbor Model

Hyunjoong Kim

soy.lovit@gmail.com

github.com/lovit

Nearest Neighbor Models

- 최인접이웃 (nearest neighbors) 기반 모델은 query vector q 와 벡터 공간에서 가까운 k 개의 점들을 이용하여 분류/회귀 문제를 합니다.
- 모델, 데이터 분포에 대한 가정이 없는 non-parametric models 입니다.

Nearest Neighbor Models for Classification

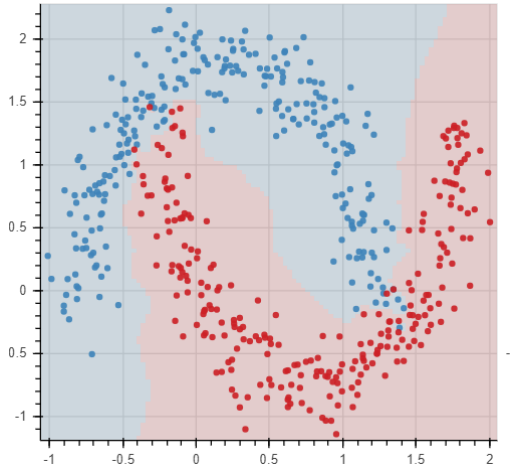
- 최인접이웃 (nearest neighbors) 기반 모델은 분류 문제에 이용할 수 있습니다.

$$f(q) = F\left(\sum_{x_i \in NN(q)} w(x_i, q) y_i\right)$$

$w(x_i, q)$: weight between neighbors and query

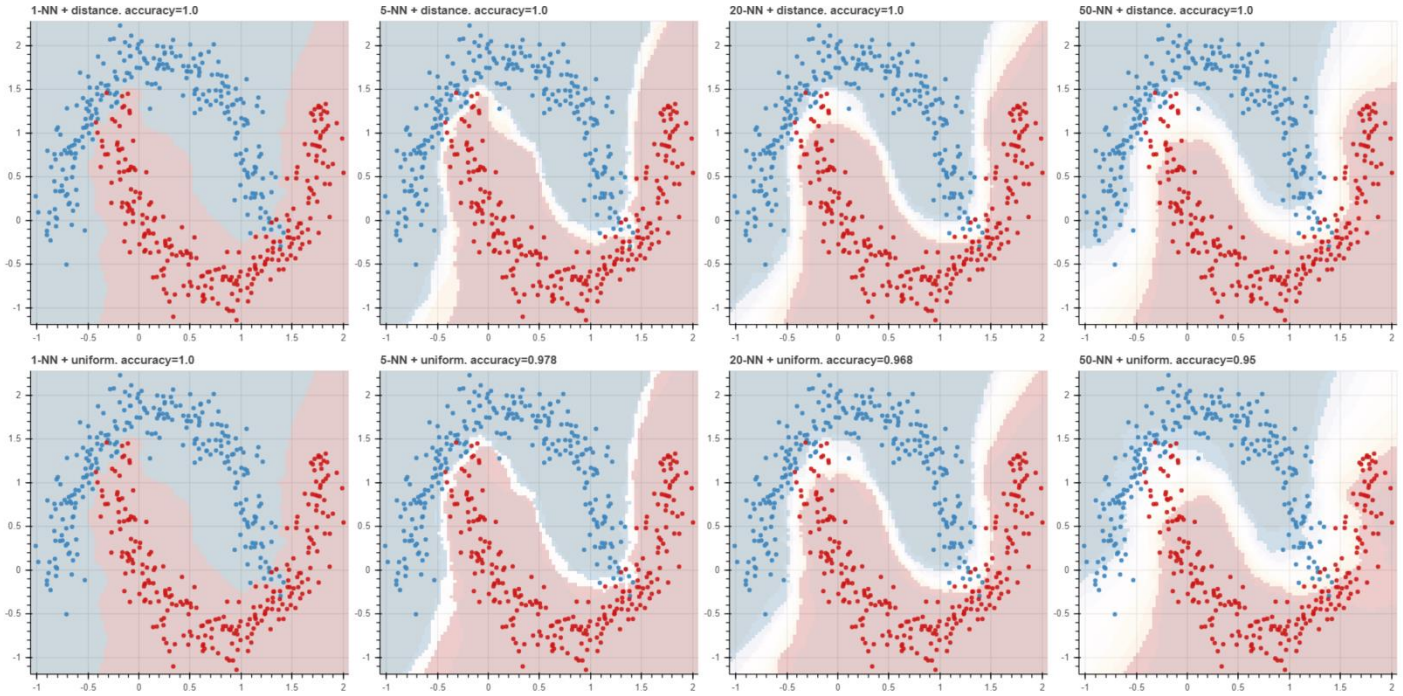
such as $\frac{1}{|NN(q)|}, \frac{1}{d(x_i,q)}$ or $\exp(-d(x_i, q))$

- Query vector q 와 가까운 k 개의 레이블로 q 의 레이블을 선택합니다.



Nearest Neighbor Models for Classification

- k 와 weights 방식에 따라 경계면의 양상이 달라집니다.
  - k 가 커질수록 모델은 over-fitting 에서 under-fitting 방향으로 학습합니다.
  - 최인접이웃의 레이블의 분산을 이용하면 예측의 confidence 도 계산할 수 있습니다.



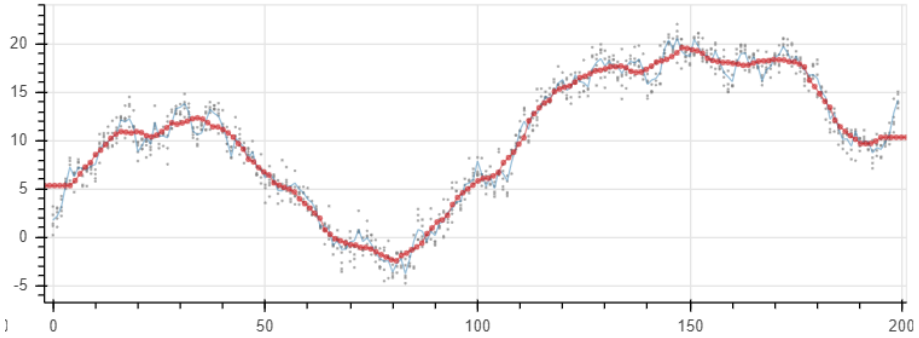
Nearest Neighbor Models for Regression

- 최인접이웃 (nearest neighbors) 기반 모델은 회귀 문제에도 이용할 수 있습니다.

$$f(q) = F\left(\sum_{x_i \in NN(q)} w(x_i, q) y_i\right)$$

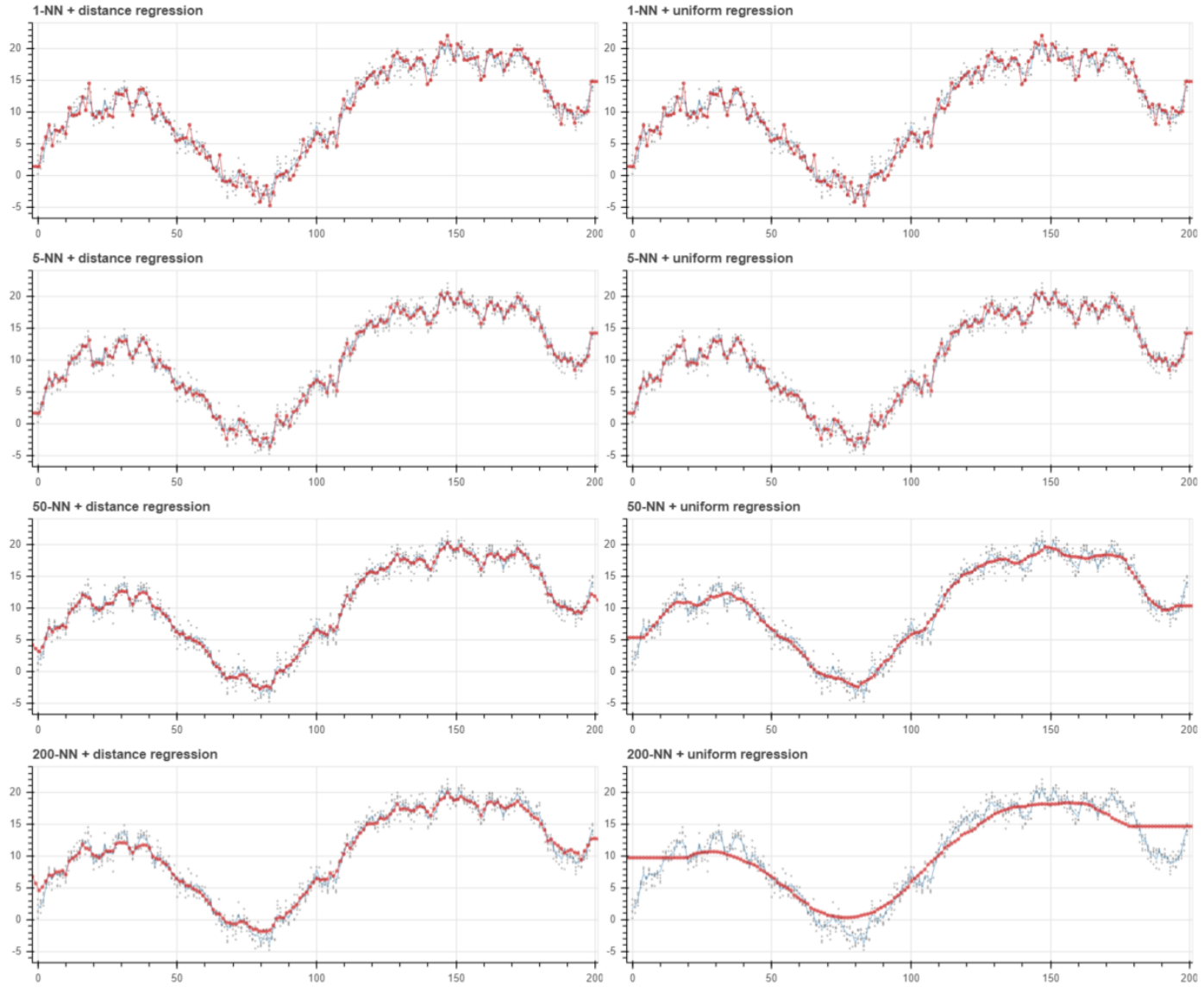
$w(x_i, q)$: weight between neighbors and query

such as $\frac{1}{|NN(q)|}, \frac{1}{d(x_i,q)}$ or $\exp(-d(x_i, q))$



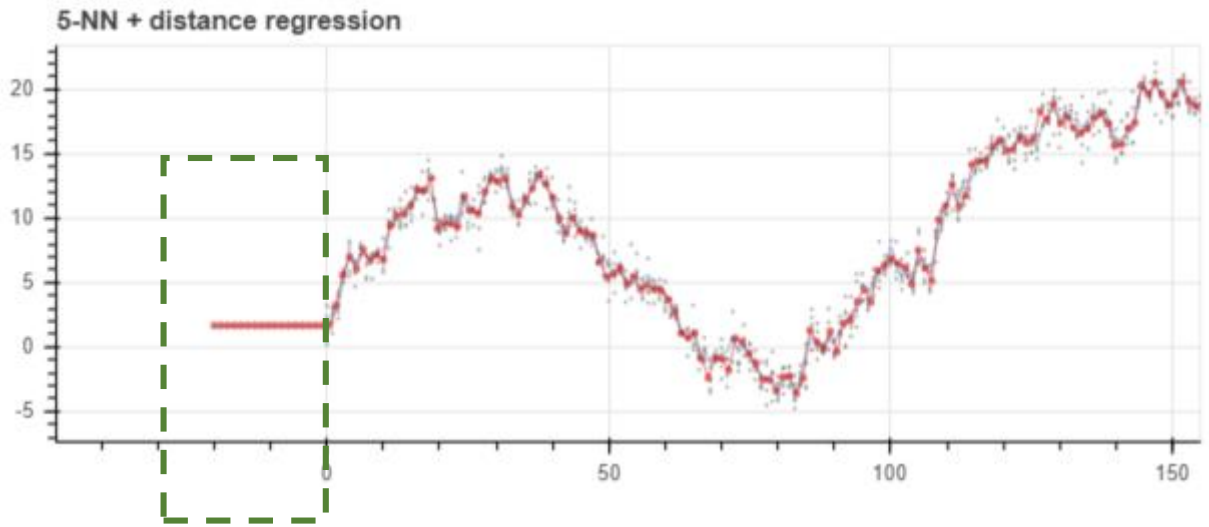
Nearest Neighbor Models for Regression

- 회귀 문제에서도 k 와 weights 방식에 따라 추세선의 경향이 달라집니다.



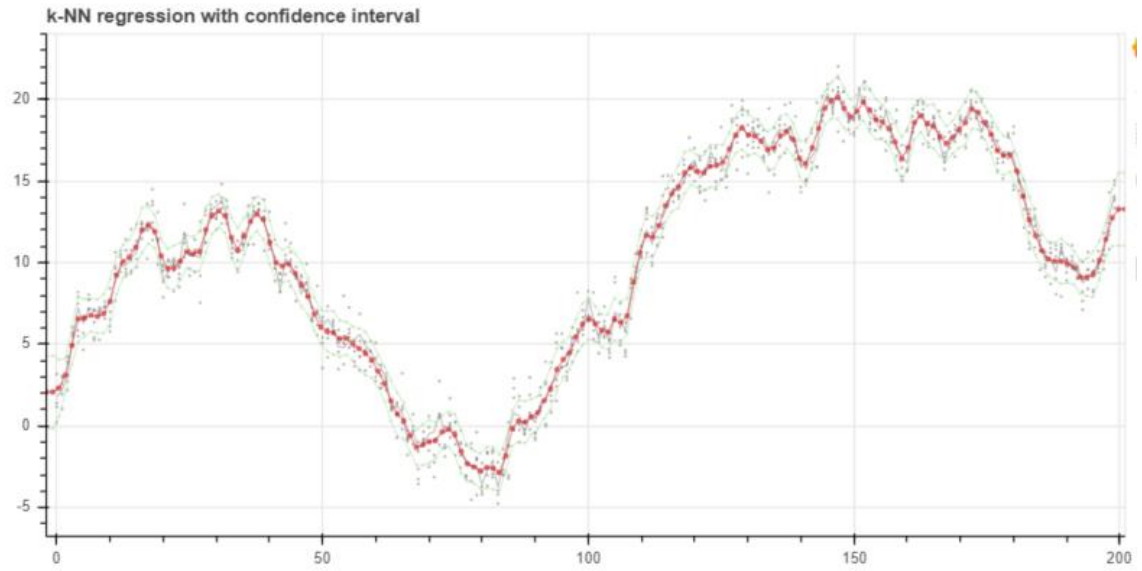
Extrapolation

- 학습데이터 (reference data) 가 존재하지 않는 구간에서는 제대로된 예측이 어렵습니다.
  - 각 모델마다 extrapolation 을 하는 경향이 다릅니다 (bias, inductive bias)



Nearest Neighbor Models for Regression

- $\text{std}(y_i \mid y_i \in NN(q))$ 을 이용하면 신뢰 구간도 추정할 수 있습니다.
  - $\left(\frac{1}{|NN(q)|}\sum_i w(q, y_i)y_i - \text{std},\ \frac{1}{|NN(q)|}\sum_i w(q, y_i)y_i + \text{std}\right)$



Nearest Neighbor Models

- Parametric models 은 함수의 형태를 명시적으로 가정합니다.
  - $y = f(x) = A^T x + b$ 처럼 $x \rightarrow y$ 의 관계에 대한 형태를 가정하고,
    이를 설명할 수 있는 패러매터 A, b 를 학습합니다.
- Non-parametric models 은 명시적인 함수 형태를 가정하지 않습니다.
  - 데이터의 분포, $x \rightarrow y$ 관계에 대한 형태를 가정하지 않습니다.
  - 임의의 형태의 모든 함수를 표현할 수 있다는 의미입니다.

Nearest Neighbor Models

- 데이터가 자주 업데이트 되거나, 분포 가정이 어려울 때 쓴다.
  - 순차적으로 입력되는 데이터의 최근 k 개만을 이용하여 다음 값을 예측한다면 최근의 $k+1$ 부터는 데이터를 제거해도 됩니다. 매번 모델을 만드는 것보다 최인접이웃을 기반으로 예측을 수행하는 것이 효율적입니다.
  - 데이터의 분포를 가정할 수 없거나, 매우 복잡한 분포를 지닐 때에도 분류, 회귀 예측이 가능합니다.

Nearest Neighbor Models

- 최인접이웃 모델을 이용하기 위해서는 두 가지 어려움을 해결해야 합니다.
  - 데이터의 representation 이 좋아야 합니다. Reference data X 에서 query vector q 와 비슷한 점은 벡터 간 거리도 가까워야 합니다.
  - 하나의 q 에 대하여 reference dataset 과의 거리를 계산해야 하기 때문에 $O(n)$ 의 예측계산비용이 듭니다.

Nearest Neighbor Models

- Representation 은 다른 모델을 이용하여 학습/변경할 수 있습니다.
  - MNIST 를 분류하는 CNN nets 은 앞 부분에서 이미지를 linear separable 한 새로운 벡터로 representation 을 변경합니다.

Trained CNN encoder

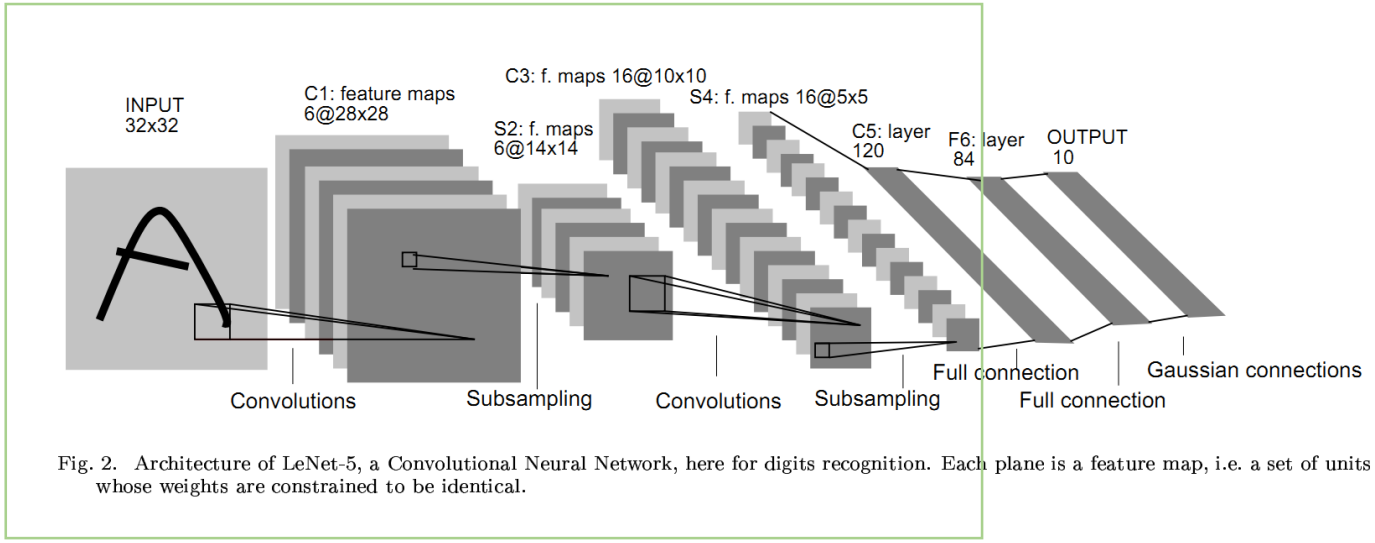
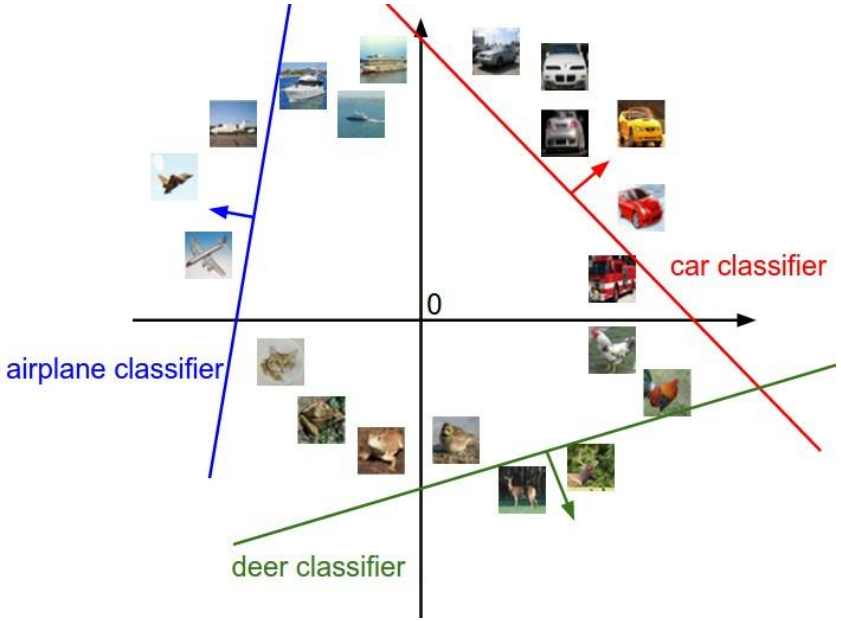


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Nearest Neighbor Models

- 이미지 분류의 관점에서 이들을 잘 구분할 수 있는 형태로 변경된 representation 을 이용하면, 이미지 분류의 관점에서 비슷한 이미지들을 검색하는 이미지 검색기를 만들 수 있습니다.



http://cs231n.github.io/linear-classify/

Nearest Neighbor Models

- Representation + Distance metric 조합을 생각해야 합니다.
  - 소프트맥스를 이용한 경우라면 각 클래스별로 데이터가 한 방향에 모여있습니다.
  - 이때는 벡터의 크기 차이의 영향력이 큰 Euclidean 보다 방향성의 영향력이 큰 Cosine 이 더 적합합니다.

Trained CNN encoder

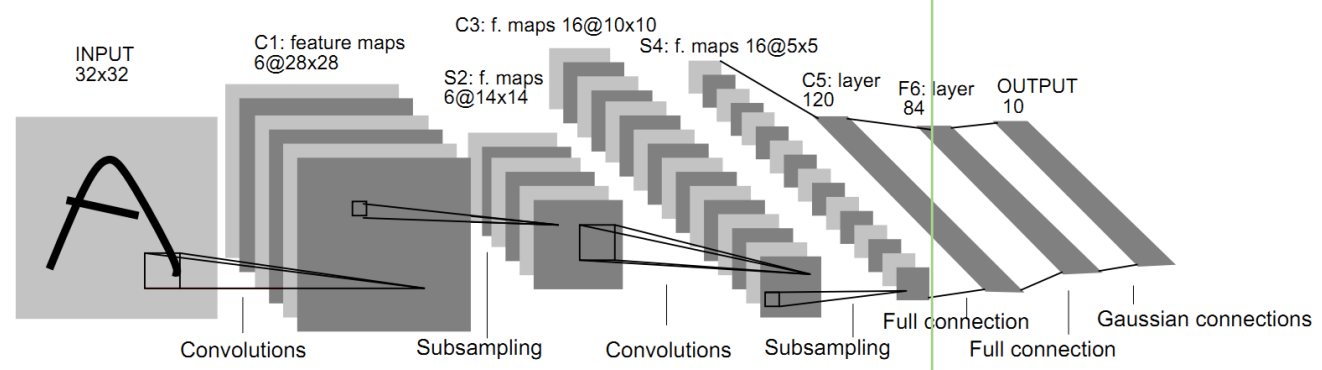


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Nearest Neighbor Models

- 각 변수마다 스캐일이 다른 경우에는 거리 척도를 정의하기 어렵습니다.
  - 테이블 형식의 데이터들은 Cosine, Euclidean 으로 거리를 측정할 수 있는 수준으로 데이터의 representation 을 변경해야 최인접이웃 모델을 이용할 수 있습니다.
  - 다른 과업을 위해 학습된 모델의 일부분을 encoder 로 이용하는 경우들이 많습니다.

Nearest Neighbor Models

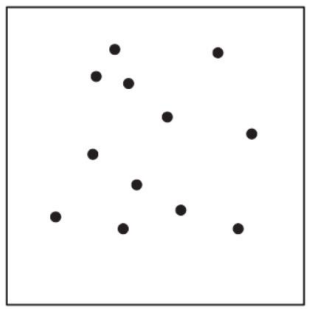
- 최인접이웃의 계산 시간은 $O(n)$ 이 아닙니다.
  - Approximated Nearest Neighbor Search (ANNS) 분야는 주어진 query q 의 최인접 이웃이 될 법한 점들만 골라서 그 점들에 대해서만 실제 거리 계산을 합니다.
  - 근사방법이기 때문에 정확한 최인접이웃을 찾지 못하기도 합니다. 하지만 거리 계산의 횟수를 크게 줄입니다. 이들을 인덱서 (Indexer) 라 하며, 학습이 잘 된 경우에는 reference data 의 크기와 관계없이 거의 일정한 속도로 빠르게 최인접이웃을 찾을 수 있습니다.

Approximated Nearest Neighbor Search

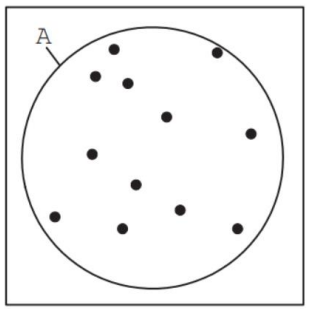
- 다양한 방법이 있지만, 대체로 세 가지 방법이 기초가 됩니다.
  - Tree based indexer (k-d tree, ball tree)
  - Graph based indexer (NN-descent)
  - Hashing based indexer (LSH)
- 최근에 제안된 방법론도 위의 방법들을 알면 이해하기 쉽습니다.
  - HNSW (2016), Faiss (2017), Neighborhood Graph and Tree, NGT (2018)

Ball Tree

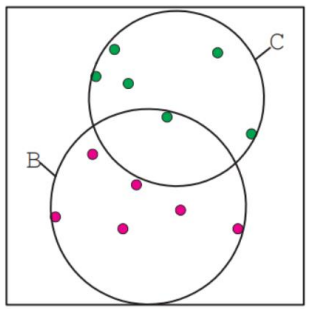
- 공간을 여러 개의 구형으로 나눈 뒤, 이들을 계층적 구조로 연결합니다.
  - BallTree 는 scikit-learn 에서 구현체를 제공합니다.



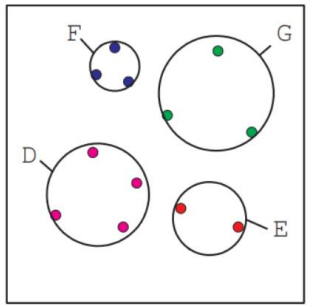
1a. A dataset



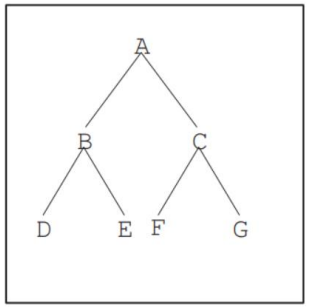
1b. Root node



1c. The 2 children



1d. The 4 grandchildren

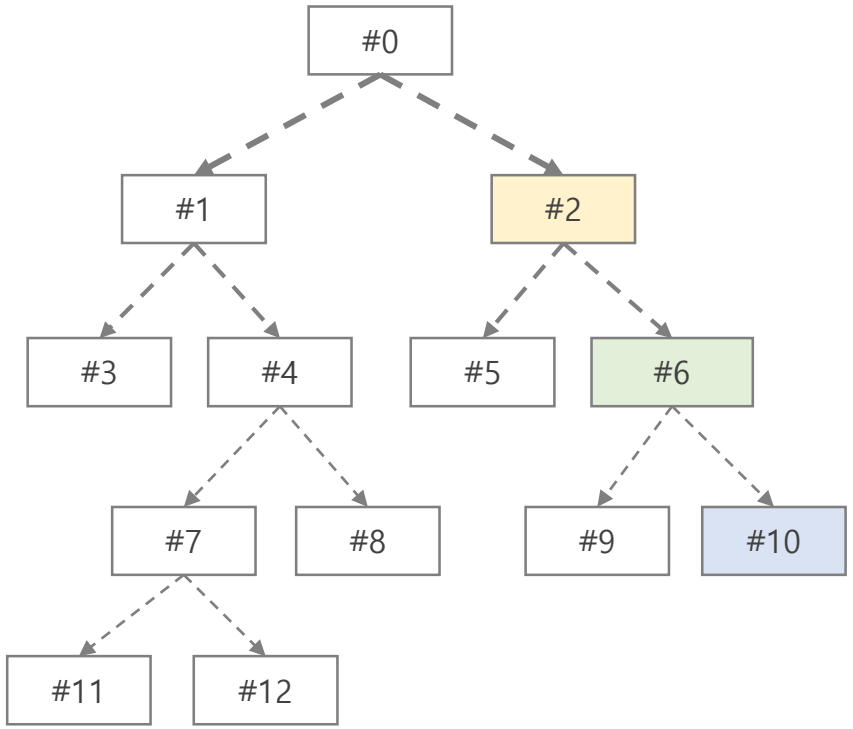


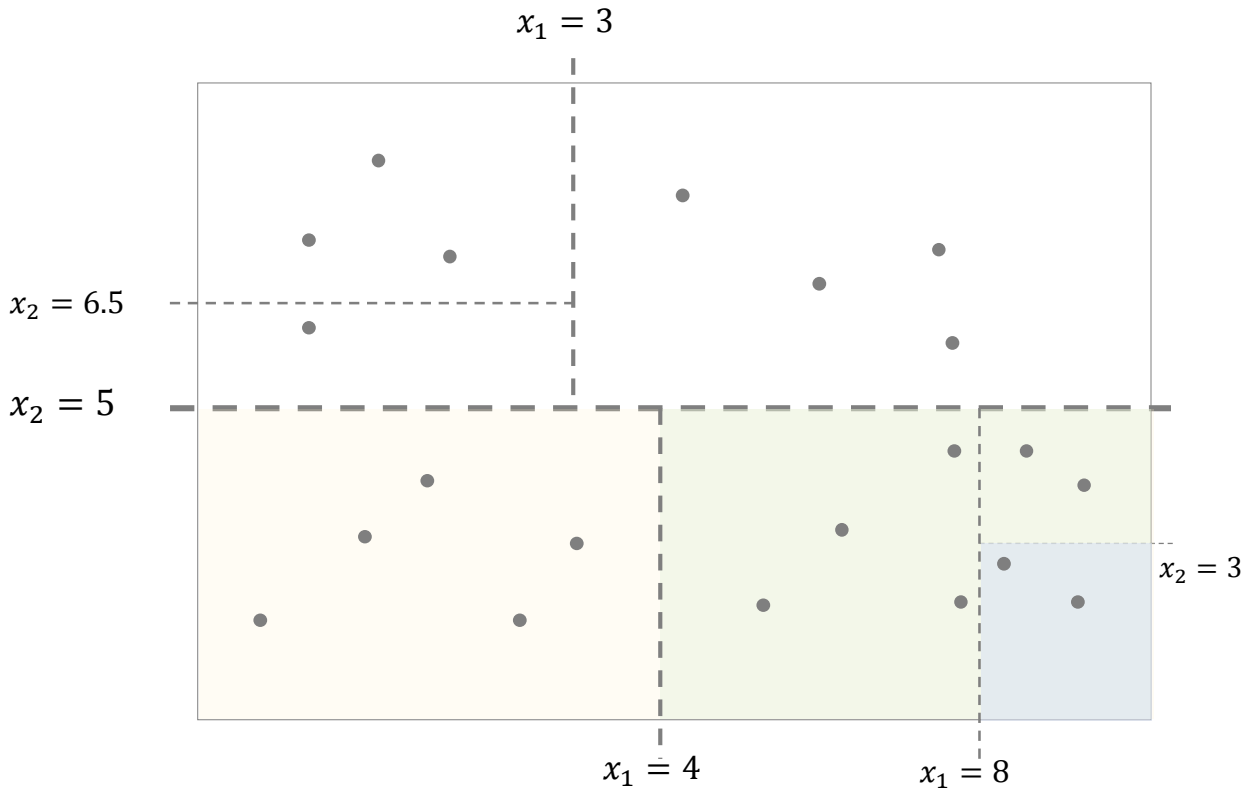
1e. The internal tree structure

Liu, T., Moore, A. W., & Gray, A. (2006). New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, *7*(Jun), 1135-1158.

K-D Tree

- K-D Tree 는 공간을 더 작은 부분공간으로 재귀적으로 구분합니다. 대체로 벡터 공간의 차원이 10 보다 클 경우에는 큰 이득이 없다고 알려져 있습니다.





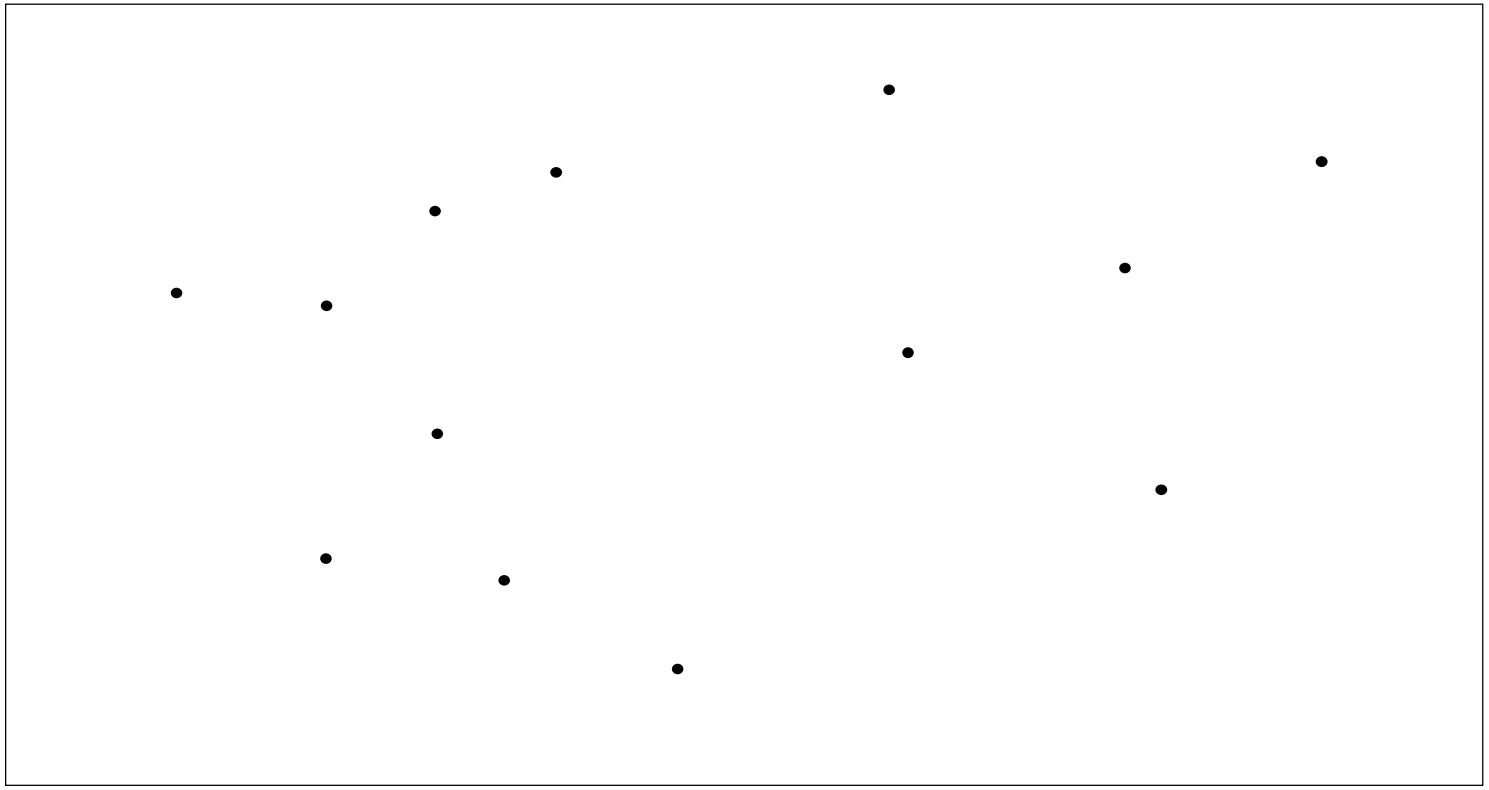
Graph based neighbor search

- k-nearest neighbor search 는 점들 간 거리의 “순서”를 학습해야 합니다.
  - 트리 기반 방법은 공간을 분할하며, 해당 공간은 “거리”를 학습합니다.
  - 밀도가 다른 공간에서 성능의 차이가 발생합니다.
- 최인접그래프에는 점들 간의 가까운 순서가 학습되어 있습니다.
  - 최인접이웃 그래프와 랜덤 그래프를 함께 이용하여 인덱싱을 할 수 있습니다.

Graph based neighbor search

Concept:

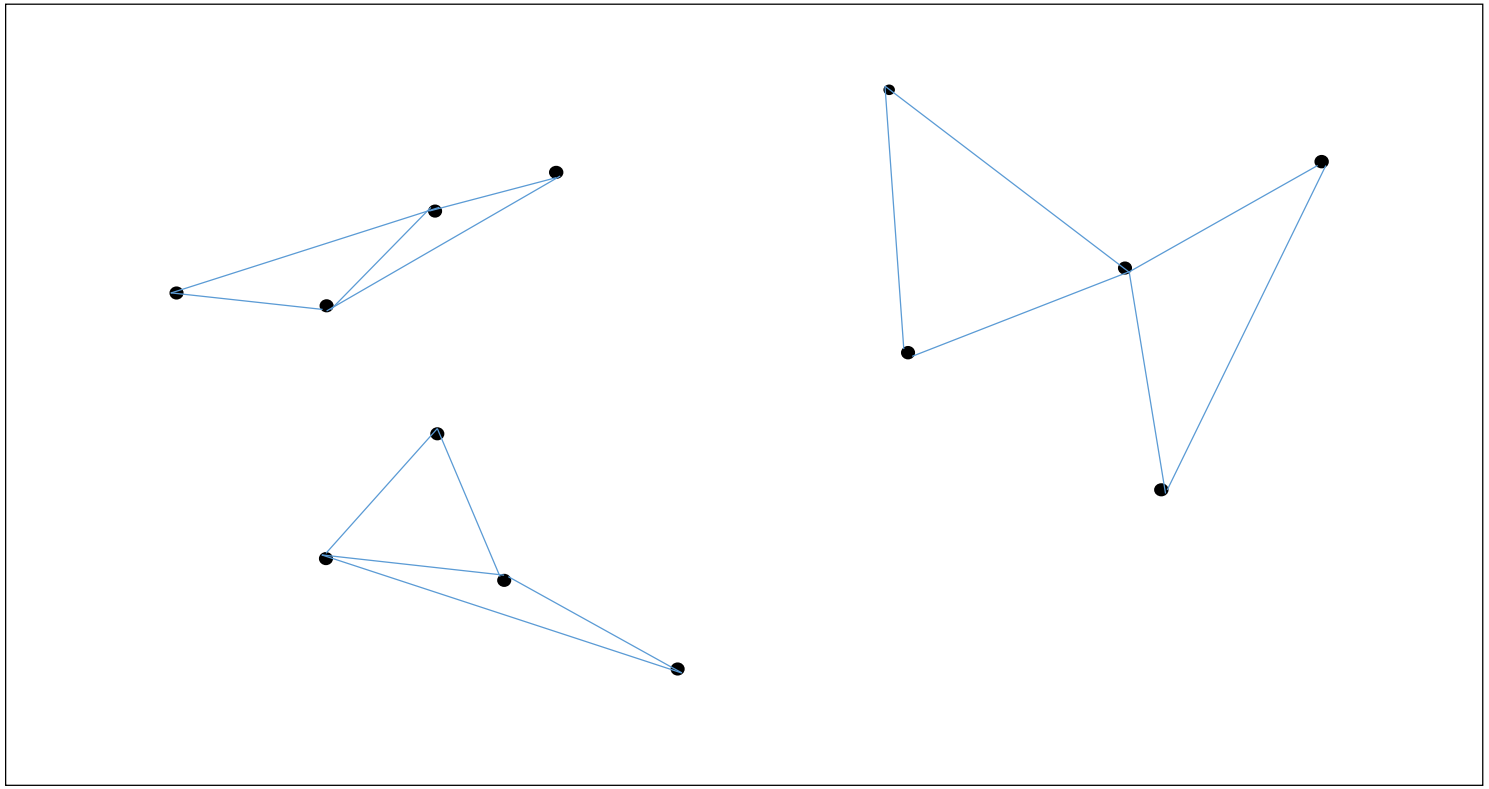
combining nearest neighbor(NNG) and random neighbor graph(RNG)



Graph based neighbor search

combining nearest neighbor(NNG) and random neighbor graph(RNG)

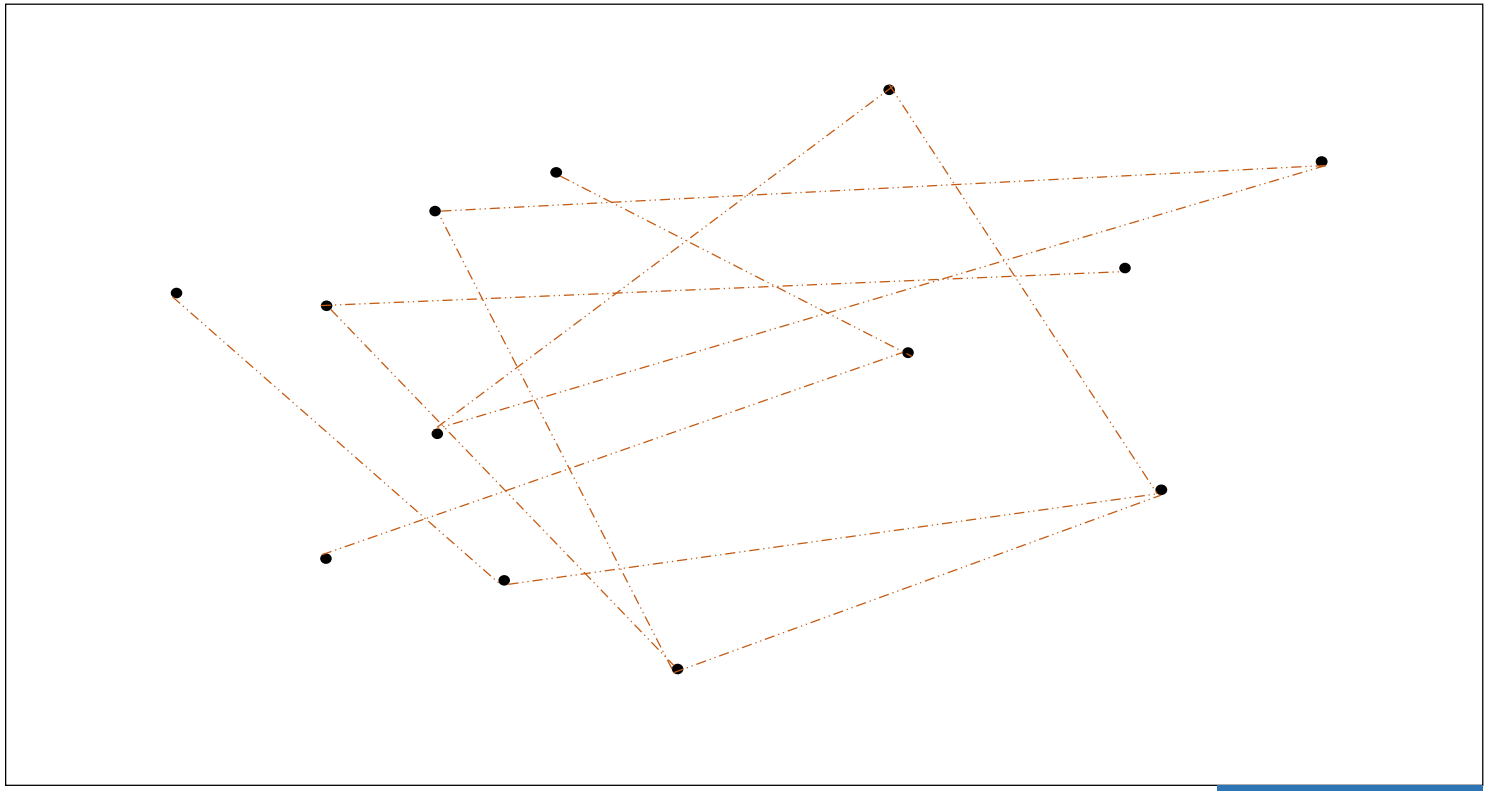
2-NNG: find two nearest neighbors and link them



Graph based neighbor search

combining nearest neighbor(NNG) and random neighbor graph(RNG)

1-RNG: for all vertexes, link them with randomly chosen 1 point

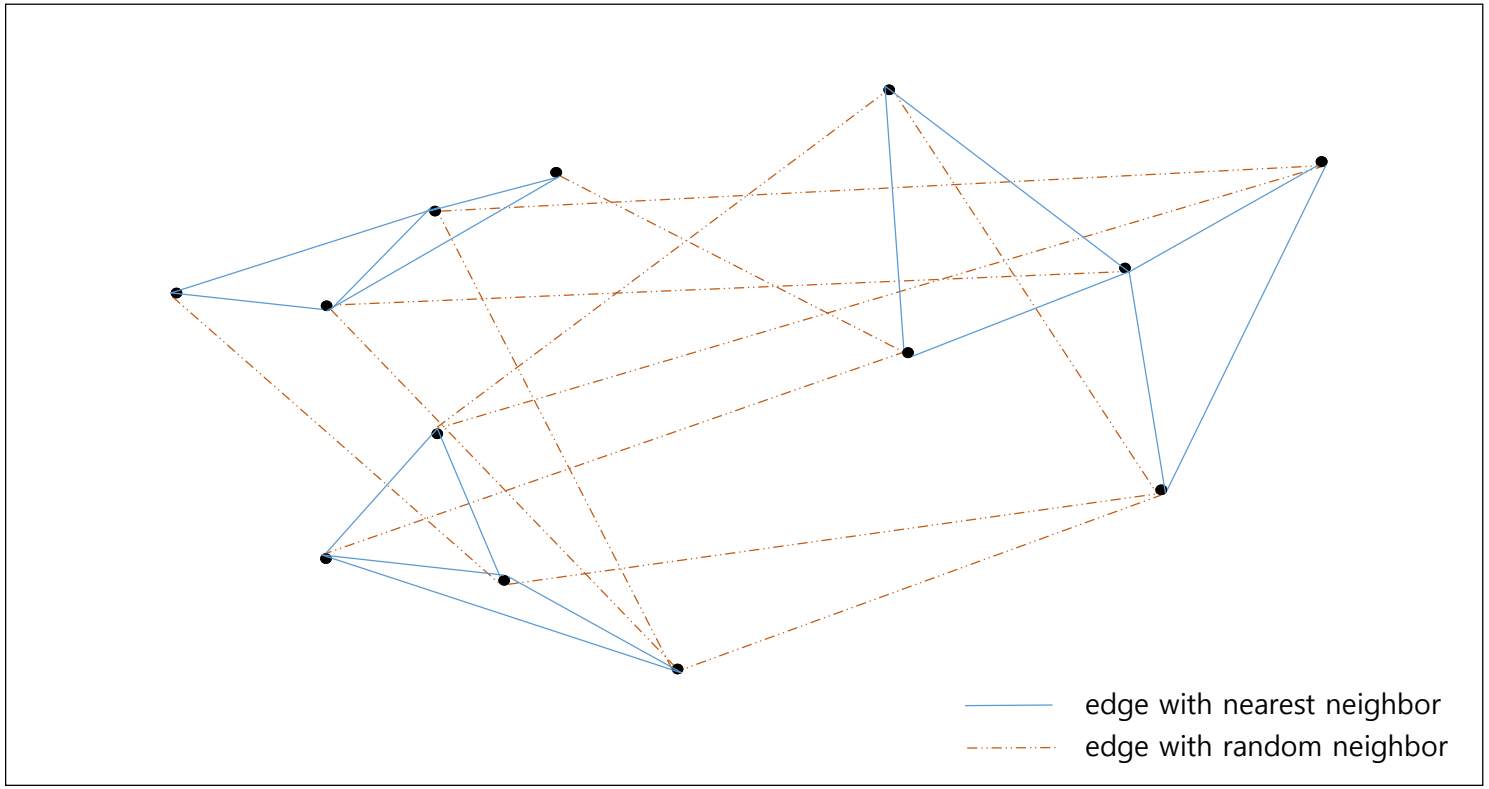


connectivity

Graph based neighbor search

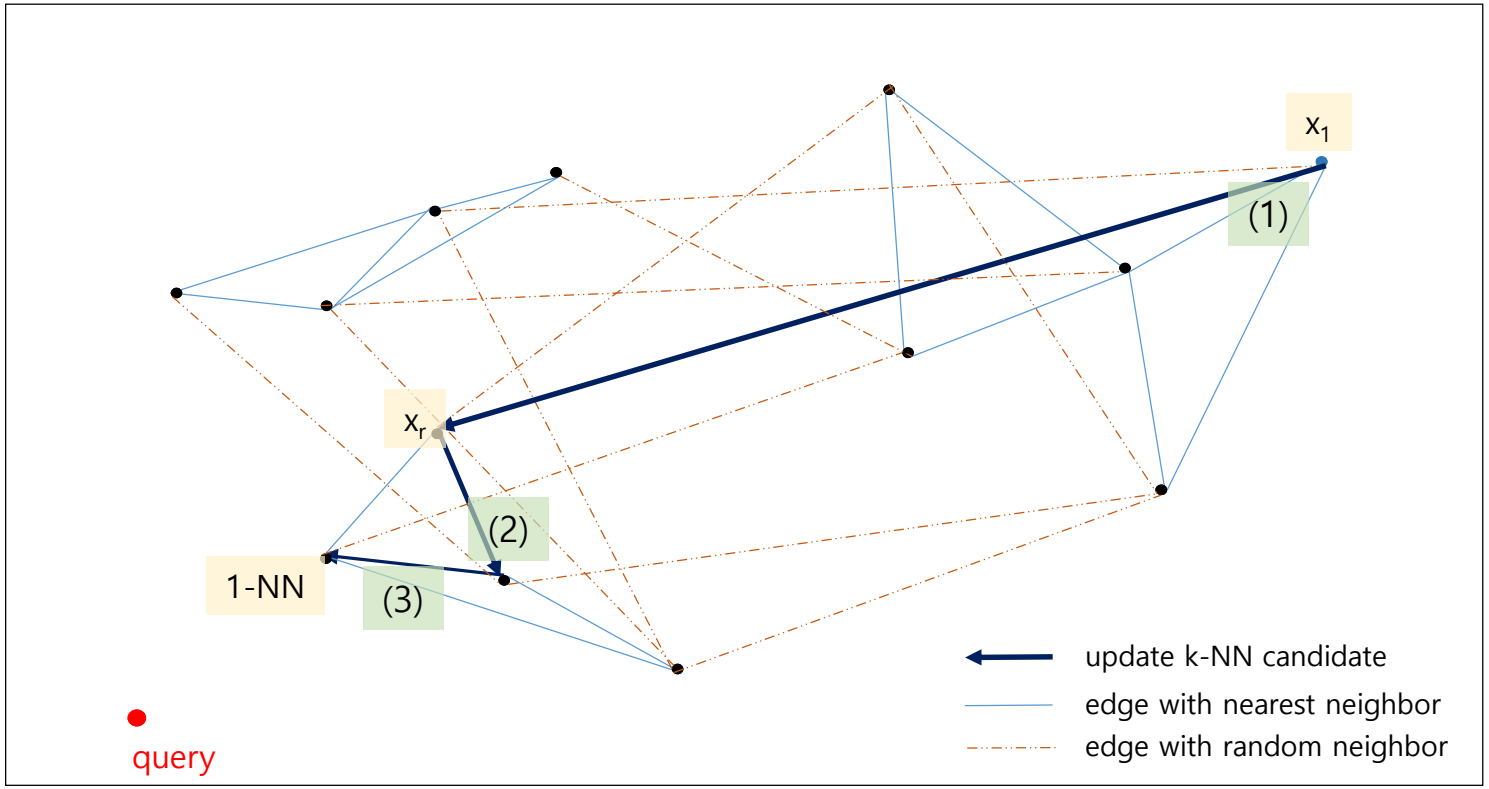
Concept:

combining nearest neighbor(NNG) and random neighbor graph(RNG)



Graph based neighbor search

Search (query processing)



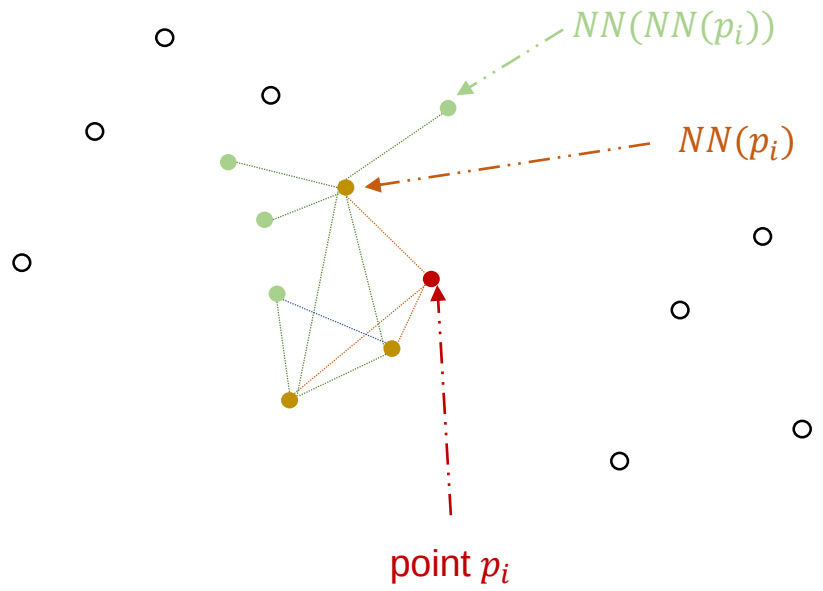
NN-descent

- 그러나 한 번은 최인접이웃 그래프를 만들어야 합니다.
  - 인덱서는 하나의 쿼리가 주어졌을 때 최인접이웃을 탐색하기 위한 방법이며 비슷한 쿼리가 여러 번 입력되는 상황을 가정합니다.
  - 최인접이웃 그래프 생성은 한 점에 대한 단 한번의 최인접이웃 검색입니다.
- NN-descent 는 최인접이웃 그래프 생성 방법입니다.
  - 거리 척도와 관계없이 이용할 수 있으며, 구현체는 최인접이웃 검색 기능도 제공합니다.

Dong, W., Moses, C., & Li, K. (2011, March). Efficient k-nearest neighbor graph construction for generic similarity measures., WWW2011 (pp. 577-586). ACM.

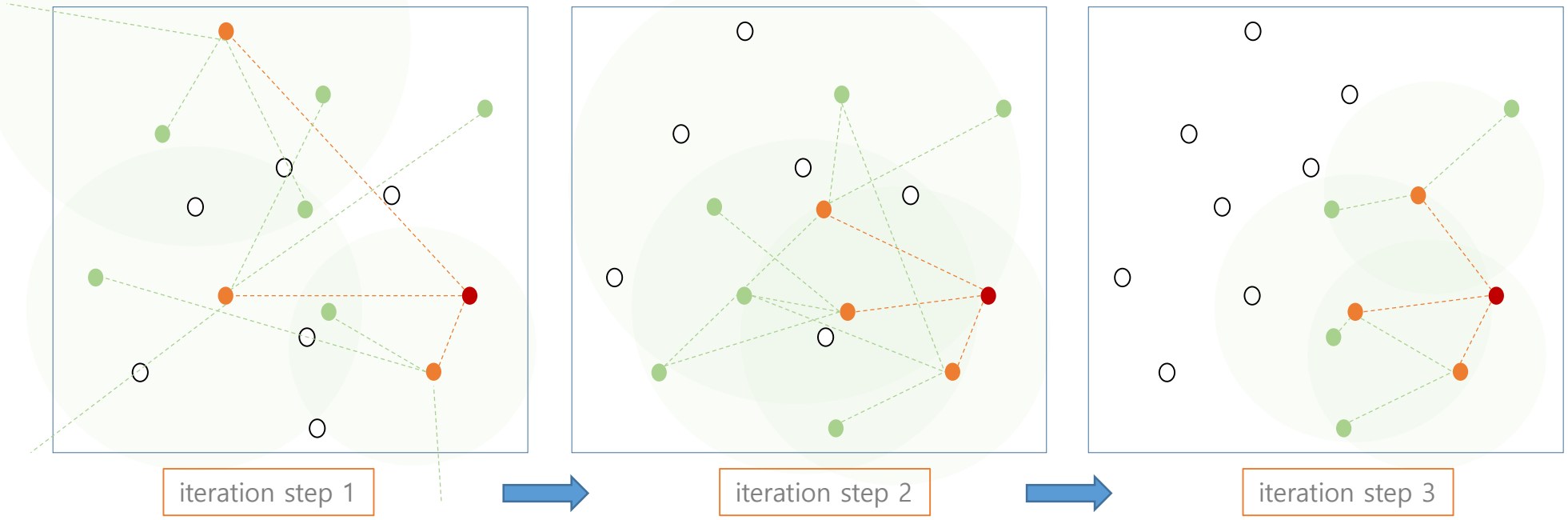
NN-descent

- “이웃의 이웃은 이웃일 가능성이 높다”는 사실을 이용합니다.
  - 최인접이웃의 근사 그래프를 이용하여 최인접이웃의 정확도를 개선합니다.



NN-descent

- 매 반복마다 점진적으로 최인접이웃이 개선됩니다.



NN-descent

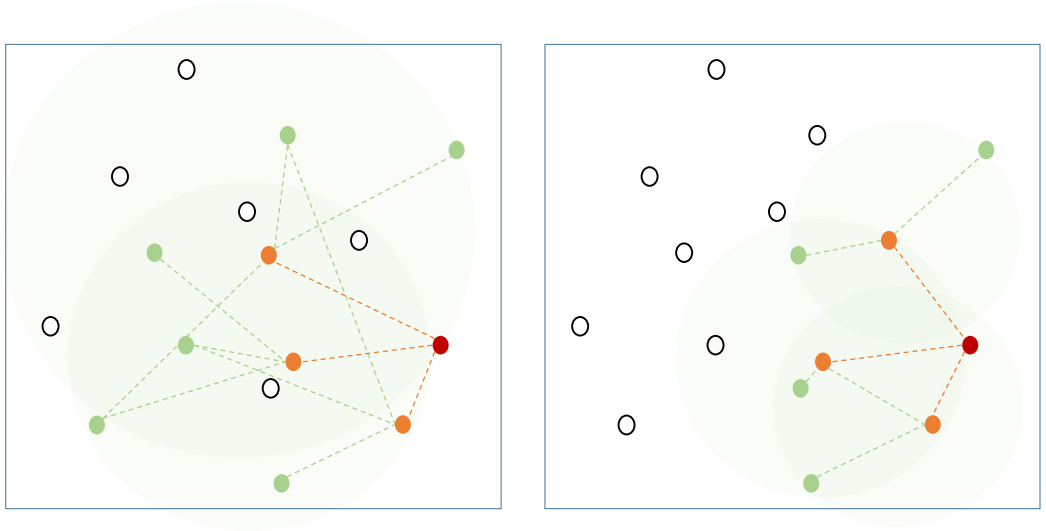
Algorithm 1: NNDescent

Data: dataset V, similarity oracle σ, K
Result: K-NN list B
begin
  $B[v] \longleftarrow \textsc{Sample}(V, K) \times \{\infty\}, \quad \forall v \in V$
  loop
    $R \longleftarrow \textsc{Reverse}(B)$
    $\overline{B}[v] \longleftarrow B[v] \cup R[v], \quad \forall v \in V;$
    $c \longleftarrow 0$ //update counter
    for $v \in V$ **do**
      for $u_1 \in \overline{B}[v], u_2 \in \overline{B}[u_1]$ **do**
        $l \longleftarrow \sigma(v, u_2)$
        $c \longleftarrow c + \textsc{UpdateNN}(B[v], \langle u_2, l \rangle)$
    return B **if** $c = 0$

function $\textsc{Sample}(S, n)$
return Sample n items from set S

function $\textsc{Reverse}(B)$
begin
  $R[v] \longleftarrow \{u \mid \langle v, \cdots \rangle \in B[u]\} \quad \forall v \in V$
  return R

function $\textsc{UpdateNN}(H, \langle u, l, \ldots \rangle)$
Update K-NN heap H; return 1 if changed, or 0 if not.



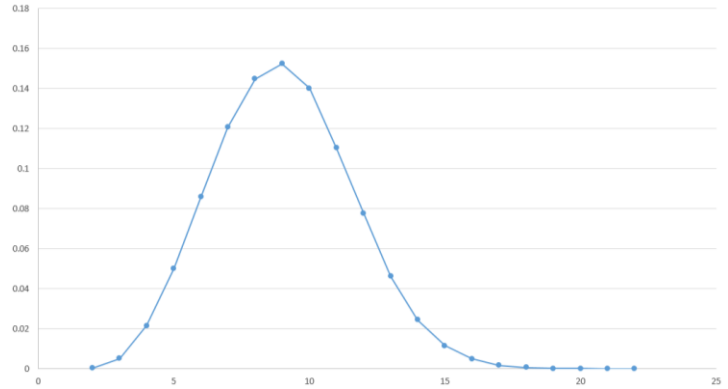
- (n, k) 크기의 B 에 대한 메모리만 이용하며,
- 최대 $(k + k^2) \times \#iters$ 의 거리 계산을 합니다.
  $k \ll n$ 이면 $k^2 \times \#\, iters \ll n^2$ 입니다.

NN-descent

- 새로운 Query point q 에 대해서는 인덱싱이 되어 있는 nearest neighbors 와 random neighbors 의 neighbors 중에서 q 와 가까운 점들을 업데이트 합니다.
  - 이웃이 개선되지 않거나 최대 반복횟수에 도달하면 업데이트를 중지합니다.

Graph based neighbor search

- 그래프 기반 방법의 장점은 일정한 query time 입니다.
  - 트리 기반 방법은 밀도가 높은 지역에서 여러 번의 거리 계산을 해야 합니다.
  - Stanley Milgram's 6 degree of separation 처럼 그래프를 이용할 경우 일정한 반복 횟수 안에 최인접이웃을 찾을 가능성이 높습니다.



Coverage of iteration steps (# of hobs)

Hash based Indexer

- 다양한 방법들이 제안되었으나 고차원의 데이터에 대해서는 hashing 기반 방법이 가장 널리 이용됩니다.
  - B+ tree, KD tree 와 같은 트리기반 방법들은 10 차원 이상이 되면 인덱싱 효과가 거의 없음이 증명되었습니다.
  - Random Projection 에 기반한 Locality Sensitive Hashing (LSH) 방법이 고차원 데이터의 neighbor search 를 위해 주로 이용됩니다.

Random Projection

- 고차원 벡터 u, v 간의 거리를 보존하는 저차원 벡터 x, y 를 학습합니다.

$$x = Mv$$

where $x \in R^k$, $v \in R^N$, $M \in R^{(k \times N)}$ and $k \ll N$

  - M 은 column 의 크기가 1 인 unit vector 이며, 임의로 생성됩니다.
  - Johnson-Linderstrauss Lemma 에 의하면 임의로 생성된 M 의 column 은 거의 직교 (almost orthogonal) 입니다.

Random Projection

$$x = Mv, y = Mu$$

$$x^T y = (Mv)^T (Mu) = v^T M^T Mu$$

$$\Rightarrow\ v^T u$$

when M is orthonormal ($M^T M = I$)

Random Projection

- Johnson-Linderstrauss Lemma

  - given $0 < \varepsilon < 1$, a set of X of m points in R^N, $k > 8\ln(m)/\varepsilon^2$, there exists a linear map $f: R^N \rightarrow R^k$ such that

$$(1-\varepsilon)|u-v|^2 \leq |f(u)-f(v)|^2 \leq (1+\varepsilon)|u-v|^2$$

    where f is orthogonal projection

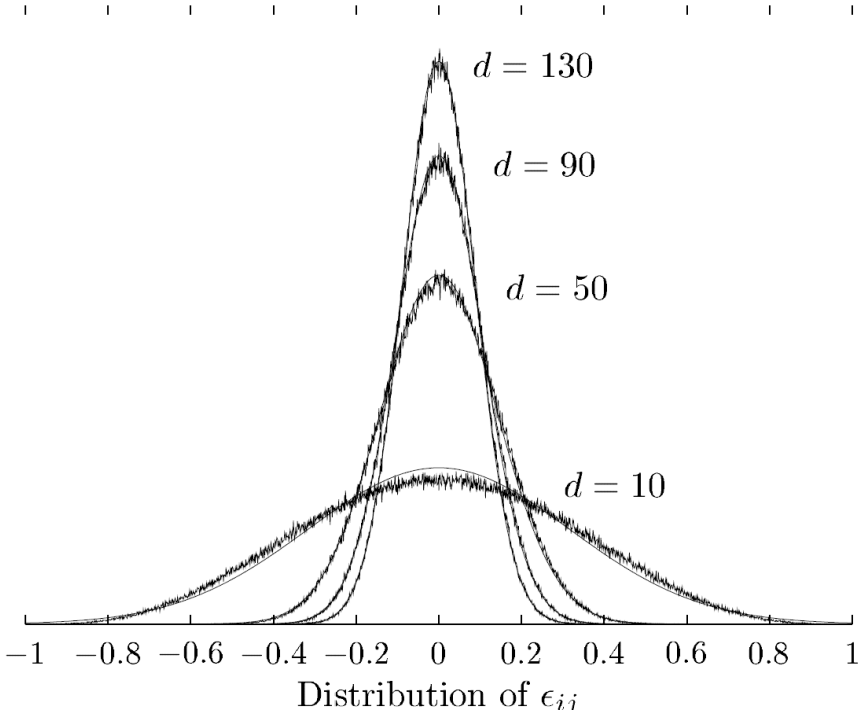
Random Projection

- Random mapper $M \in R^{k \times N}$ is almost orthogonal
  - 모든 columns 이 정확히 orthogonal 일 필요도 없습니다.
  - Orthogonal 은 두 벡터가 서로 상관없다는 의미입니다. (correlation 0)

In a high-dimensional space, there exists a much larger number of almost orthogonal than strictly orthogonal, thus random directions might be sufficiently close to orthogonal

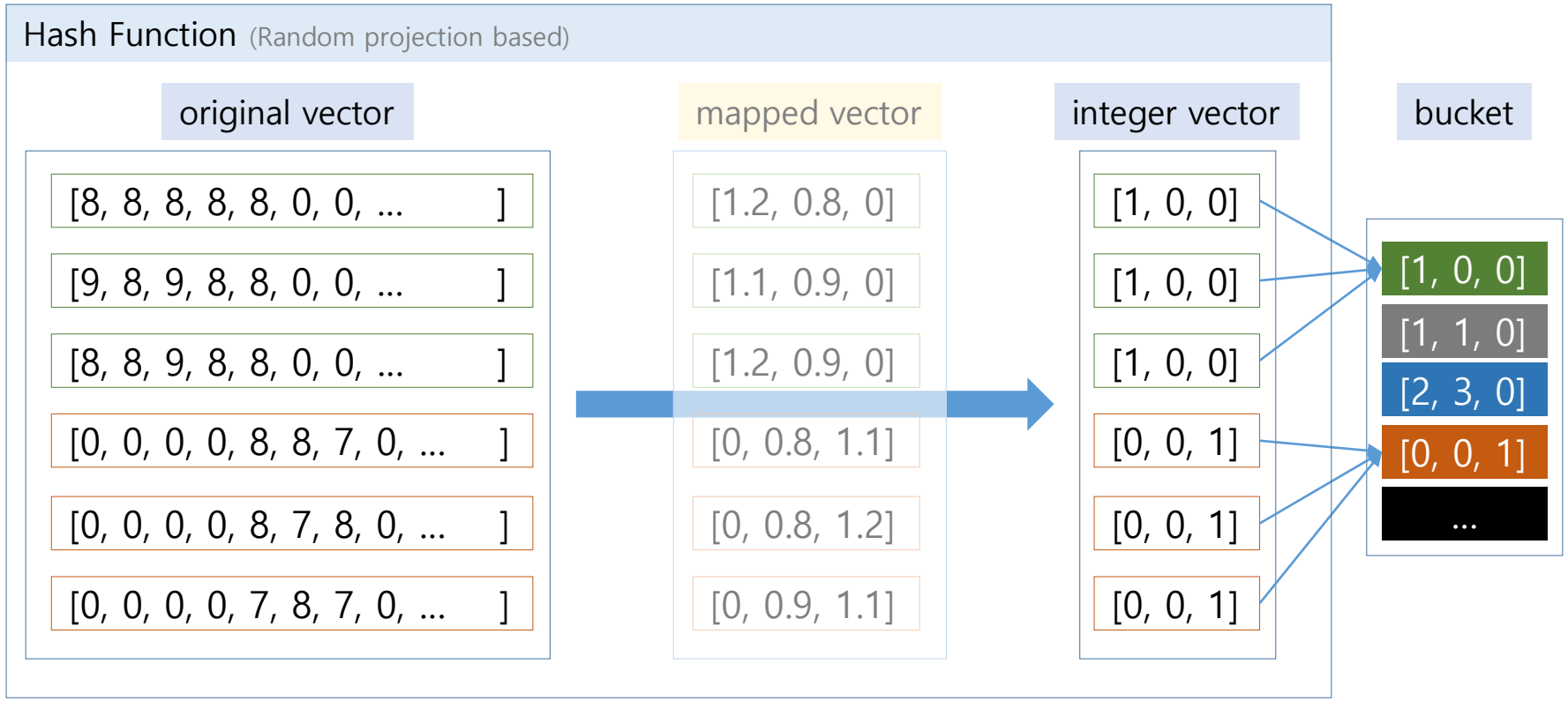
Random Projection

- Random mapper $M \in R^{k \times N}$ is almost orthogonal
  - Let assume $x=Mu$, $y=Mv$ where $x,y \in R^k$, $u,v \in R^N$
    - $x \cdot y = u^T M^T M v$
    - $M^T M = I + \epsilon$, where $\epsilon_{ij} = m_i^T m_j$



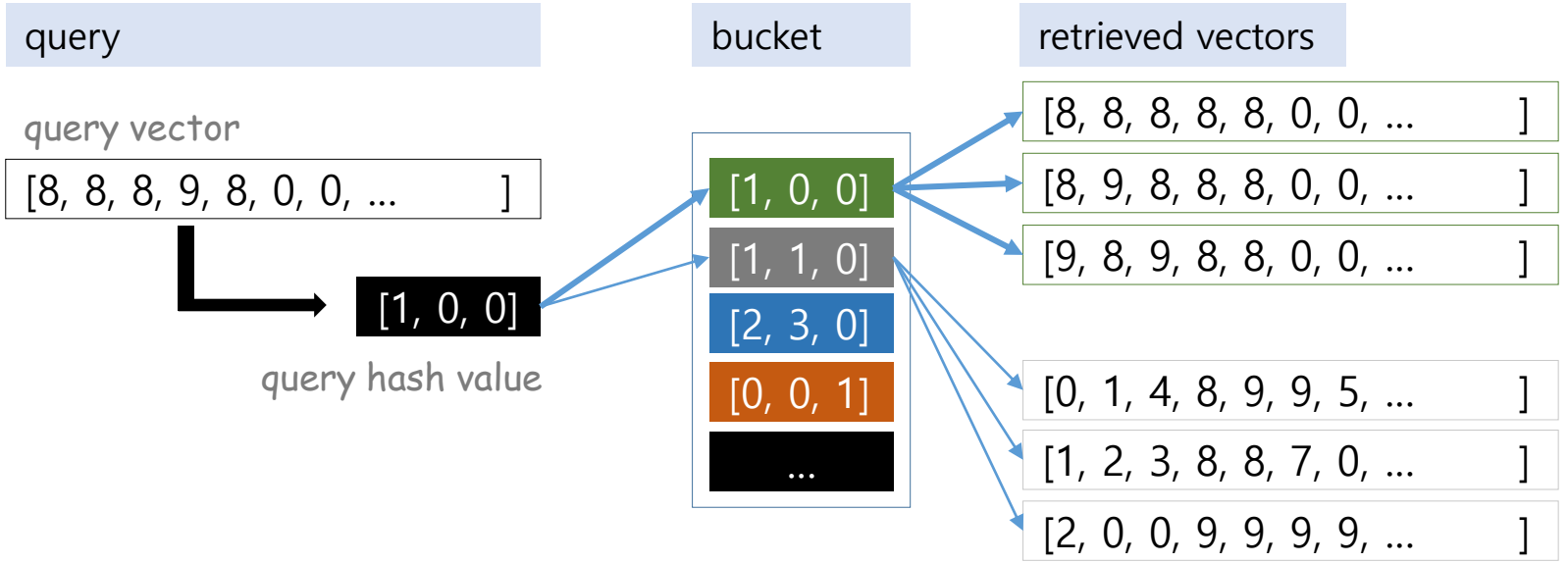
Locality Sensitive Hashing

- Random projection 를 통하여 거리를 보존하는 저차원 벡터를 얻습니다.
- 저차원 벡터를 integer vector 로 바꾼 뒤, 이를 hash code 로 이용합니다.



Locality Sensitive Hashing

- Query 에 대하여 동일한 mapper 를 이용하여 hash code 를 만든 뒤, 같은 hash code 를 지니는 벡터들에 대하여 실제 거리를 계산합니다.
- 같은 hash code 를 지니는 데이터의 개수가 k 보다 작을 경우, 비슷한 hash code 를 지니는 데이터도 최인접 이웃의 후보에 추가합니다.



Locality Sensitive Hashing

- LSH 는 최인접이웃을 빠르게 탐색하기 위하여 단순한 (작은 차원의, 정수형) 벡터로 representation 을 변형합니다.

Locality Sensitive Hashing

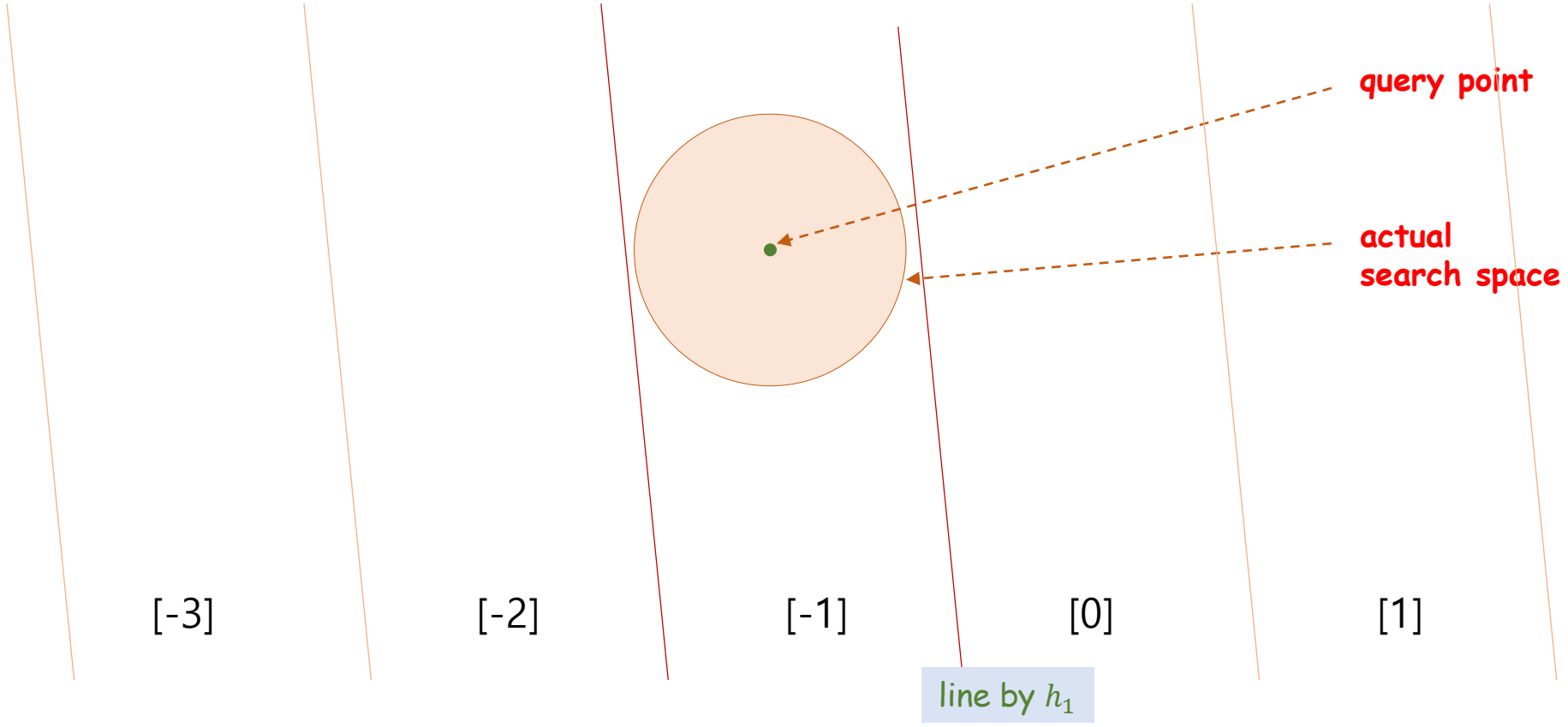
- m 차원의 hash code 를 만드는 함수는 m 개의 random projection 으로 이뤄져 있습니다.

$g_j = (h_1, \ldots, h_m)$, m 개의 random projection

- $h_i(x) = \left\lfloor \frac{a_i^T x - b_i}{r} \right\rfloor$,
- a_i is randomly generated direction vector,
- $b_i \sim U[0, r]$

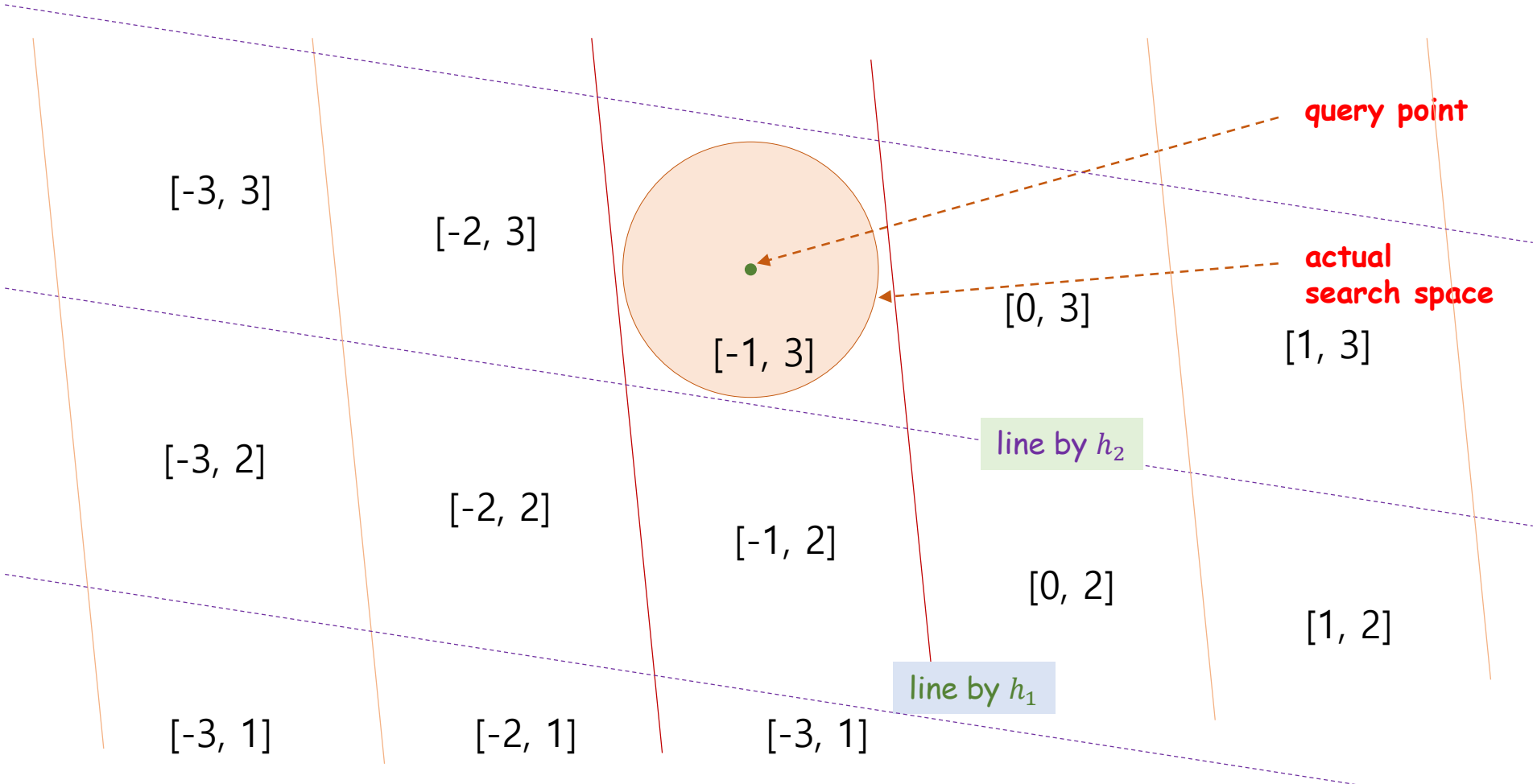
Locality Sensitive Hashing

- 한 개의 integer 는 한 종류의 평행한 선들로 구분되는 공간입니다.



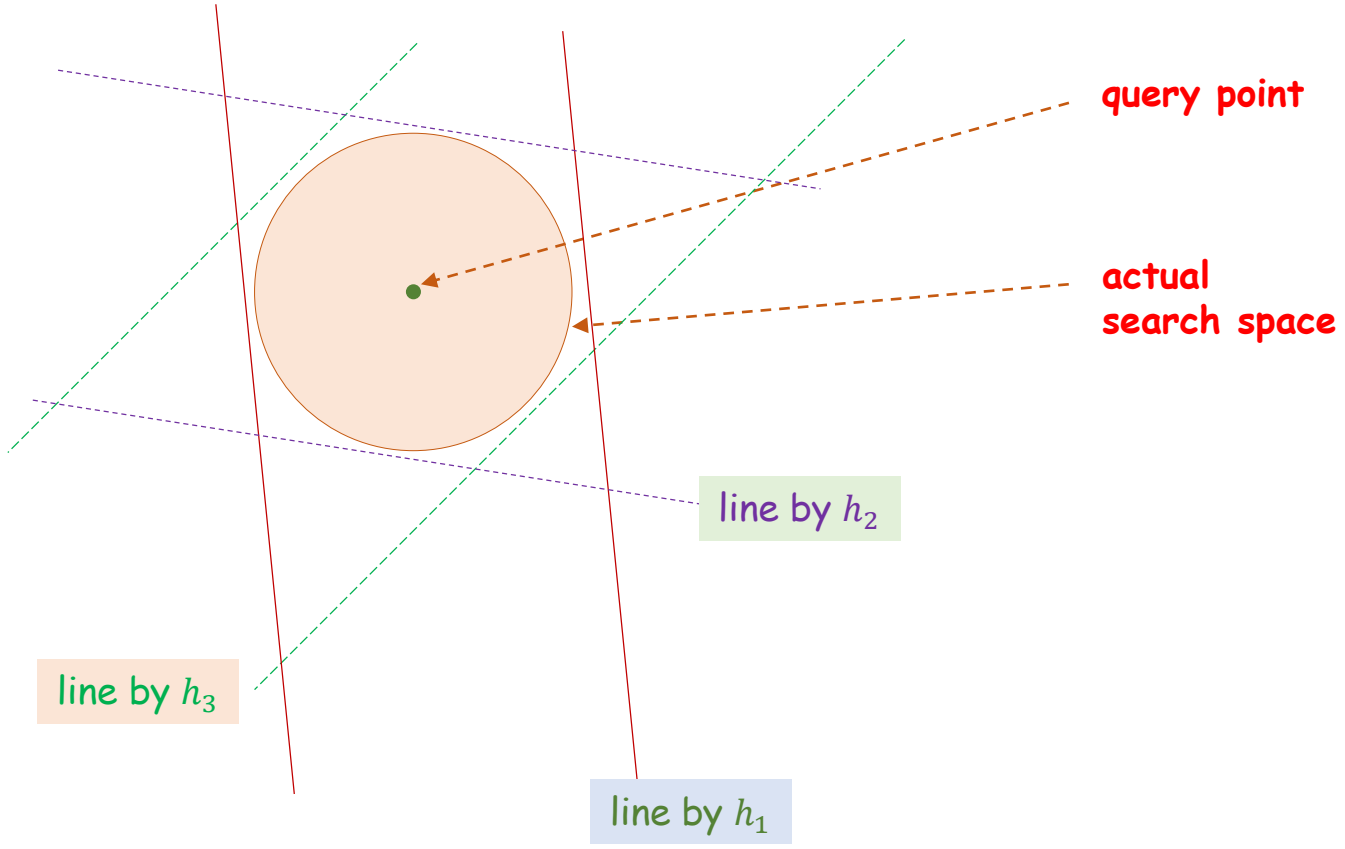
Locality Sensitive Hashing

- 두 개의 integers 는 두 종류의 평행한 선들로 구분되는 공간입니다.



Locality Sensitive Hashing

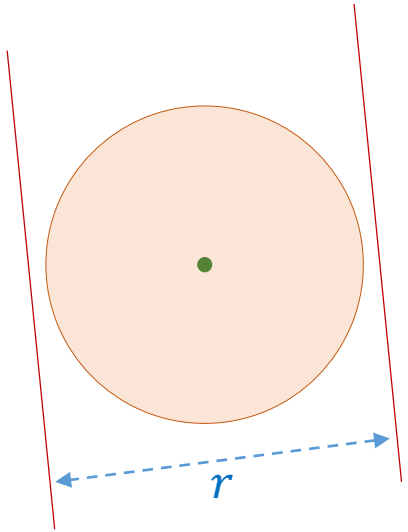
- Hash code 의 길이가 길어질수록 각 bucket 은 '구' 모양에 가까워집니다.
  - *k*-nearest neighbor search space 에 가까워집니다.



Locality Sensitive Hashing

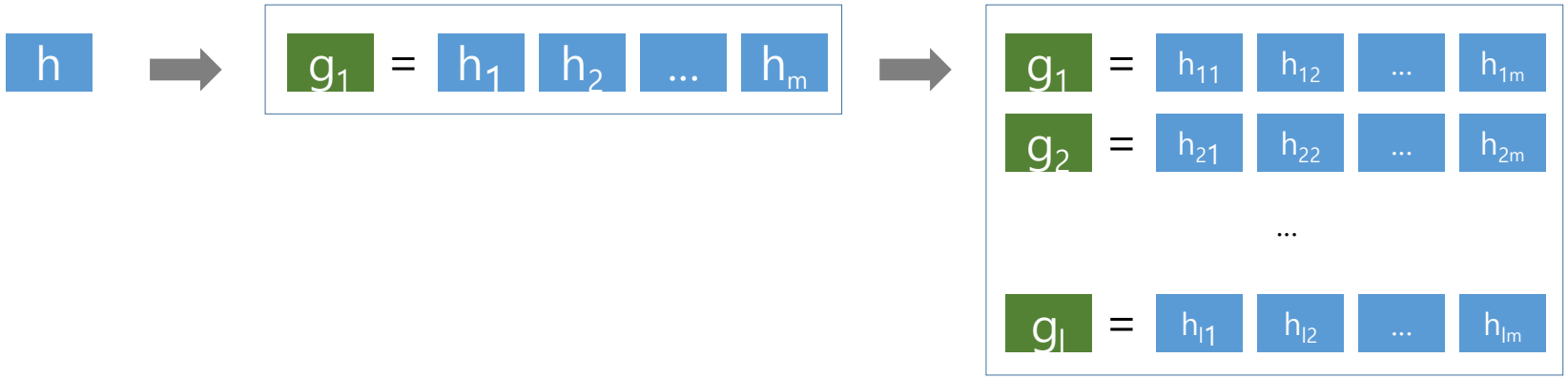
- r 은 평행한 선분 간의 거리로, bucket 의 두께에 해당합니다.

$$g_j = (h_1, \dots, h_m) \quad \text{where} \quad h_i(x) = \left\lfloor \frac{a_i^T x - b_i}{r} \right\rfloor$$



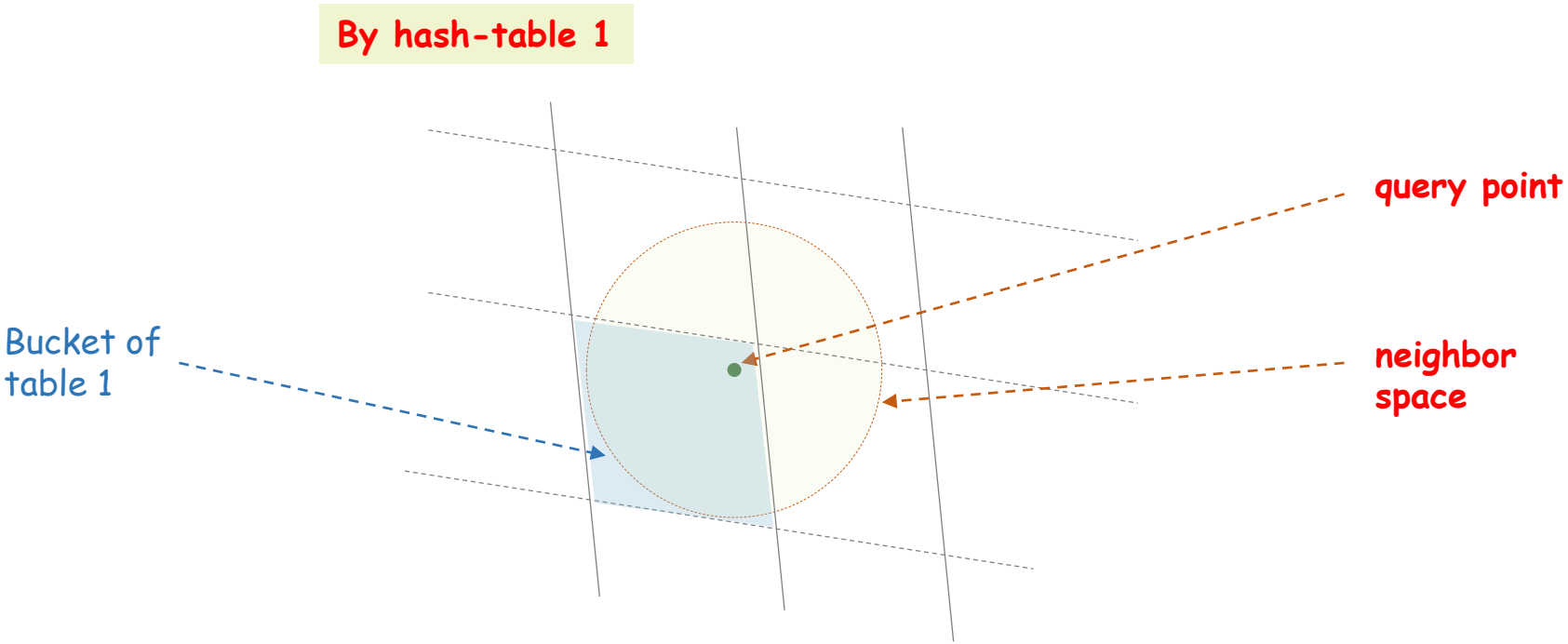
Locality Sensitive Hashing

- LSH 는 여러 개의 $g_j = (h_1, \ldots, h_m)$ 를 겹쳐서 이용합니다.
  - 하나의 $\boldsymbol{g}$ 는 nearest neighbor search 의 성능이 낮습니다.
  - 각각의 $\boldsymbol{g}$ 의 사각지대를 여러 장의 $\boldsymbol{g_1, g_2, \ldots}$ 로 보완합니다.



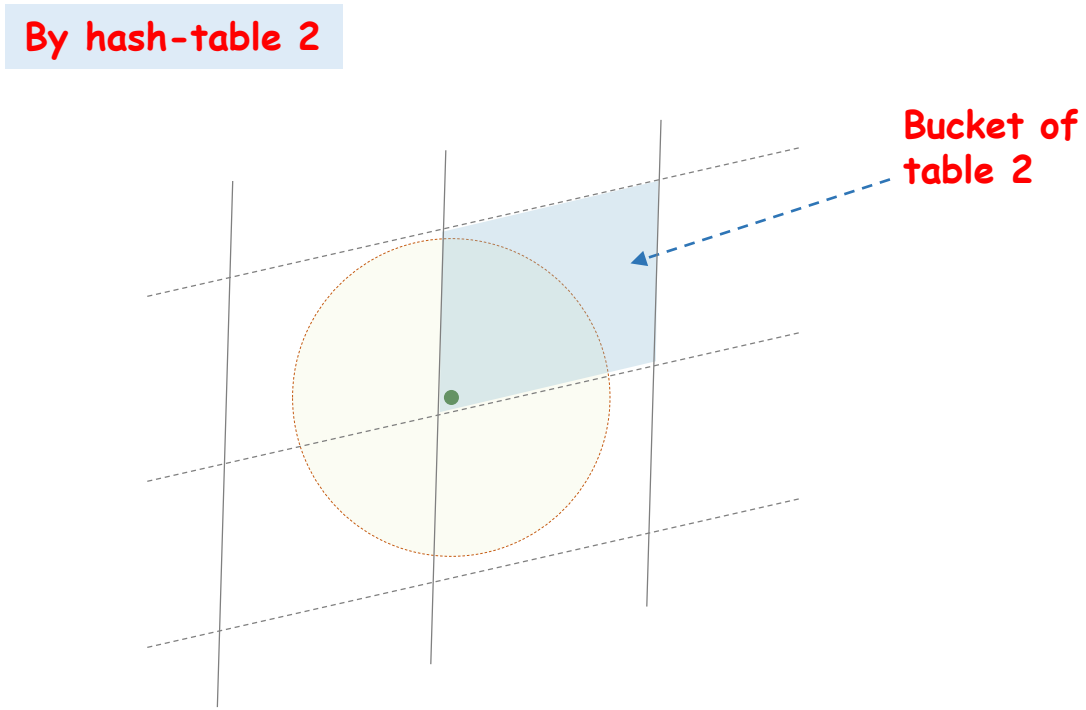
Locality Sensitive Hashing

- 한 개의 $\boldsymbol{g} = (h_1, \ldots, h_m)$ 를 이용할 경우, query 가 한쪽의 모서리에 위치할 수 있습니다.



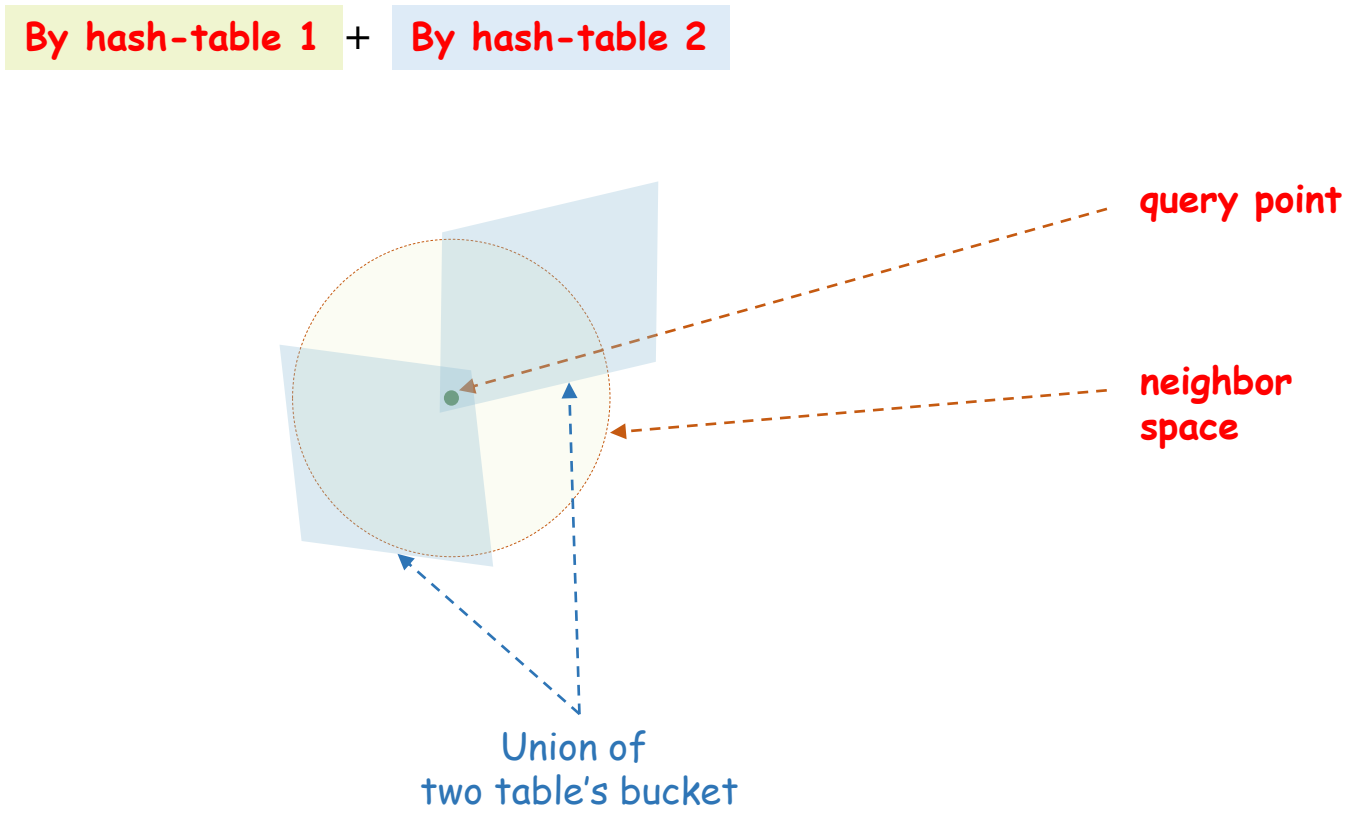
Locality Sensitive Hashing

- 한 개의 $\boldsymbol{g} = (h_1, \ldots, h_m)$ 를 이용할 경우, query 가 한쪽의 모서리에 위치할 수 있습니다.



Locality Sensitive Hashing

- 두 개의 g_1, g_2 를 함께 이용하면 두 buckets 의 데이터들을 최인접이웃의 후보로 이용할 수 있습니다.



Locality Sensitive Hashing

- 각 distance 마다 hash function 은 다르게 정의됩니다.
  - 앞의 예시는 Euclidean distance 를 보존하는 LSH 입니다.
  - Cosine distance 를 보존하기 위하여 아래의 함수를 이용할 수 있습니다.
    - $h_{ij}(x) = \left\lfloor \frac{\theta(x, a_{ij})}{r} \right\rfloor$, a_{ij}는 random vector

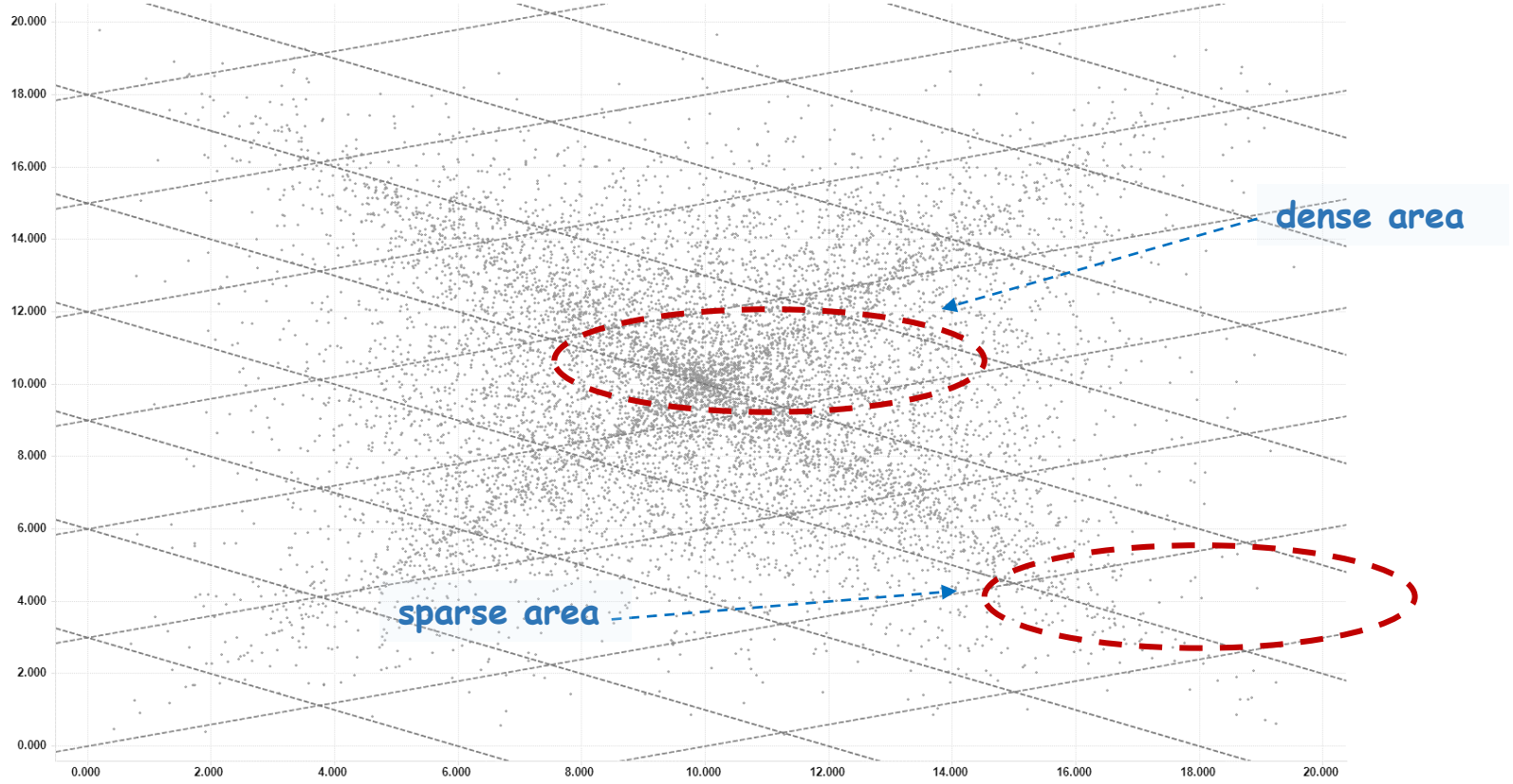
Locality Sensitive Hashing

- 실제 엔진으로 이용할 때에는 각 상황에 적절한 방법들이 추가됩니다.
  - Twitter 검색 DB *
  - Elasticsearch 의 “MoreLikeThisQuery” 함수는 LSH 를 기반으로 만들어졌습니다.

* Sundaram, N., Turmukhametova, A., Satish, N., Mostak, T., Indyk, P., Madden, S., & Dubey, P. (2013). Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. Proceedings of the VLDB Endowment, 6(14), 1930-1941. ISO 690
** https://www.elastic.co/blog/this-week-in-elasticsearch-and-apache-lucene-2016-01-18

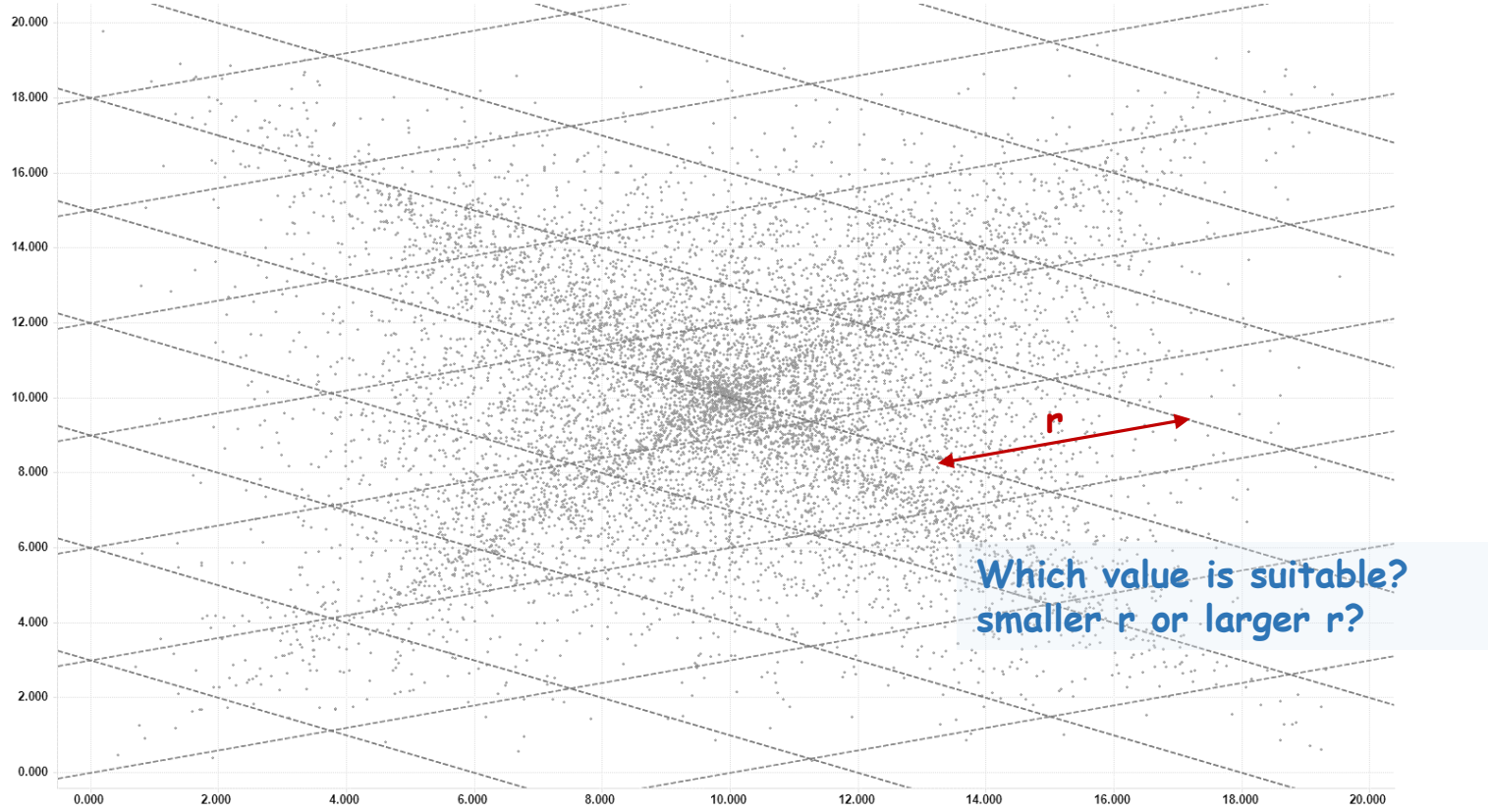
Locality Sensitive Hashing

- **한계점 1.** 각 bucket 의 밀도에 따라 거리 계산량이 다르며, sparse 한 경우 후보의 개수가 부족할 수 있습니다.



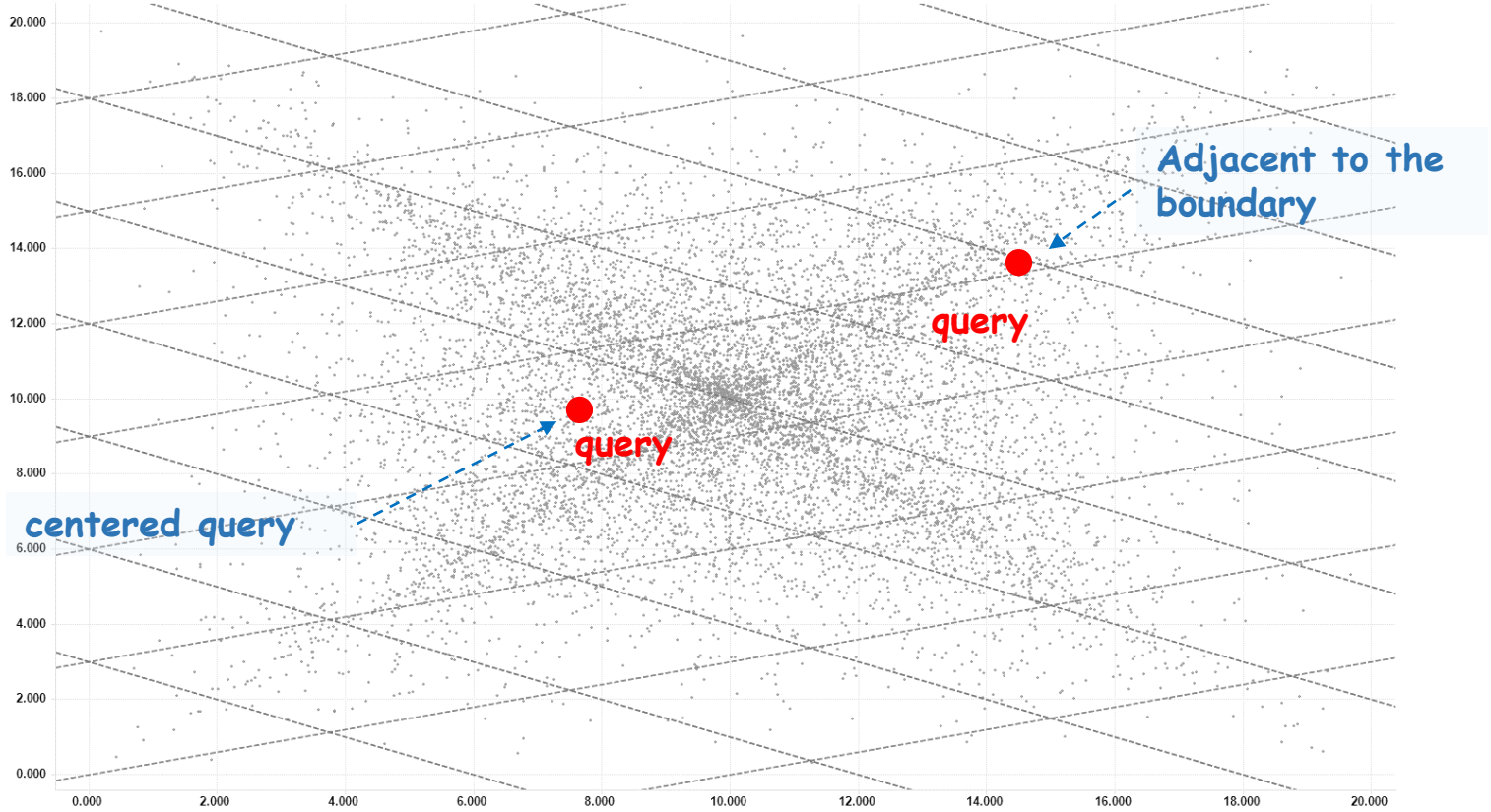
Locality Sensitive Hashing

- **한계점 2**. 적절한 패러메터를 설정하기 어렵습니다.



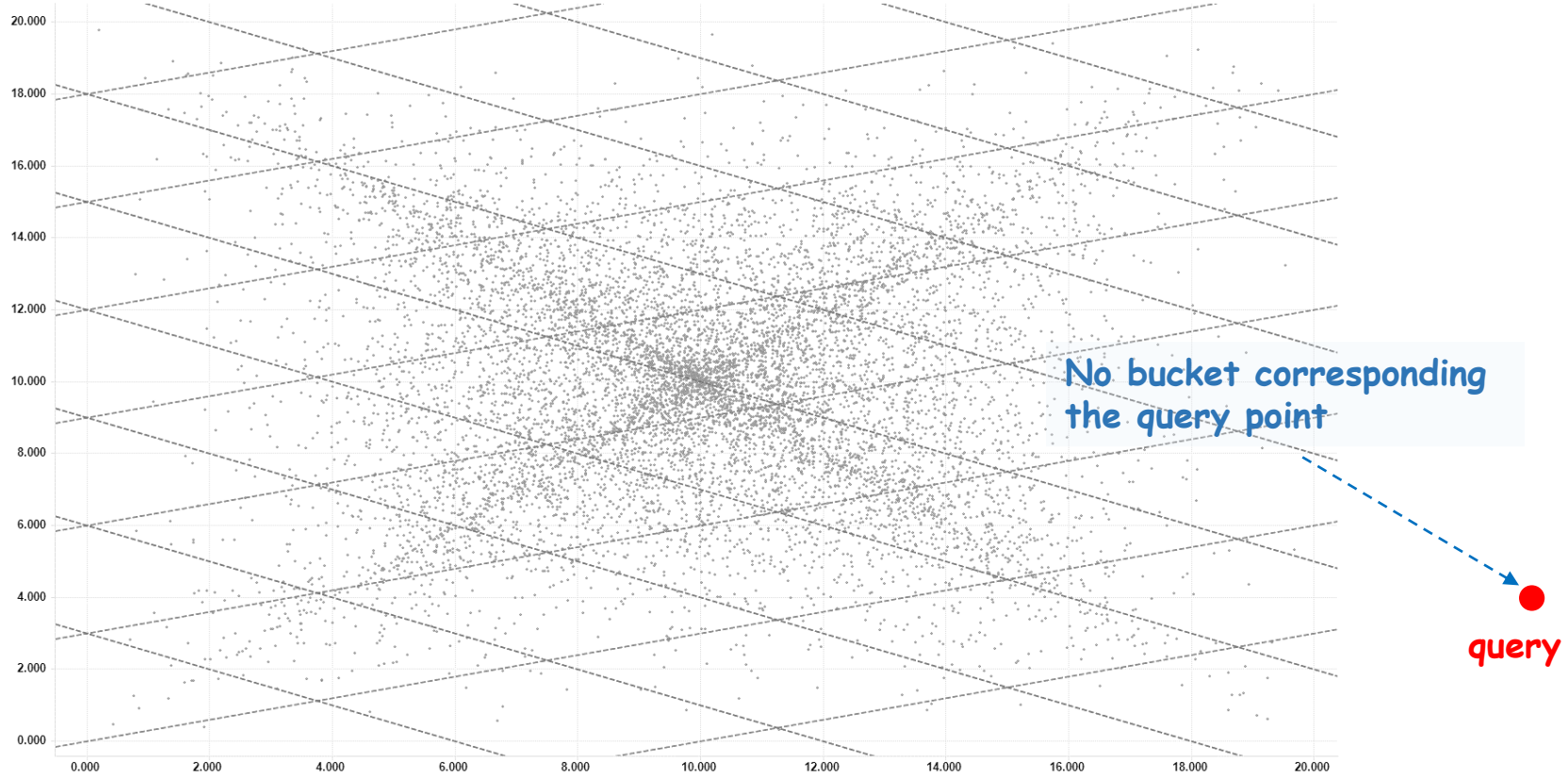
Locality Sensitive Hashing

- **한계점 3**. Query 가 bucket 의 가장자리에 위치하면 좋은 이웃 후보를 얻기 어렵습니다.



Locality Sensitive Hashing

- **한계점 4.** Query 가 포함된 bucket 에 reference data 가 없을 수도 있습니다.
- **하지만,** incremental indexing 은 가능합니다.



Locality Sensitive Hashing

- **한계점 5**. 거리 척도마다 hash 함수를 새로 디자인해야 합니다.
  - Cosine 의 경우 $h_i = sgn\,(m^T x)$ 같은 함수를 이용할 수도 있습니다.

- Scikit-learn >= 0.21.3 에서 LSHForest 구현체의 성능이 좋지 않아 deprecated 되었습니다.
- Scikit-learn 과 조금 다르게 구현한 LSH 는 실습 자료에 있습니다.

Collaborative Filtering (CF)

- CF 는 추천엔진의 한 가지 방식입니다.
  - 한 사용자 u 에게 추천할 아이템 i 를 선택하기 위하여
  - u 와 비슷한 아이템 구매 이력을 지닌 $NN(u)$ 를 탐색하고,
  - $NN(u)$ 이 구매하거나 선호하지만 아직 u 가 구매하지 않은 아이템을 추천합니다.

Collaborative Filtering (CF)

- CF 는 다음의 단계로 구성되어 있습니다.
  - 사용자의 벡터화:

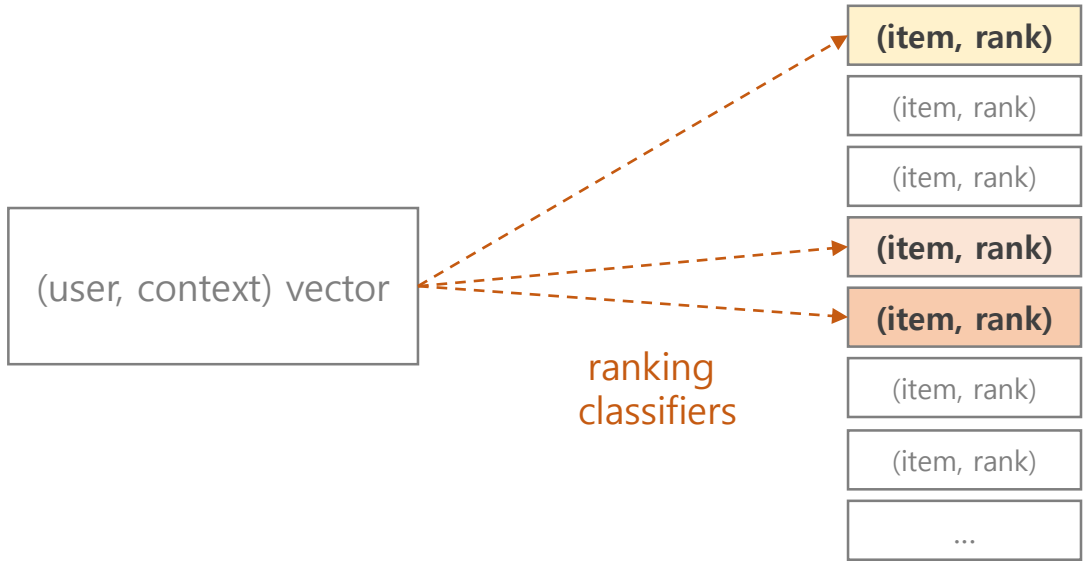
    모든 사용자 u 를 벡터로 표현합니다. 벡터 표현의 기준은 다양합니다. 임의의 인코딩 방법이 이용될 수 있습니다 (User embedding, SVD 등).
  - 유사 사용자 탐색 및 아이템 순위 선정 (ranking)

    한 사용자 u 와 벡터 기준으로 비슷한 k 명을 탐색합니다. 빠른 탐색을 위하여 ANNS 가 이용됩니다. 탐색된 유사 사용자가 선호하는 아이템을 랭킹합니다
  - 최종 아이템 순위 선정 (re-ranking)

    다양한 기준으로 선택된 아이템의 랭킹들을 종합하는 등, 아이템의 순위를 최종조절합니다.

Collaborative Filtering (CF)

- 그 외에도 현재 상황 (context vector) 을 기반으로 추천할 아이템의 랭킹을 직접 계산하는 classifiers (LambdaRank, LambdaMART 등), 방법도 이용됩니다.
  - XGBoost 에는 LambdaMART 가 구현되어 있습니다.



https://mlexplained.com/2019/05/27/learning-to-rank-explained-with-code/
https://github.com/dmlc/xgboost/blob/master/doc/parameter.rst