

INTRODUCTION TO NLP

Session 6

David Buchaca Prats
2022

TF-IDF VECTOR

- The objective of tf-idf representation is to emphasize the most relevant words present in the documents.
- Let X be a corpus.
- Let $w \in W$ be a word vocabulary of D words.
- We want to emphasize in our corpus:
 - Words that appear frequently in the document: term frequency
 - $\text{tf}(x; W) = (\#\{w_1 \mid w_1 \in x\}, \dots, \#\{w_D \mid w_D \in x\})$
 - Words that appear rarely in the corpus: inverse document frequency
 - $\text{idf}(W; X) = (\text{idf}(w_1; X), \dots, \text{idf}(w_D; X))$

BASIC NOTATION

- Let X be a set of documents.
 - You can think about X as a list of strings containing all the documents in your corpus.
- Let X_w the set of documents containing word w .
 - You can think about X_w as a list of strings where each string contains word w .
- Let $x \in X$ be a Document in the corpus
- Let W be a vocabulary
- $tf(x; W) = (\#\{w_1 \mid w_1 \in x\}, \dots, \#\{w_D \mid w_D \in x\})$
- $idf(W; X) = (idf(w_1; X), \dots, idf(w_D; X))$ where $idf(w; X) = \log \left(\frac{|X|}{1 + |X_w|} \right)$
- Tf-idf vector is: $tf(x; X) \odot idf(X)$

TF-IDF SUMMARY

- We have defined $\text{tfidf}(x, X) = \text{tf}(x; X) \odot \text{idf}(X)$, where

$$\text{idf}(w, X) = \left(\log \left(\frac{|X|}{1 + |X_{w_1}|} \right), \dots, \log \left(\frac{|X|}{1 + |X_{w_D}|} \right) \right)$$

- If a word appears in a few documents the idf vector will increase its weight.
- If a word appears in a lots of documents documents the idf vector will decrease its weight.
- Let us consider a corpus of 1 milion documents:

- If w appears in 100 documents: $\text{idf}(w; X) = \log \left(\frac{1000.000}{1 + 100} \right) = 9.200$

- If \bar{w} appears in 100.000 documents: $\text{idf}(\bar{w}; X) = \log \left(\frac{1000.000}{1 + 100.000} \right) = 2.197$

FINDING THE MOST SIMILAR DOCUMENT TO A QUERY

- Let x^q be a query document the nearest neighbour problem consist on finding the document x^* that is the closest to x^{nn} .

- We are interested in finding the nearest neighbour x^{nn}

$$x^{nn} = \arg \min_{x \in X} d(x, x^q)$$

- How not to do it: Brute numpy force

- **`ind = np.argmin(np.sum((X - x_query)**2, axis=1))`**

- This is bad because:

- Needs to store the distance between each element in X and the query.

- Needs to compute the distance between the query and all the elements in X.

COSINE SIMILARITY

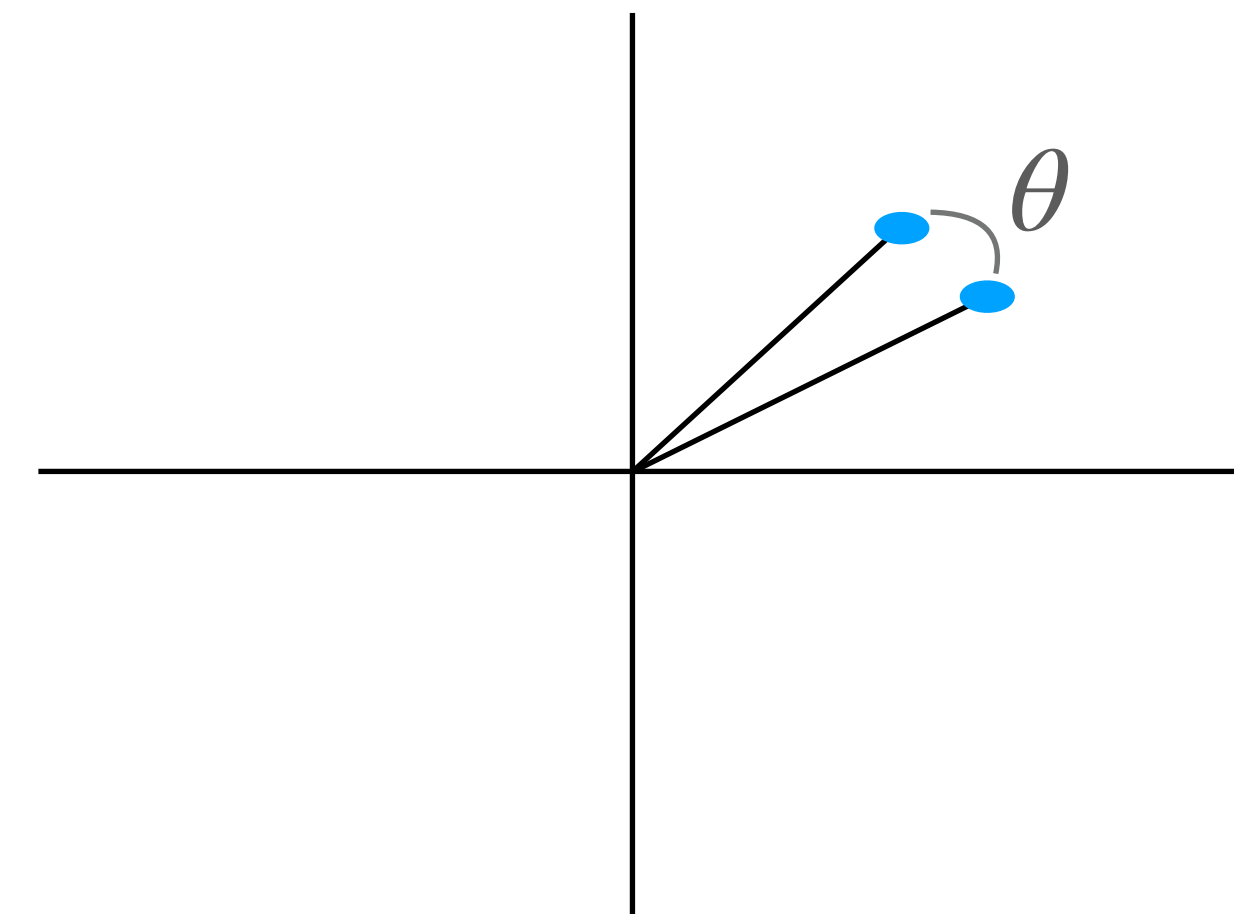
➤ Remember that the scalar product between vectors is $p^T \cdot q = \|p\| \|q\| \cos(\theta)$

➤ Hence, isolating $\cos(\theta)$ we have $\cos(\theta) = \frac{p^T \cdot q}{\|p\| \|q\|} = \frac{p^T}{\|p\|} \cdot \frac{q}{\|q\|}$

➤ The cosine similarity is defined as $cs(p, q) = \frac{p^T \cdot q}{\|p\| \|q\|} = \frac{p^T}{\|p\|} \cdot \frac{q}{\|q\|}$

➤ The cosine distance is defined as:

➤ $cd(p, q) = 1 - cs(p, q)$



NORMALIZATION IS A DOUBLE EDGE SWORD

- Should a tweet be similar to a long document ?
- We might not want to normalize all the time.
 - A possible trick is to threshold at a maximum the number of word counts
- Different distances should be used for different purposes:
 - Cosine similarity is sensible for assessing similarity between document vectors.
 - Euclidean distance is better for measuring other factors, such as the number of times a document has been read.

NORMALIZED COSINE DISTANCE AND L2 NORM

- The L2 norm of a vector x is defined as $\|x\|_2 := \sqrt{\sum_{d=1}^D (x_d)^2} = \sqrt{x^T x}$
- The euclidean distance between x and y is defined as the L2 norm of the difference
$$\|(x - y)\|_2 := \sqrt{\sum_{d=1}^D (x_d - y_d)^2} = \sqrt{(x - y)^T (x - y)}$$
- Note that sqrt is monotonic and does not change the order in which closest elements are found. Hence, if all we want is which elements are the closest, we can use:
$$\|(x - y)\|_2^2 = (x - y)^T (x - y) = x^T x - x^T y - y^T x - y^T y = x^T x - 2x^T y + y^T y$$
- If x and y are normalized, then $\|(x - y)\|_2^2 = (x - y)^T (x - y) = 2 - 2x^T y = 2 - 2\cos(\theta)$

FINDING NEAREST NEIGHBOURS FAST

- There are two different strategies for nearest neighbour search:
 - Exact nearest neighbour search.
 - Can be done with a KD-Tree.
 - The retrieval time cannot be controlled.
 - The result can be guaranteed that it is the best.
 - Not practical for high dimensionality
 - Approximate nearest neighbour search
 - Can be done with LSH.
 - The retrieval time can be controlled.
 - The result cannot be guaranteed that it is the best.
 - Scale to high dimensionality

THE HASHING TRICK

- We have seen how to perform Document classification storing a Vocabulary that maps strings to positions in the feature space. Using this vector we could represent a Document as vector of word counts.
 - Nevertheless, there is an alternative to storing a Vocabulary: The Hashing Trick
- Let C be the set of characters used in the language, $C = \{a, b, \dots, z\}$ where $\alpha := |C|$
- Let $\text{char} : C \longrightarrow \mathbb{N}$ a function that maps a character from the alphabet to an integer.
- Given a string S we can map it to an integer using $\phi : C^* \longrightarrow \mathbb{N}$ defined as:

$$\phi(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \text{char}(s_i)$$

- Note that this function will produce a different integer for each different string.

THE HASHING TRICK

- For practical reasons many times the hashing function $\phi : C^* \longrightarrow \mathbb{N}$ depends on a parameter M (the number of features) and is defined as:

$$\phi(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \text{char}(s_i) \quad \text{mod}(M)$$

- Note that this function might not produce a different integer for each different string but if M is large the probability of two strings colliding is very small.
- Users can make M can be arbitrarily big, making the function injective.

EXAMPLE OF HASHING TRICK

➤ We have defined: $\phi(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \text{char}(s_i)$

➤ $C = \{a, b\}$ where $\text{char}(a) = 0$ and $\text{char}(b) = 1$

➤ Let us consider $S=a$

$$\phi(S) := 2^1 + \sum_{i=0}^{1-1} 2^{1-(i+1)} \cdot \text{char}(s_i) = 2 + (2^{1-(0+1)} \cdot 0) = 2$$

➤ Let us consider $S=b$

$$\phi(S) := 2^1 + \sum_{i=0}^{1-1} 2^{1-(i+1)} \cdot \text{char}(s_i) = 2 + (2^{1-(0+1)} \cdot 1) = 2 + 1 = 3$$

EXAMPLE OF HASHING TRICK

- We have defined: $\phi(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \text{char}(s_i)$
- $C = \{a,b\}$ where $\text{char}(a) = 0$ and $\text{char}(b) = 1$
- Let us consider $S=aa$. $\phi(S) := 2^2 + \sum_{i=0}^{2-1} 2^{2-(i+1)} \cdot \text{char}(s_i) = 4 + (2^{2-(0+1)} \cdot 0 + 2^{2-(1+1)} \cdot 0) = 4 + 0 = 4$
- Let us consider $S=ab$. $\phi(S) := 2^2 + \sum_{i=0}^{2-1} 2^{2-(i+1)} \cdot \text{char}(s_i) = 4 + (2^{2-(0+1)} \cdot 0 + 2^{2-(1+1)} \cdot 1) = 4 + 1 = 5$
- Let us consider $S=bb$. $\phi(S) := 2^2 + \sum_{i=0}^{2-1} 2^{2-(i+1)} \cdot \text{char}(s_i) = 4 + (2^{2-(0+1)} \cdot 1 + 2^{2-(1+1)} \cdot 1) = 4 + 2 + 1 = 7$
- Let us consider $S=aaa$
 $\phi(S) := 2^3 + \sum_{i=0}^{3-1} 2^{3-(i+1)} \cdot \text{char}(s_i) = 8 + (2^{3-(0+1)} \cdot 0 + 2^{3-(1+1)} \cdot 0 + 2^{3-(2+1)} \cdot 0) = 8$

FEATURE HASHING

- In Sklearn you have `sklearn.feature_extraction.FeatureHasher`

```
1 h2 = FeatureHasher(input_type='string')
2 D2 = [['we', 'went', 'there']]
3 f2 = h2.transform(D2)
4 f2
```

executed in 4ms, finished 17:05:33 2021-05-09

<1x1048576 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in Compressed Sparse Row format>

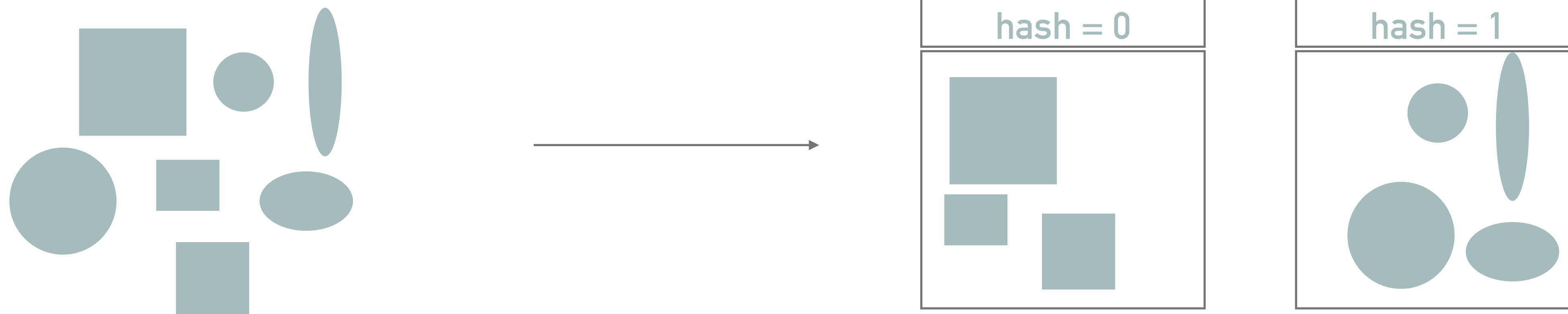
```
1 h2 = FeatureHasher(input_type='string', n_features=2048)
2 D2 = [['we', 'went', 'there']]
3 f2 = h2.transform(D2)
4 f2
```

executed in 3ms, finished 17:05:30 2021-05-09

<1x2048 sparse matrix of type '<class 'numpy.float64'>'
with 3 stored elements in Compressed Sparse Row format>

NEAREST NEIGHBOUR SEARCH

- Finding similar items in a big Dataset can be very expensive, in many applications the concept of ‘closest item’ from a query is fuzzy because a query can be represented as a vector in a space that is not ‘clear’ or ‘human readable’.
- A Hash function $\phi : \mathcal{X} \longrightarrow \mathbb{N}$ maps feature vectors to integers.
- A Hash Table $T : \mathbb{N} \longrightarrow \mathcal{X}^*$ maps integers to collections of feature vectors
- In our case we will consider the feature space $\mathcal{X} = \mathbb{R}^D$.
- Imagine that we want to group vectors into buckets of similar vectors:



SIMPLE HASH TABLE

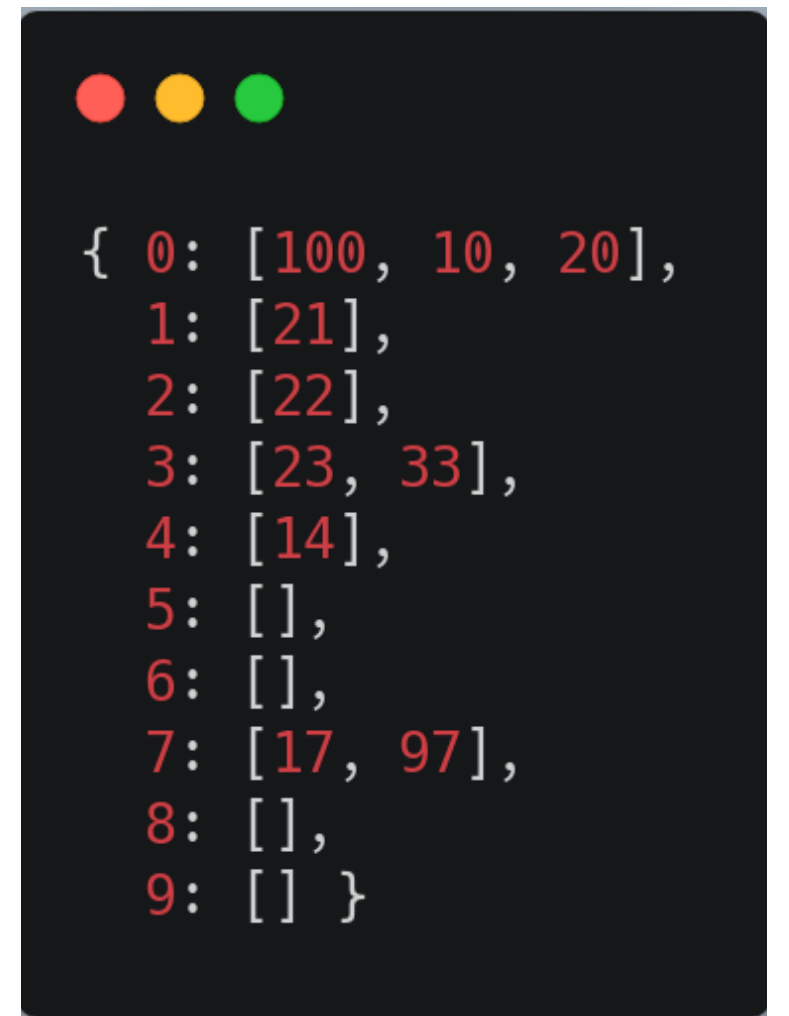
- The following function takes as input
 - value_l: list of input values
 - n_buckets: number of buckets
- Returns a hash_table
- Example:
 - input [100, 10, 14, 17, 97, 20, 21, 22, 23, 33]
- Note that numbers are in a bucket if when divided by the number of buckets they get the same remainder

```
def basic_hash_table(value_l, n_buckets):  
    def hash_function(value, n_buckets):  
        return int(value) % n_buckets  
  
    hash_table = {i:[] for i in range(n_buckets)}  
  
    for value in value_l:  
        hash_value = hash_function(value, n_buckets)  
        hash_table[hash_value].append(value)  
  
    return hash_table
```

```
{ 0: [100, 10, 20],  
  1: [21],  
  2: [22],  
  3: [23, 33],  
  4: [14],  
  5: [],  
  6: [],  
  7: [17, 97],  
  8: [],  
  9: [] }
```

PROBLEM WITH THE PREVIOUS HASH TABLE SOLUTION

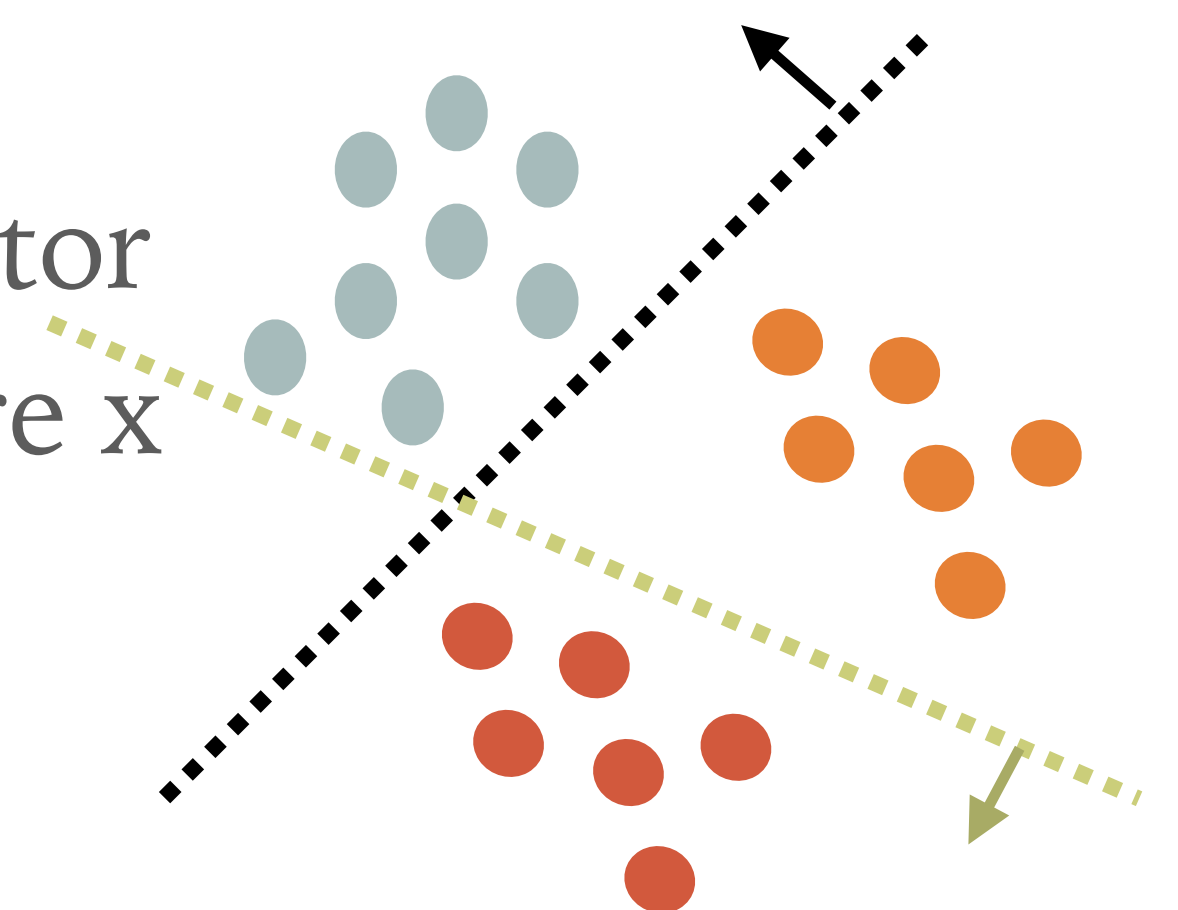
- The previous hash table is constructed using the % operation which does not take into account the distance between numbers.
- If we want numbers that are close to be in the same bucket we need to use 'spatial' information to create buckets, the % operator does not give us this type of information.
- To create a hashing function that is sensitive to the position of objects in the feature space we will partition the feature space in regions. We will assign to each region a number (or hash value).
- Since elements in the same region will be assigned the same hash value we will make our hash function sensitive to the location of our data.
- This is the basis of Locality Sensitive Hashing (LSH).



```
{ 0: [100, 10, 20],  
  1: [21],  
  2: [22],  
  3: [23, 33],  
  4: [14],  
  5: [],  
  6: [],  
  7: [17, 97],  
  8: [],  
  9: [] }
```

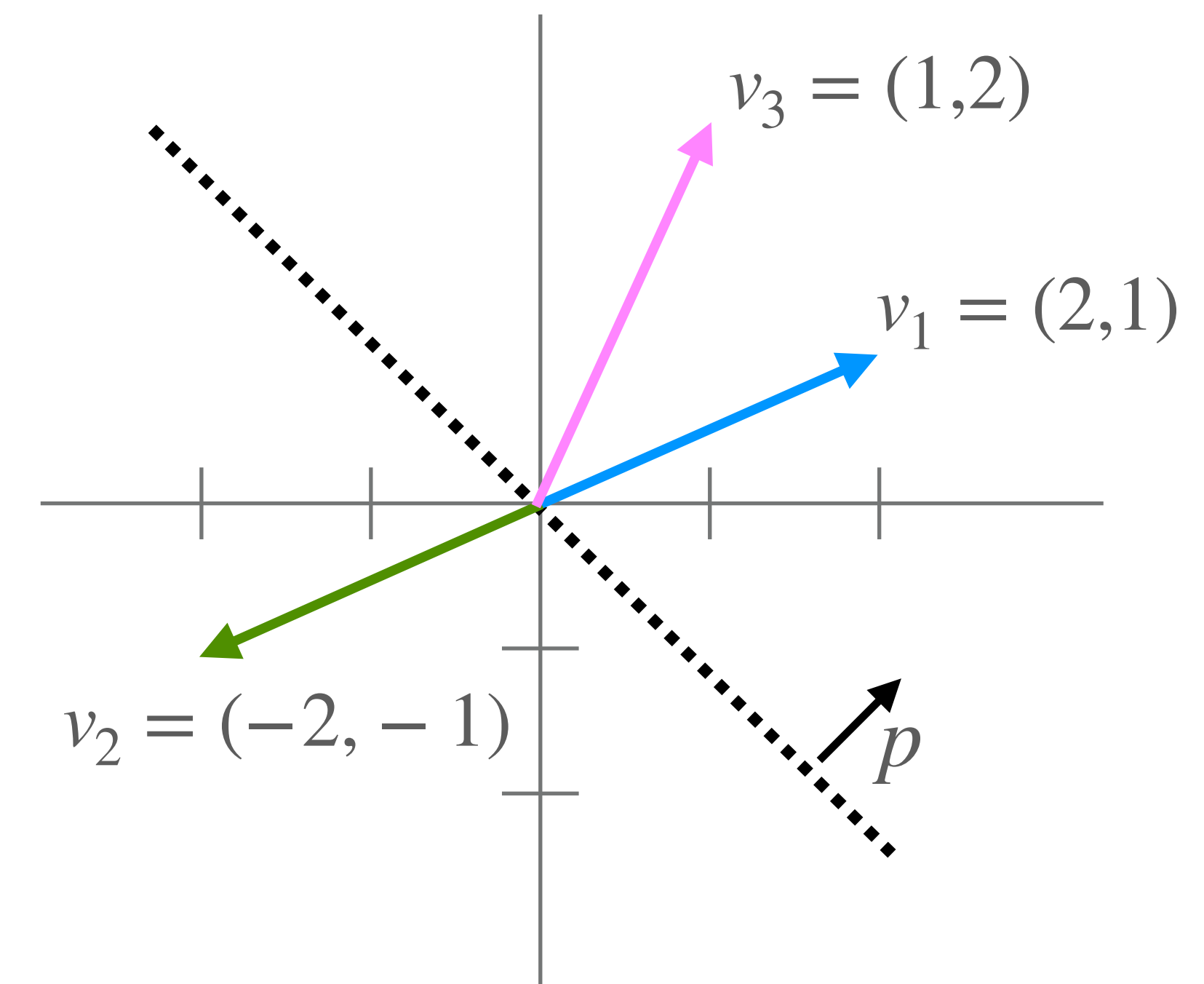
KEY IDEA FOR LSH: RELATIVE POSITION TO HYPERPLANES

- We can group points in the feature space by assigning an identifier build using the relative position of different planes that we can set up in the feature space.
- We consider that the “relative position” of a point with respect to a plane is a binary valued function that states whether a point is on the left or on the right of a hyperplane.
- A point x will be on the right of a hyperplane if the normal vector used to define the hyperplane points to the same side where x is located.
- A point x will be on the left of a hyperplane if the normal vector used to define the hyperplane points to the opposite side where x is located



REMEMBERING THE SCALAR PRODUCT

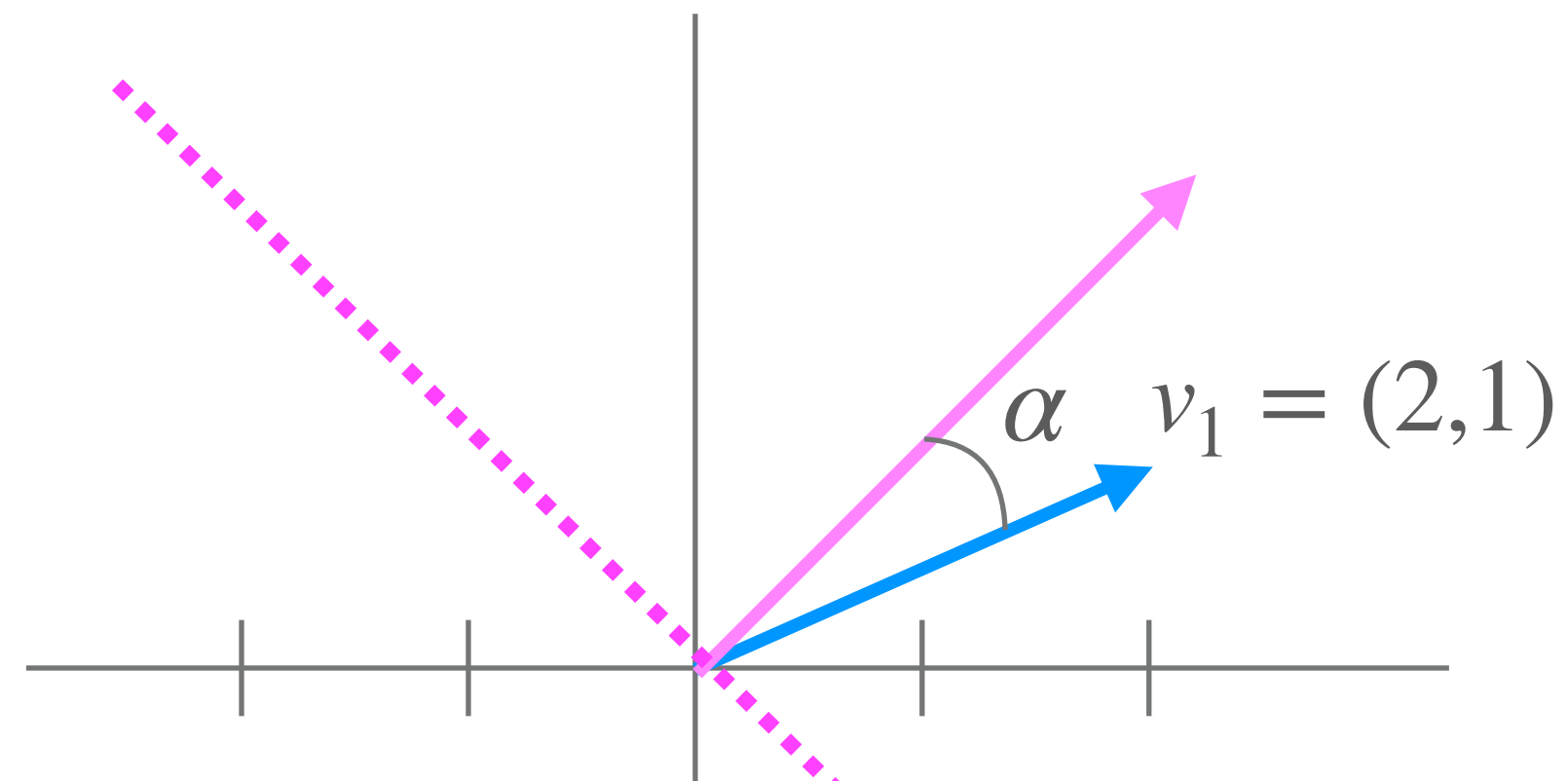
- Let us consider:
 - A plane defined by $p = (1,1)$
 - A vector $v_1 = (2,1)$
 - A vector $v_2 = (-2, -1)$
 - A vector $v_3 = (1,2)$
- Notice that the sign of the dot products:
 - $p \cdot v_1 = (1,1) \cdot (2,1)^T = 2 + 1 = 3$
 - $p \cdot v_2 = (1,1) \cdot (-2, -1)^T = -2 - 1 = -3$
 - $p \cdot v_3 = (1,1) \cdot (1,2)^T = 1 + 2 = 3$



RELATIVE POSITION COMPUTED WITH A SCALAR PRODUCT

- Let us consider a vector x and a hyperplane defined with a vector p .
- The dot product between p and x can be written as a sum over the products of the coordinates of the vectors or as a product of the norms of the vectors times the cosine of the angle that they form.

$$p \cdot x^T = \sum_{i=1}^D p_i x_i = \|p\| \cdot \|x\| \cdot \cos(\alpha)$$



RELATIVE POSITION COMPUTED WITH A SCALAR PRODUCT

➤ Key idea:

➤ The sign of the dot product $x \cdot p$ tells us:

- $p \cdot x^T > 0$ implies that x is in the right hand side of the plane defined by p
- $p \cdot x^T = 0$ implies that x is in the plane defined by p
- $p \cdot x^T < 0$ implies that x is in the left hand side of the plane defined by p

```
def side_of_plane(p,v):  
    dot_product = np.dot(p,v.T)  
    sign_dot_product = np.sign(dot_product)  
    sign_dot_product_scalar = sign_dot_product.item()  
    return sign_dot_product_scalar
```

FROM RELATIVE POSITIONS TO HASH VALUE

- Given a set of hyperplanes defined by normal vectors $p_1, \dots, p_K \in \mathbb{R}^D$ we can use the relative position across hyperplanes to find a hash value for any vector.
- Let $h_k(x; p_k) = \text{sign}(p_k \cdot x) = \begin{cases} 1 & p_k \cdot x \geq 0 \\ 0 & p_k \cdot x < 0 \end{cases}$
- Using this function h_k we define:

$$\phi(x; \{p_1, \dots, p_K\}) = \sum_{k=0}^{K-1} 2^k \cdot h_k(x; p_k)$$

EXAMPLE

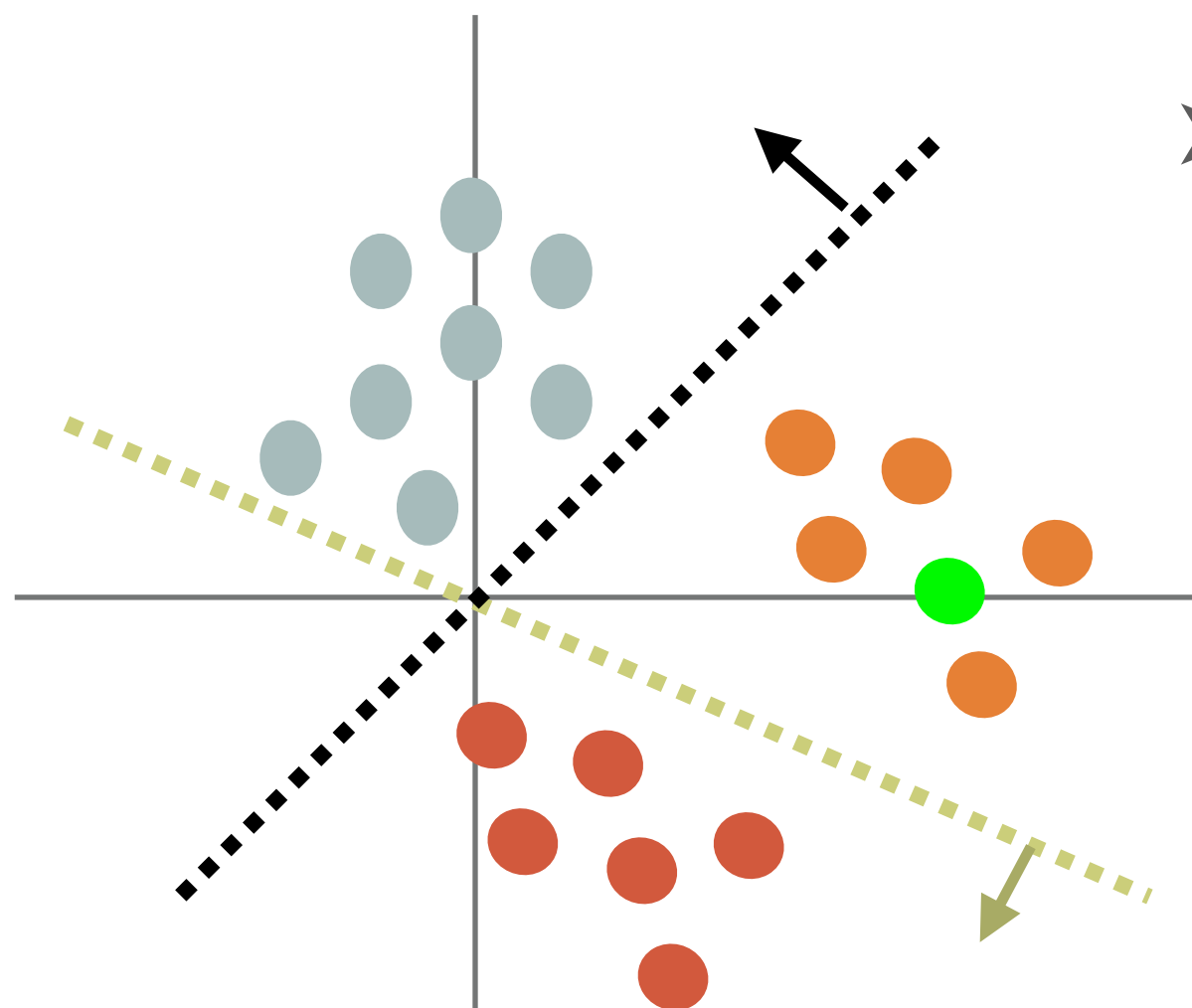
- Consider $p_1 = (-1, 1), p_2 = (-1, -1)$
- What is the hash value assigned to $x = (3, 0)$

- We need to know:

- $p_1 \cdot x^T = (-1, 1) \cdot (3, 0) = -3$

- $p_2 \cdot x^T = (-1, -1) \cdot (3, 0) = -3$

- Therefore: $\phi(x; \{p_1, p_2\}) = 2^0 \cdot 0 + 2^1 \cdot 0 = 0$



SUMMARY

- To compute a LSH hash we need:
 - K planes
 - A dot product function
 - A sign function
- We can use the dot product to verify if a point is in the left or the right hand side of a plane (by the sign of the dot product between the point and a normal vector to the plane).

```
def side_of_plane(p,v):  
    dot_product = np.dot(p,v.T)  
    sign_dot_product = np.sign(dot_product)  
    sign_dot_product_scalar = sign_dot_product.item()  
    return sign_dot_product_scalar
```

```
def hash_multiple_planes(p_list, v):  
    hash_value = 0  
    for i,p in enumerate(p_list):  
        sign = side_of_plane(p,v)  
        hash_i = 1 if sign >=0 else 0  
        hash_value += 2**i *hash_i  
  
    return hash_value
```

'VECTORIZED' VERSION

- Instead of calling `np.dot(p, V.T)` K times we can...
- Create a matrix containing the normal vectors to the planes:

$$P = \begin{bmatrix} - & - & -p_1 & - & - & - \\ - & - & -p_2 & - & - & - \\ - & - & -p_3 & - & - & - \end{bmatrix}$$

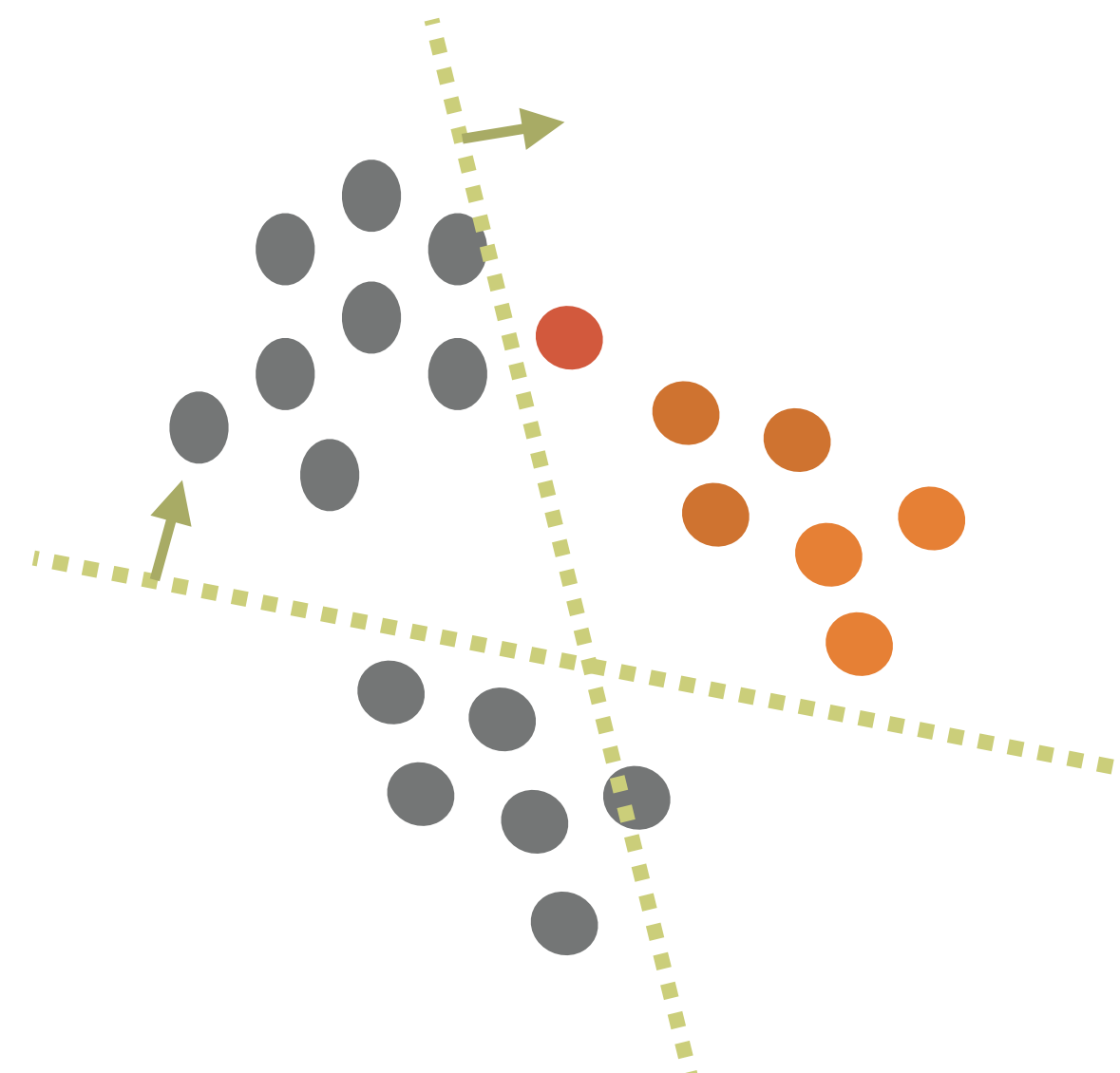
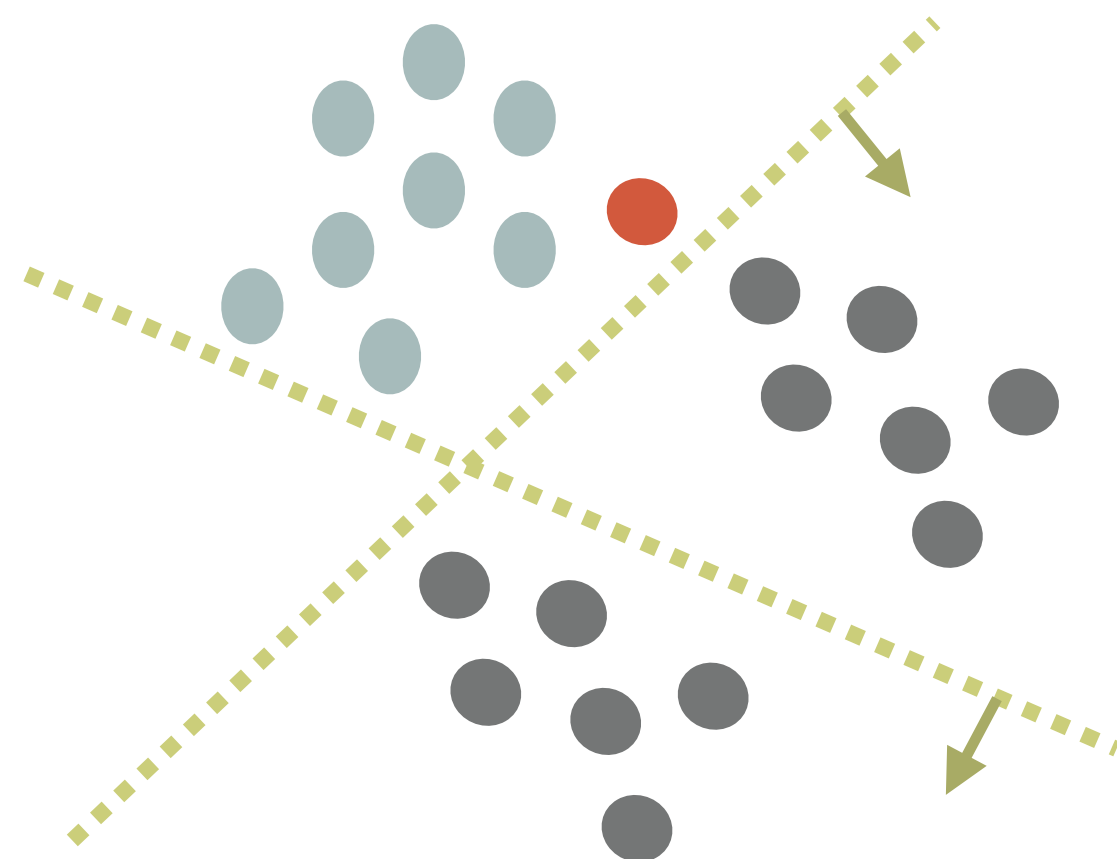
- Call `np.dot`, `np.sign` only once

```
def side_of_plane(p, v):  
    dot_product = np.dot(p, v.T)  
    sign_dot_product = np.sign(dot_product)  
    sign_dot_product_scalar = sign_dot_product.item()  
    return sign_dot_product_scalar
```

```
def hash_multiple_planes(p_list, v):  
    hash_value = 0  
    for i, p in enumerate(p_list):  
        sign = side_of_plane(p, v)  
        hash_i = 1 if sign >= 0 else 0  
        hash_value += 2**i * hash_i  
  
    return hash_value
```

HOW CAN WE FIND THE PLANES?

- We can create them at random!
- In practise multiple sets of planes are used and several hash tables are created.
- Let us consider U to be the number of 'universes' of randomly created planes.
- Then we can consider U different sets of planes defined by
$$P^{1:U} = [[p_1^1, \dots, p_K^1], [p_1^2, \dots, p_K^2], \dots, [p_1^U, \dots, p_K^U]]$$
- Each set of K planes defines a hash table, therefore, we can search not only in one hash table but in several !



LSH IN MULTIPLE UNIVERSES

- Let us consider U sets of planes

$$P^{1:U} = [[p_1^1, \dots, p_K^1], [p_1^2, \dots, p_K^2], \dots, [p_1^U, \dots, p_K^U]] = [P^1, \dots, P^U]$$

- Let us consider $T^u : \mathbb{N} \longrightarrow (\mathbb{R}^D)^*$ a hash table for universe $u \in \{1, \dots, U\}$.
Given an integer (that identifies a bucket value) this function will return all vectors assigned to that bucket.
- Given a query vector x^q we can compute a hash value for each of the universes and then use all the vectors in each of the hash buckets that we visit in each of the universes:

- The final set of vectors used to search the nearest neighbours: $\bigcup_{u=1}^U T^u (\phi(x^q; P^u))$