Nearest Neighbor Search

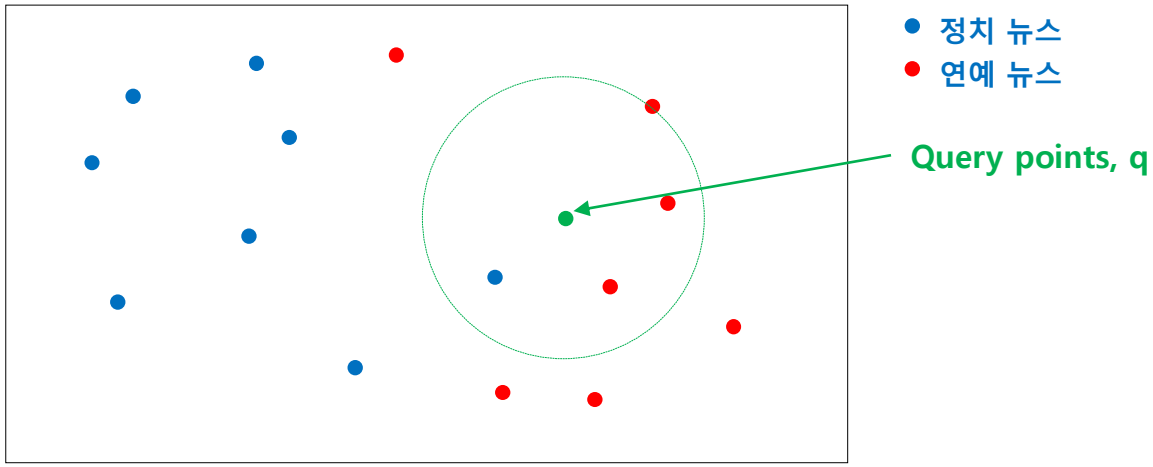
Hyunjoong Kim

soy.lovit@gmail.com

github.com/lovit

Nearest neighbor models

- NN models 은 query 와 가장 가까운 k개의 점을 이용합니다.
  - Classifier 는 다수의 labels 로 query 를 판별합니다.
  - Regression 은 이웃의 평균 y 로 query 를 예측합니다.
  - Similarity 가 weight 로 이용될 수 있습니다.



$$y(q) = \sum_{x \in K_q} w(q,x) * y(x)$$

$$w(q,x) := exp\big(-d(q,x)\big)$$

Nearest neighbor models

- *k*-NN 은 머신 러닝 알고리즘 중에서 가장 단순하지만, 가장 직관적이며, representation 이 잘되어 있다면 좋은 성능을 보여줍니다.
  - *k*-NN 은 계산 비용이 크다고 알려져 있지만, Indexer 를 이용하면 적은 계산 비용으로 neighbors search 를 할 수 있습니다.
  - 더 중요한 점은 representation 과 적절한 distance metric 을 정의하는 것 입니다.

Nearest neighbor models

- 데이터가 n 개 일 때, 주어진 query 에 대하여 k-NN을 찾기 위해서 n 번의 거리 계산을 해야 합니다.
- Logistic regression 은 두 번 의 내적만으로 classification 이 가능합니다.

$$y_\theta(q) = \begin{bmatrix} \frac{\exp(-{\theta^{(1)}}^T q)}{\exp\left(-{\theta^{(1)}}^T q\right)+\exp\left(-{\theta^{(2)}}^T q\right)} \\ \frac{\exp(-{\theta^{(2)}}^T q)}{\exp\left(-{\theta^{(1)}}^T q\right)+\exp\left(-{\theta^{(2)}}^T q\right)} \end{bmatrix}$$

2번 곱셈

$$y_\theta(x) = argmax_i \left(cos\left(q, \theta^{(i)}\right)\right)$$

N번 거리 계산 + 거리에 대하여 sorting

Nearest neighbor models

- *k*-NN 은 parametric model 처럼 함수의 구조를 가정하지 않습니다.
  - Linearly inseparable 이라 하더라도 잘 작동합니다.
  - 새로운 데이터를 추가하더라도 모델을 업데이트 할 필요가 없습니다.

Nearest Neighbor Search

- (Approximated) nearest neighbor search 문제는 주어진 query 에 대하여 가까운 이웃을 찾는 문제입니다.
  - 최소한의 거리 계산을 하면서도 높은 정확도를 유지하는 것이 목표입니다.
  - 두 개의 문제로 나뉘어집니다.
    - *k*-nearest neighbor search: k개의 최인접 점들 탐색
    - *r*-neighbor (range) search: 주어진 r보다 거리가 짧은 모든 점들 탐색

Nearest neighbor search

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

Locality Sensitive Hashing

- Random Projection
- Locality Sensitive Hashing (LSH)

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

- 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

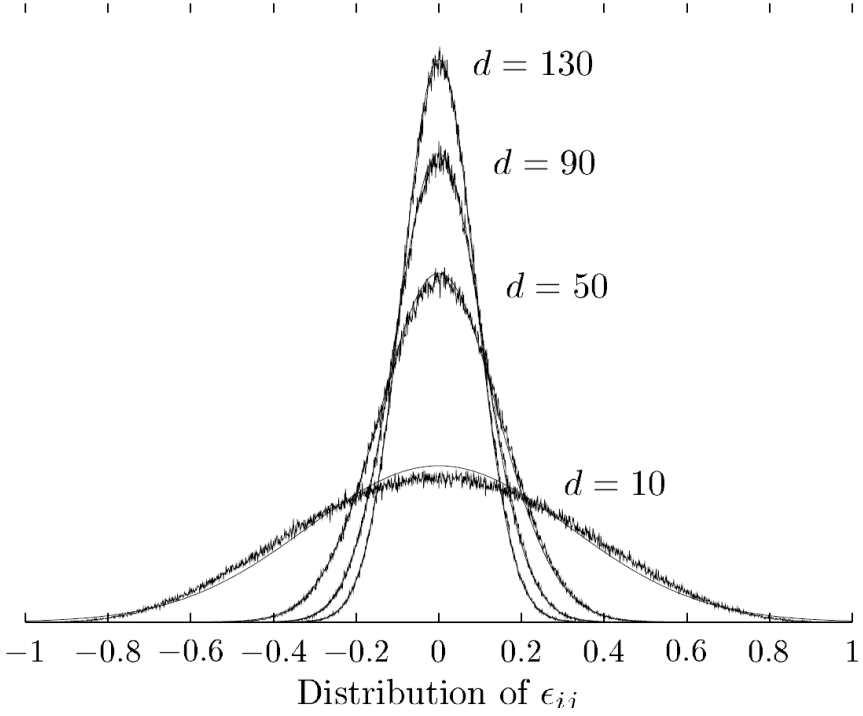
Random Projection

- 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

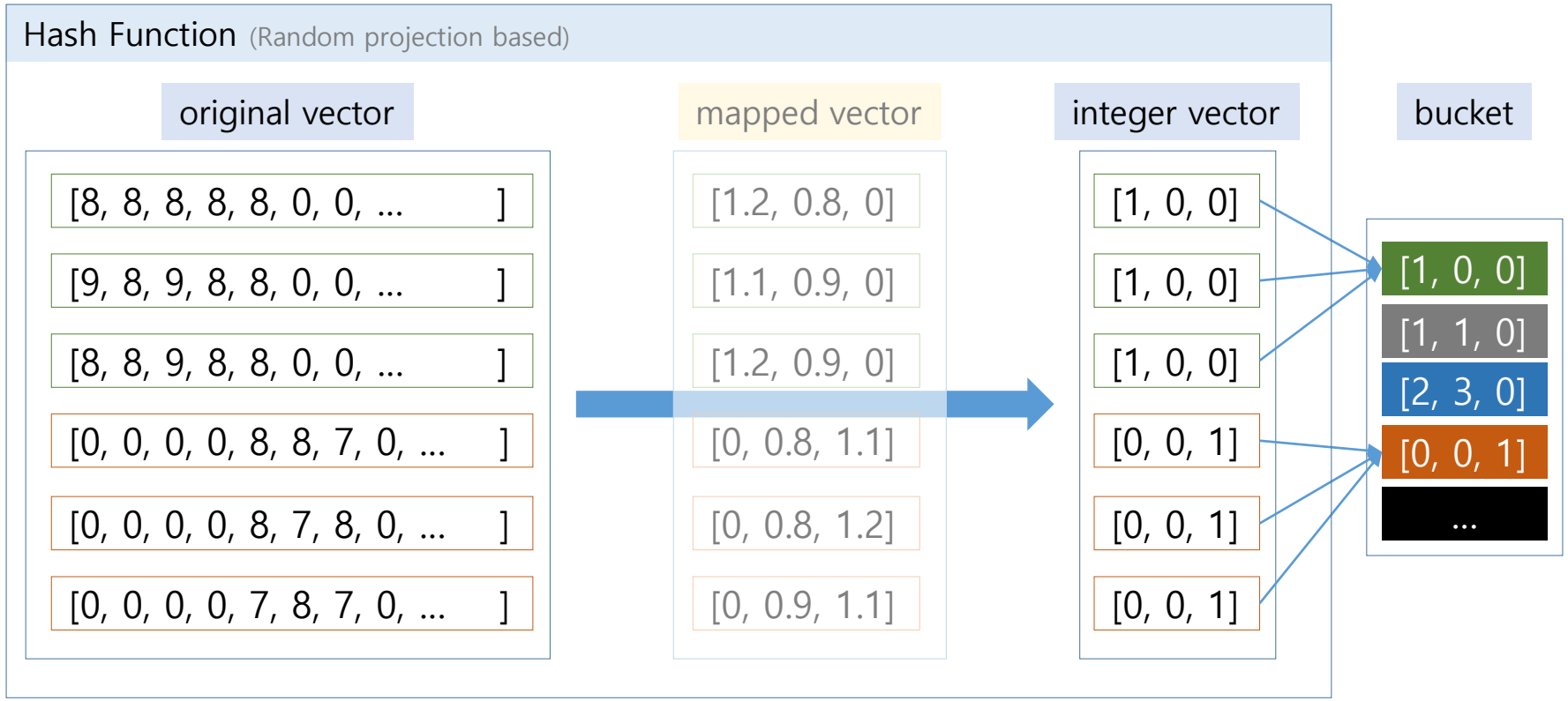
Random Projection

- 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 $\epsilon_{ij} = m_i^T m_j$



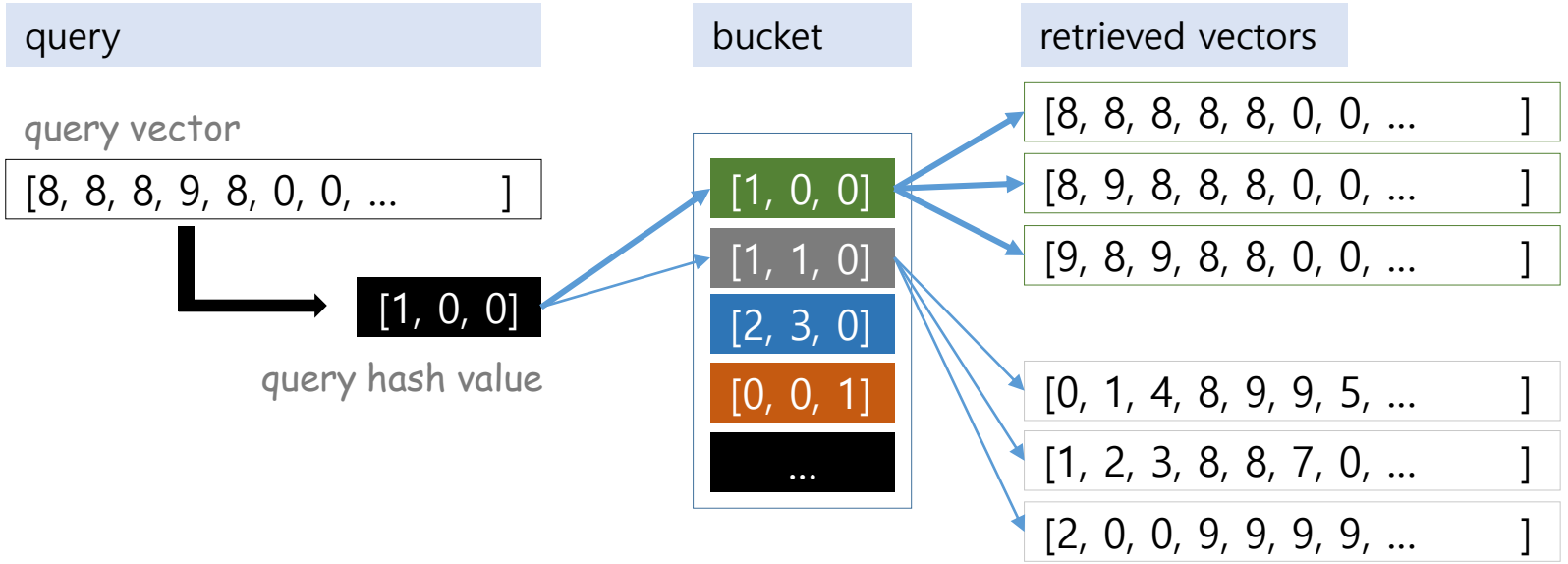
Locality Sensitive Hashing

- 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

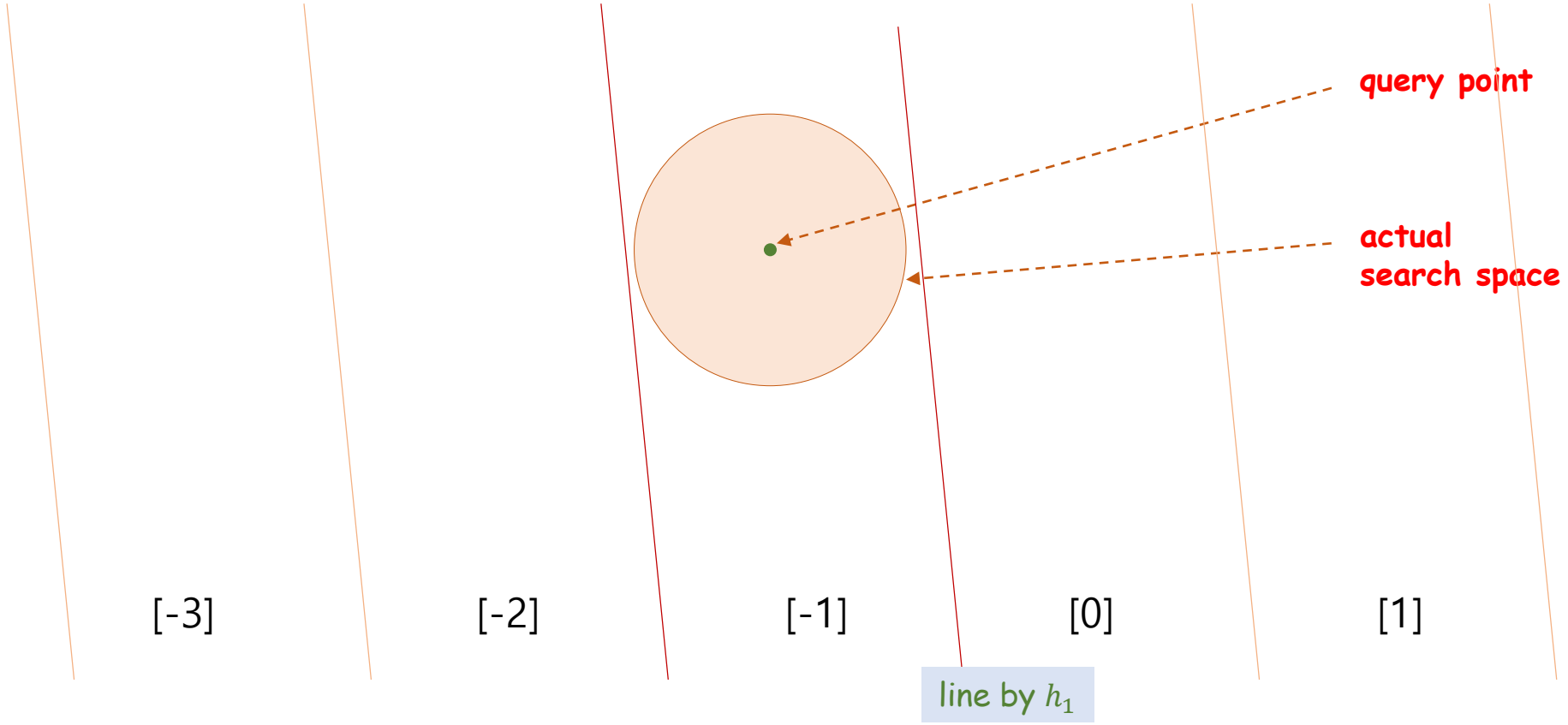
- m 차원의 hash code 를 만드는 함수는 m 개의 random projection 으로 이뤄져 있습니다.

$g_j = (h_1, \ldots, h_m)$, 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,
- $b_i \sim U[0, r]$

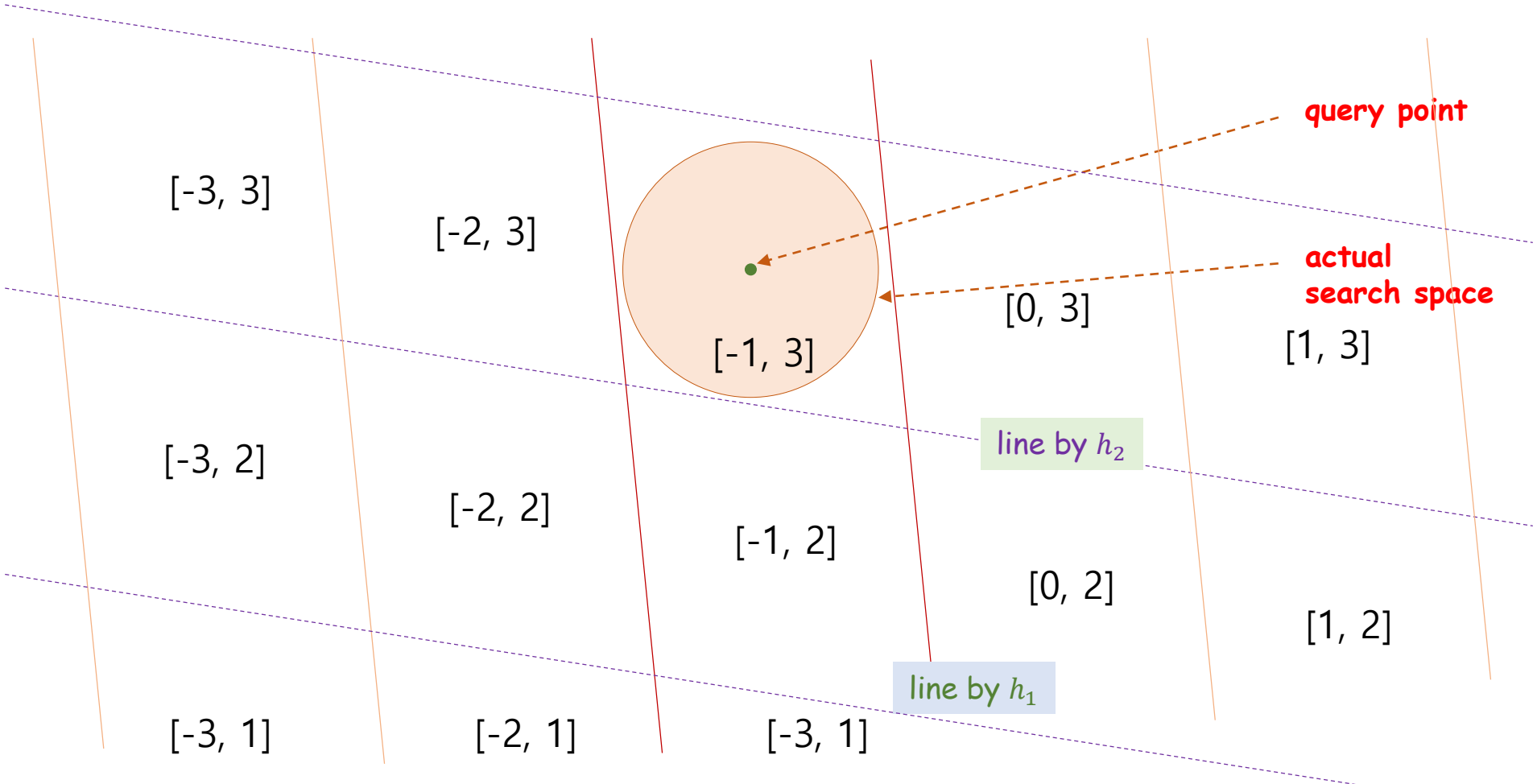
Locality Sensitive Hashing

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



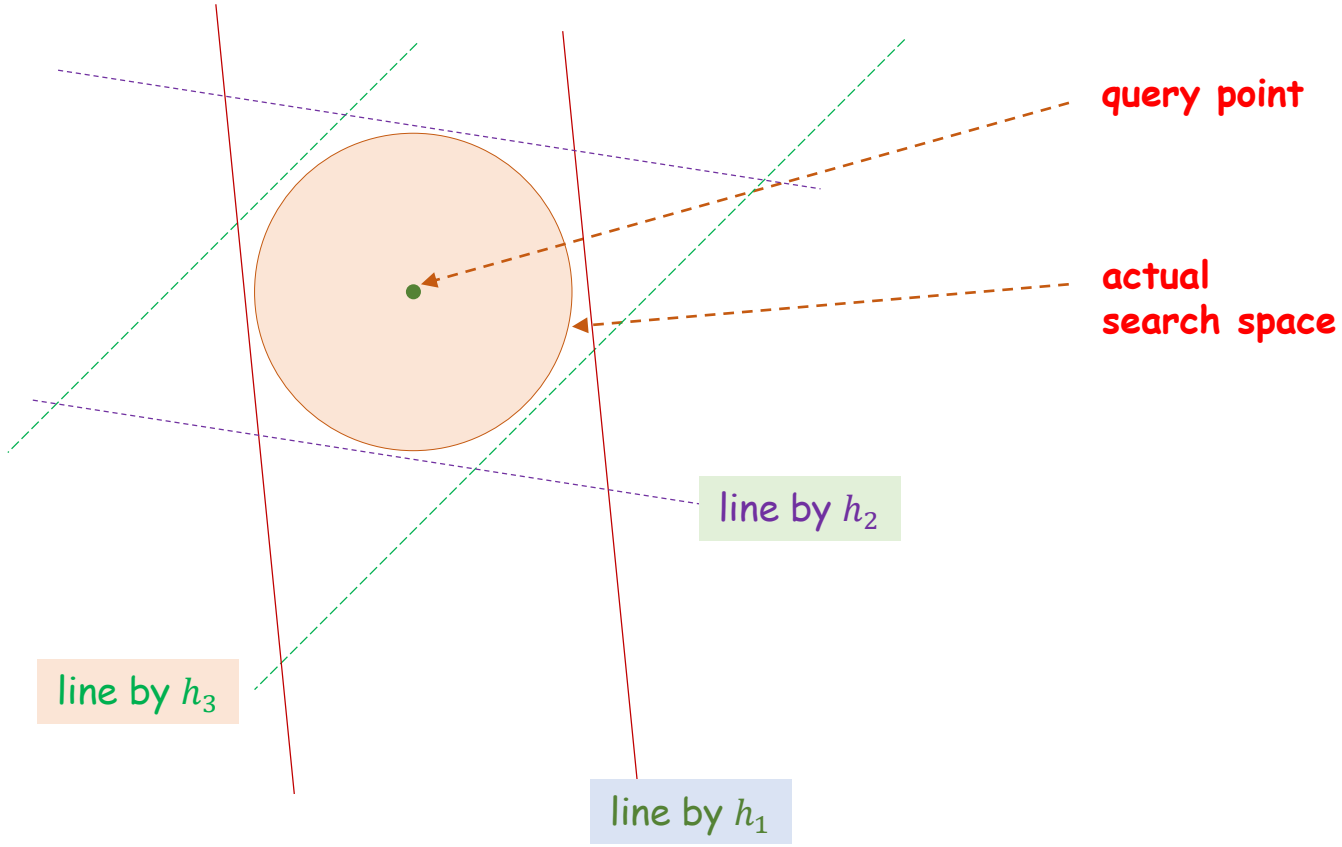
Locality Sensitive Hashing

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



Locality Sensitive Hashing

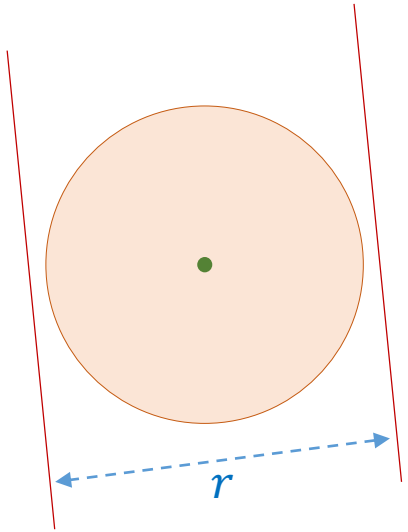
- Hash code 의 길이가 길어질수록 각 bucket 은 '구' 모양에 가까워집니다.
  - *k*-nearest neighbor search space 에 가까워집니다.



Locality Sensitive Hashing

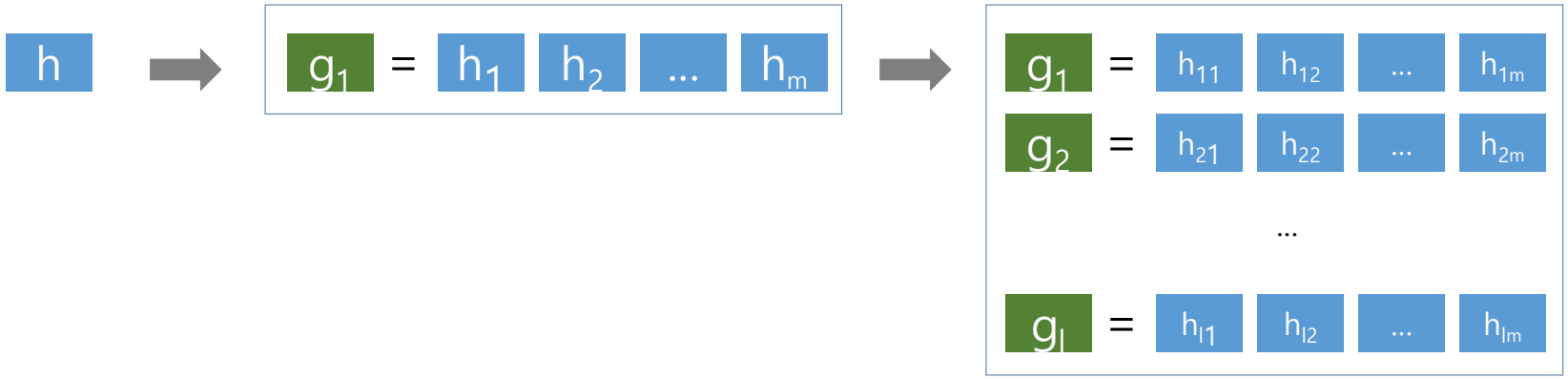
- r 은 평행한 선분 간의 거리로, bucket 의 두깨에 해당합니다.

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



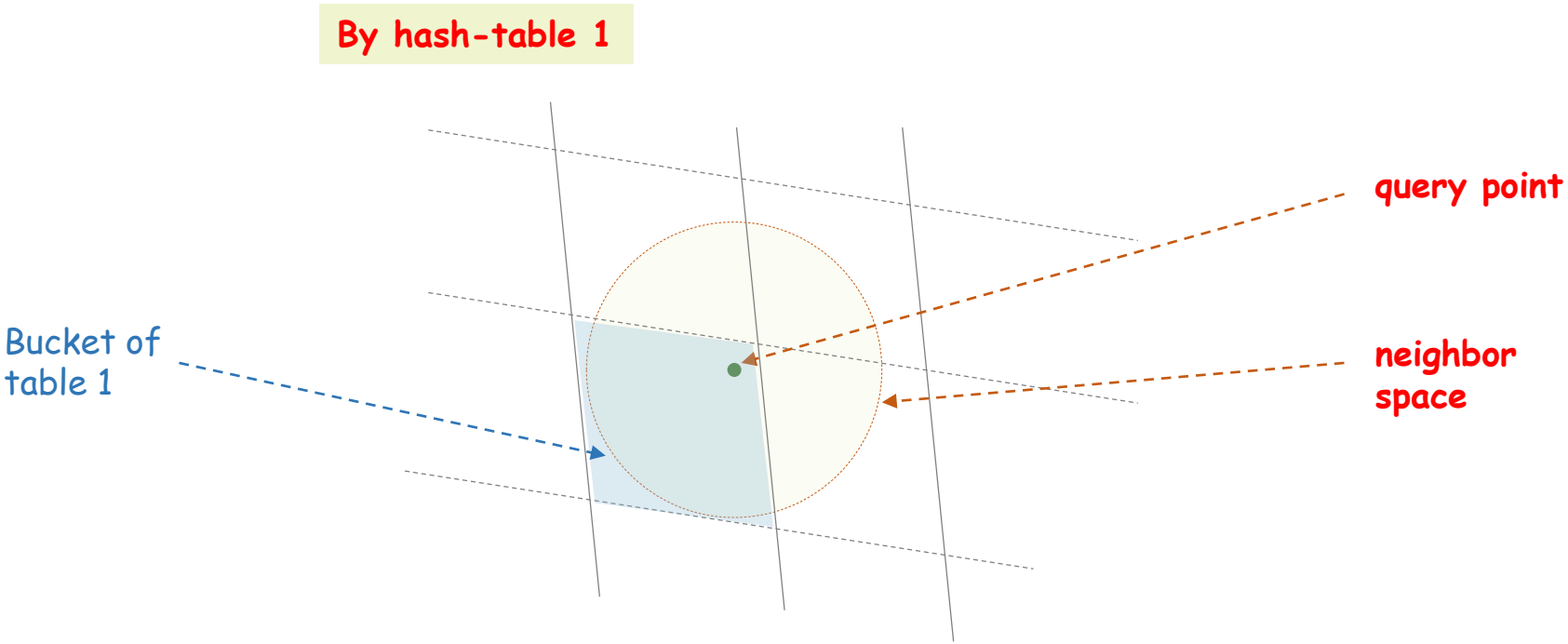
Locality Sensitive Hashing

- 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 $\boldsymbol{g_1}, \boldsymbol{g_2}, \ldots$ 로 보완합니다.



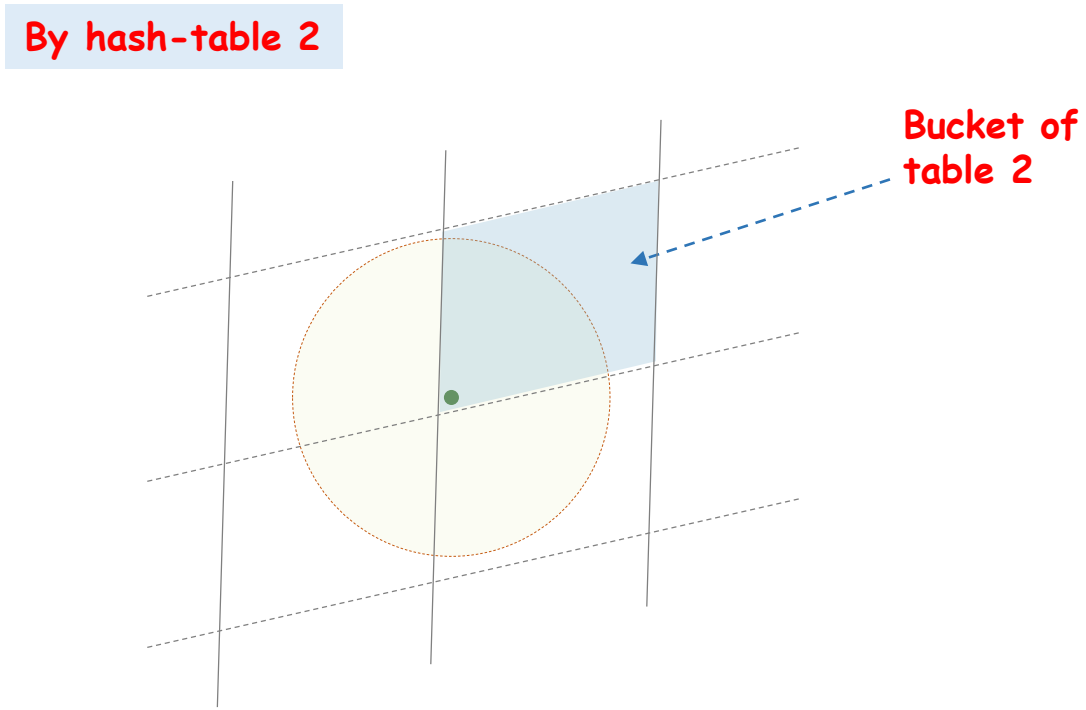
Locality Sensitive Hashing

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



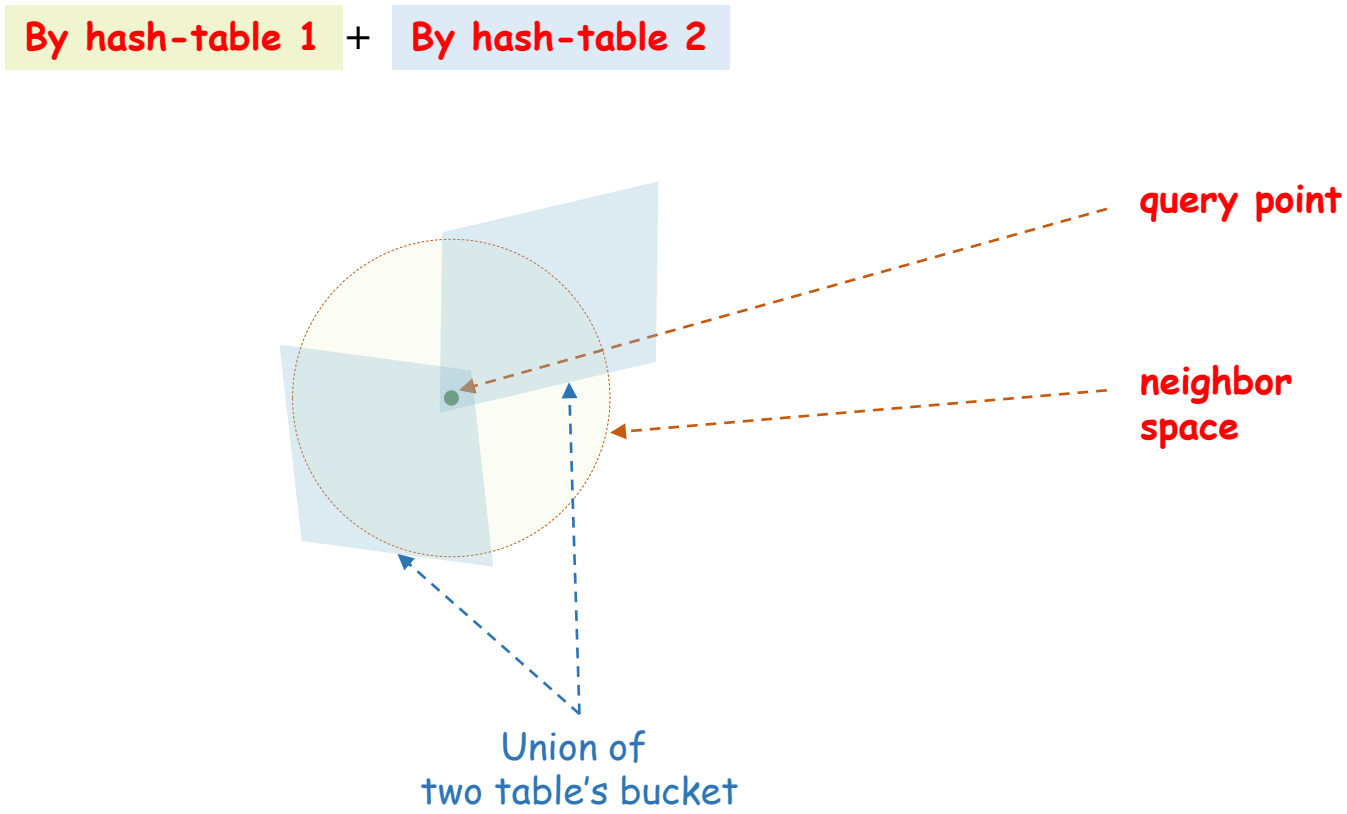
Locality Sensitive Hashing

- 한 개의 $\boldsymbol{g} = (h_1, \ldots, h_m)$ 를 이용할 경우, query 가 bucket 의 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

- scikit-learn 의 LSHForest 는 Cosine distance 를 보존합니다.
  - n_estimators: g 의 개수
  - n_candidates: 하나의 g 당 탐색하는 이웃의 개수
  - n_estimators, n_candidates 가 정확도에 가장 영향을 많이 줍니다.

sklearn.neighbors.**LSHForest**¶

class sklearn.neighbors. LSHForest (*n_estimators=10*, *radius=1.0*, *n_candidates=50*, *n_neighbors=5*, *min_hash_match=4*, *radius_cutoff_ratio=0.9*, *random_state=None*) [source]

Locality Sensitive Hashing

- Scikit-learn 의 LSHForest 는 Cosine distance 를 보존합니다.
  - n_estimators, n_candidates 을 많이 키워도 정확도 향상이 계속되지는 않습니다.
  - 성능과 속도의 최적 패러메터가 존재하며, 이는 데이터별로 다릅니다.

sklearn.neighbors.**LSHForest**¶

class sklearn.neighbors. LSHForest (*n_estimators=10*, *radius=1.0*, *n_candidates=50*, *n_neighbors=5*, *min_hash_match=4*, *radius_cutoff_ratio=0.9*, *random_state=None*) [source]

Locality Sensitive Hashing

- 실제 엔진으로 이용할 때에는 각 상황에 적절한 방법들이 추가됩니다.
  - Twitter 검색 DB *
  - A new LSH (locality sensitive hashing) TokenFilter and query is an alternative to the standard MoreLikeThisQuery ** (Elasticsearch)

* 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

- Hashing 기반 외에도 다양한 indexer 가 연구되었지만, 좋은 구현체들이 잘 공개되지는 않습니다.
- LSH 의 성능은 parameter sensitive 하며, 이를 극복하기 위한 후속 연구들도 많습니다.
- LSH 가 언제나 가장 좋은 방식은 아닙니다. 데이터의 특성과 목적에 따라 다른 indexer 가 더 좋은 성능을 보이기도 합니다.

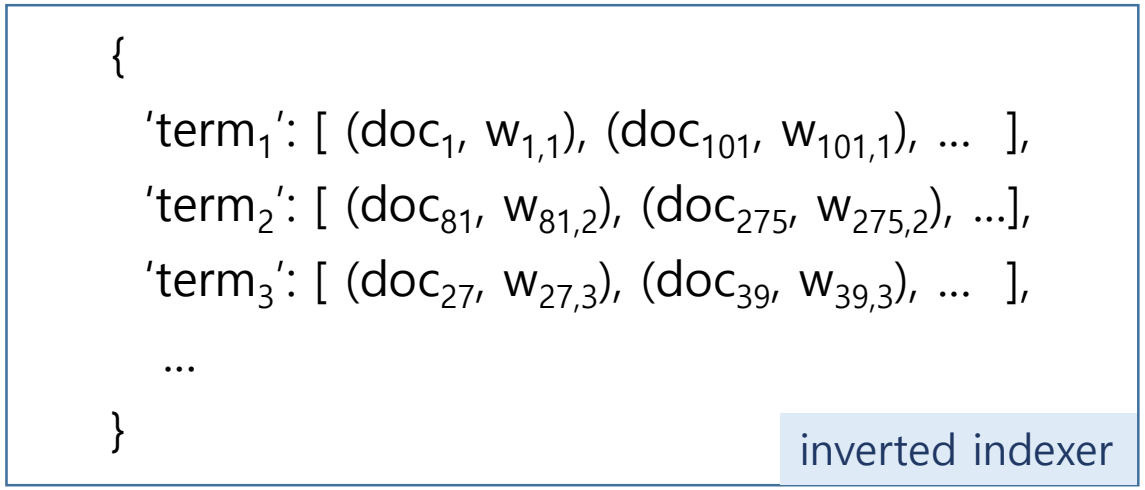
Inverted index

Inverted Indexer

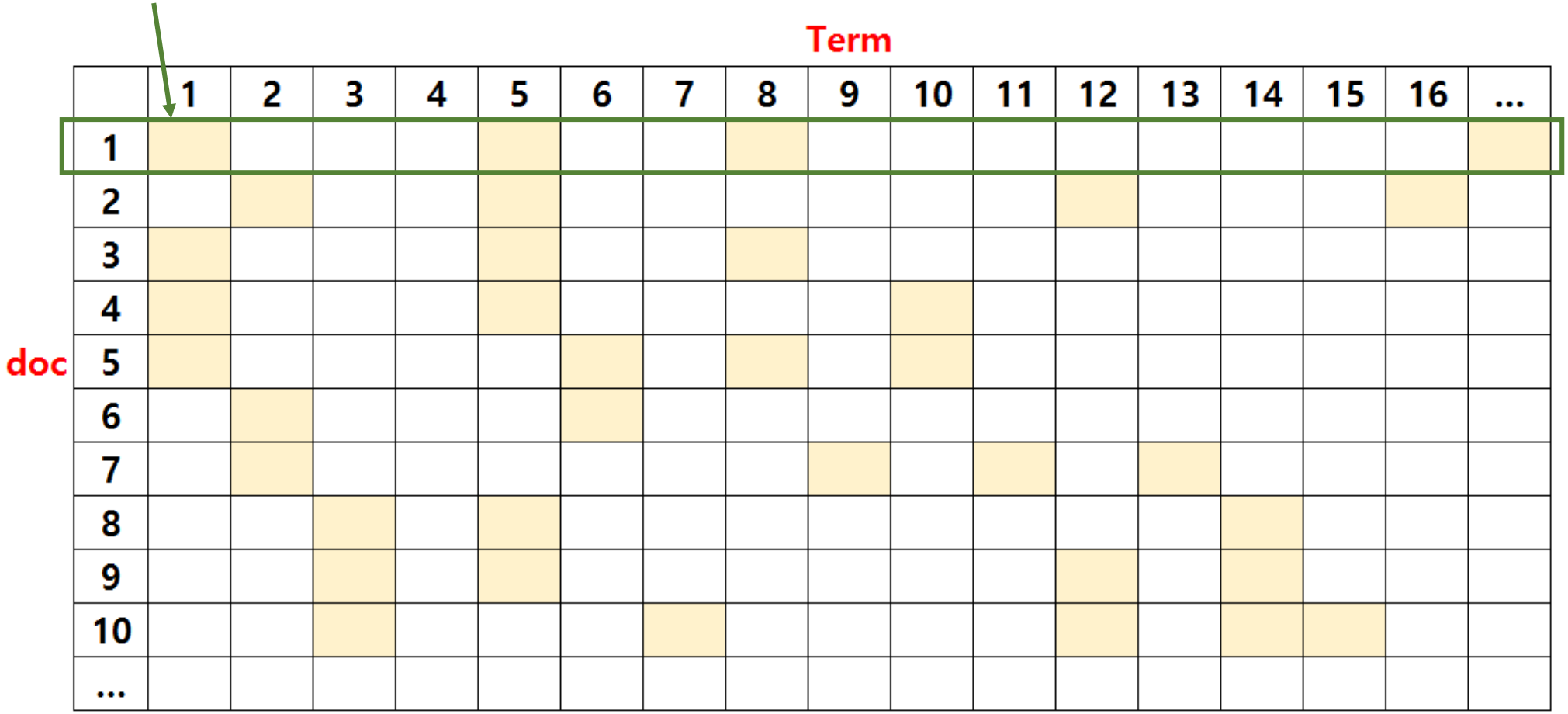
- 문장들을 bag of words 로 표현하였을 때, cosine distance 기준 비슷한 문장은 다수의 공통된 단어를 포함한 문장입니다.
- 문서처럼 벡터의 크기가 큰 경우가 아니라면 공통된 단어를 포함한 문장만을 후보로 선택하여 최인접이웃을 빠르게 검색할 수 있습니다.

Inverted Indexer

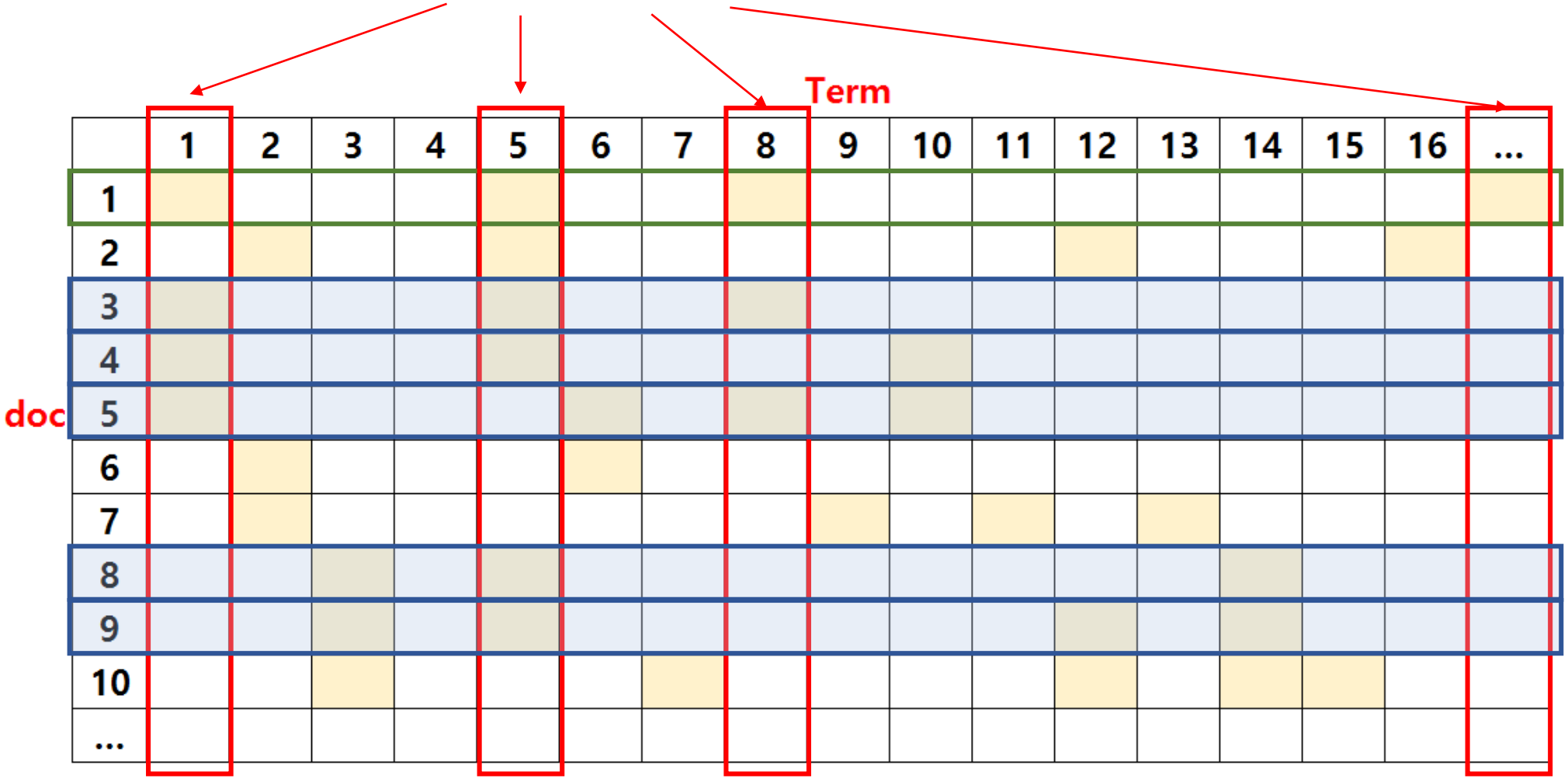
- Inverted index 는 term 기준으로 각 문서의 id 와 weight 를 저장합니다.
- 공통된 단어를 포함하는 문서들만을 최인접이웃의 후보에 포함합니다.



Which documents are similar with doc 1?



The similar documents must have at least one common term



Inverted index 를 이용한 빠른 Edit distance

- Edit distance 는 각 글자를 unit 으로 이용합니다.
- Edit distance 가 작은 두 단어는 길이가 비슷하고 겹치는 단어가 많아야 합니다.
  - 단어는 bag of characters 로 표현할 수 있습니다.
  - Sparse vector 이기 때문에 inverted index 를 이용할 수 있습니다.

Inverted index 를 이용한 빠른 Edit distance

- Edit distance 의 길이가 d 이하인 두 단어는 다음의 조건을 만족합니다.
  - $n_1 = len(w_1), n_2 = len(w_2)$ 일 때, $|n_1 - n_2| \leq d$
    - 두 단어의 길이의 차이는 d 보다 작다.
  - $len\big(set(w_1)\big) - len\big(set(w_1) \cap set(w_2)\big) \leq d$
    - 두 단어에 공통으로 있지 않는 글자의 숫자의 개수가 d 보다 작다.

Inverted index 를 이용한 빠른 Edit distance

- Character 기준으로 inverted index 를 만듭니다.
  - Index[c] = [word1, word2, …]
- 주어진 query word 의 characters 를 이용하여 다음의 조건을 만족하는 후보를 탐색합니다.
  - 두 단어의 길이의 차이는 d 보다 작다.
  - 두 단어에 공통으로 있지 않는 글자의 숫자의 개수가 d 보다 작다.

Inverted index 를 이용한 빠른 Edit distance

- 132,864 개의 단어 사전에서 Edit distance 를 이용하여 거리가 1 이하인 단어를 탐색하는 예시입니다.
  - 132,864 개의 단어와의 거리계산 및 정렬을 위하여 약 2.49 초가 소요됩니다.
  - Inverted index 를 이용하면 7 개의 단어와 실제 거리 계산을 수행합니다.
    - 0.0056 초가 소요됩니다.

https://github.com/lovit/inverted_index_for_hangle_editdistance

Inverted index 를 이용한 빠른 Edit distance

```
from fast_hangle_levenshtein import LevenshteinIndex


nouns = ['양식어가', '식품유통사', 'ETN전담팀', '도로주행', '로우프라이스펀드', …]
print(len(nouns)) # 132,864
indexer = LevenshteinIndex(nouns, verbose=True)
print(indexer.levenshtein_search('분식회계'))
```

```
[('분식회계', 0), ('분식회', 1), ('분식회계설', 1), ('분석회계', 1)]
```


https://github.com/lovit/inverted_index_for_hangle_editdistance

Faiss

- 최근 Facebook research 에서서 dense vector 의 nearest neighbor search 를 위하여 Faiss 를 제안했습니다.
  - GPU 도 이용하여 거리 계산을 빠르게 수행합니다.

* Johnson, J., Douze, M., & Jégou, H. (2017). Billion-scale similarity search with GPUs. arXiv preprint arXiv:1702.08734.
** https://github.com/facebookresearch/faiss