Text Representations: TF-IDF and Hashing Trick

Natural Language Processing

Daniel Ortiz Martínez, David Buchaca Prats

Motivation: K-Nearest Neighbours From a Query

- Closest K-nearest neighbours to a query: $X^{\mathrm{Knn}} = [x^{\mathrm{nn}_1}, ..., x^{\mathrm{nn}_K}]$
- That means $x \in X \Rightarrow \forall x^{\mathrm{nn}} \in X^{\mathrm{Knn}}, d(x, x_q) \geq \max d(x, x^{\mathrm{nn}})$
- This is related to the KNN algorithm
- How to define the feature vectors?
- How to measure distance?

TF-IDF Measure: Notation

- TF-IDF representation emphasizes the most relevant document words
- Let X be a set of documents (aka *corpus* or *dataset*)
- Let x be a document in X
- Let $w \in W$ be a word vocabulary of $|W|$ words
- Let X_w the set of documents containing word w

TF-IDF Measure: Motivation

- TF-IDF is a measure of the importance of a word to a document in a corpus
- Emphasize words that appear frequently in x: *term frequency*

$$\mathrm{tf}(w; x) = \frac{f_{w,x}}{\sum_{w' \in W} f_{w',x}}$$

- Emphasize words that appear rarely in X: *inverse document frequency*

$$\mathrm{idf}(w; X) = \frac{|X|}{|X_w|}$$

TF-IDF Measure: IDF Smooth

- IDF measure is typically smoothed to avoid numerical problems:

$$\mathrm{idf}(w; X) = \log\left(\frac{|X|}{1 + |X_w|}\right)$$

- Let us consider a corpus of 1 000 documents:
  - w appears in 100 documents: $\mathrm{idf}(w; X) = \log\left(\frac{1\,000}{1+100}\right) = 2.29$
  - w' appears in 10 documents: $\mathrm{idf}(w'; X) = \log\left(\frac{1\,000}{1+10}\right) = 4.51$

TF-IDF Measure: Calculation

- TF-IDF formula:

$$\mathrm{tfidf}(w; x, X) = \mathrm{tf}(w; x) \cdot \mathrm{idf}(w; X)$$

- A high weight in TF-IDF is reached by
  - a high word frequency in the given document x
  - a low document frequency of the word in the whole collection of documents

TF-IDF Vector

- The TF-IDF measure can be used to build feature vectors
- Each document $x \in X$ can be represented by means of a vector of $|W|$ positions
- For the w'th position, the vector contains the corresponding TF-IDF measure: $\mathrm{tfidf}(w; x, X)$

TF-IDF in Scikit-Learn

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()

X = vectorizer.fit(['the cat went to the beach',
                    'the players were happy talking',
                    'the cat liked the food'])

print(vectorizer.get_feature_names_out())

print(vectorizer.idf_)
```

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^{nn} of a collection that is closest to x^q
- Formally, we are interested in finding the nearest neighbour x^{nn}

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

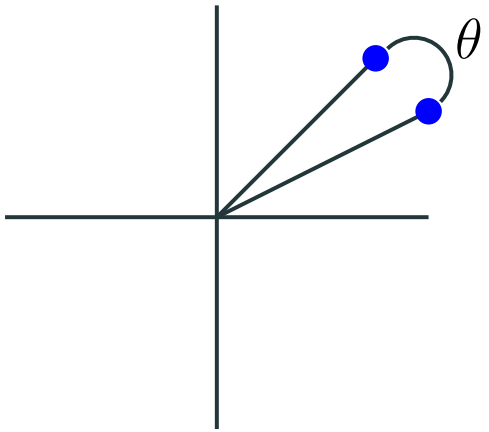
Cosine Similarity

- The cosine of two non-zero n-dimensional vectors can be derived from the Euclidean dot product:

$$p^T \cdot q = ||p||\,||q|| \cos(\theta)$$

- The cosine similarity, cs, is then given by:

$$\mathrm{cs}(p, q) = \cos(\theta) = \frac{p^T \cdot q}{||p||\,||q||} = \frac{\sum_{i=1}^{D} p_i \cdot q_i}{\sqrt{\sum_{i=1}^{D} p_i^2} \cdot \sqrt{\sum_{i=1}^{D} q_i^2}}$$



Cosine Distance

- cs measures similarity but we need to measure distances
- Cosine distance is defined as the complement of cosine similarity:

$$\mathrm{cd}(p, q) = 1 - \mathrm{cs}(p, q)$$

Euclidean Distance

- 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

$$||(x-y)||_2 = \sqrt{\sum_{d=1}^{D}(x_d - y_d)^2} = \sqrt{(x-y)^T(x-y)}$$

When to Choose Cosine over Euclidean Distance

- Cosine distance is a metric generally used when the magnitude of the vectors does not matter
- Example: working with text represented by word counts
- We could assume that if a given word is more frequent in document 1 than in document 2, this means that the documents are different
- But it could also be the case that the documents are of uneven length
- Since cosine distance pays attention to vector angles instead of magnitudes, it can correct for this

Finding Nearest Neighbours Fast

- 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.
- 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.

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
- Alternative to storing a vocabulary: the hashing trick
- Let C be the set of characters used in the language, where $\alpha := |C|$
- Let char : $C \longrightarrow \mathbb{N}$ a function that maps a character from the alphabet to an integer
- We can map a string S to an integer using θ : $C^* \longrightarrow \mathbb{N}$ defined as*:

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

*this function will return a different integer for each different string

The Hashing Trick

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

$$\theta(S) := \left(\alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \mathrm{char}(s_i) \right) \mod (M)$$

- This function may not produce a different integer for each different string but for large M values the collision probability will be very small
- The value of M can be arbitrarily big, making the function injective

The Hashing Trick: Example

- We have defined: $\theta(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \mathrm{char}(s_i)$
- $C = \{\mathrm{a}, \mathrm{b}\}$ where $\mathrm{char}(\mathrm{a}) = 0$ and $\mathrm{char}(\mathrm{b}) = 1$
- Let us consider $S = \mathrm{a}$

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

- Let us consider $S = \mathrm{b}$

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

The Hashing Trick: Example

- We have defined: $\theta(S) := \alpha^{|S|} + \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \cdot \mathrm{char}(s_i)$
- $C = \{\mathrm{a}, \mathrm{b}\}$ where $\mathrm{char}(\mathrm{a}) = 0$ and $\mathrm{char}(\mathrm{b}) = 1$
- Let us consider $S = \mathrm{ab}$

$$\theta(S) := 2^2 + \sum_{i=0}^{2-1} 2^{2-(i+1)} \cdot \mathrm{char}(s_i) = 4 + (2^{2-(0+1)} \cdot 0 + 2^{2-(1+1)} \cdot 1) = 5$$

- Let us consider $S = \mathrm{aaa}$

$$\theta(S) := 2^3 + \sum_{i=0}^{3-1} 2^{3-(i+1)} \cdot \mathrm{char}(s_i) = 8 + (2^{3-(0+1)} \cdot 0 + 2^{3-(1+1)} \cdot 0 + 2^{3-(2+1)} \cdot 0) = 8$$

Feature Hashing

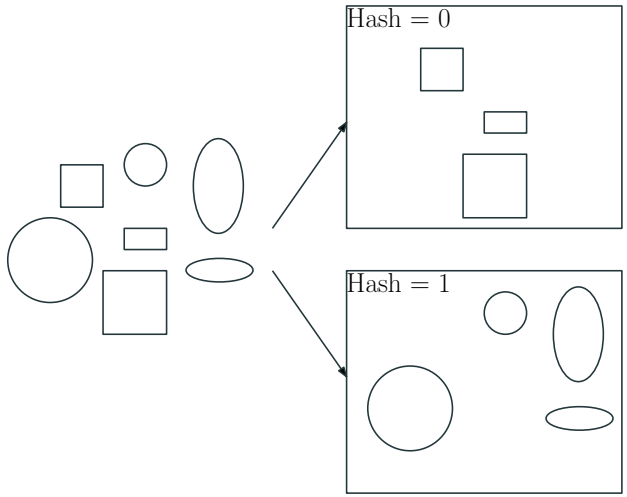
Scikit-learn provides the `sklearn.feature_extraction.FeatureHasher` function

```
import sklearn.feature_extraction
h = sklearn.feature_extraction.FeatureHasher(input_type='string')
d = [['we','went','there']]
f = h.transform(d)

# Alternatively...
h = sklearn.feature_extraction.FeatureHasher(input_type='string', n_features=2048)
d = [['we','went','there']]
f = h.transform(d)
```

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 non human readable space
- A hash function $\theta : \mathcal{X} \longrightarrow \mathbb{N}$ 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

- Example of a simple hashing function

```
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
```

- Example:

```
basic_hash_table([100,10,14,17,97,20,21,22,23,33], 10)

 {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 close numbers in the same bucket we need to use spatial information, 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)

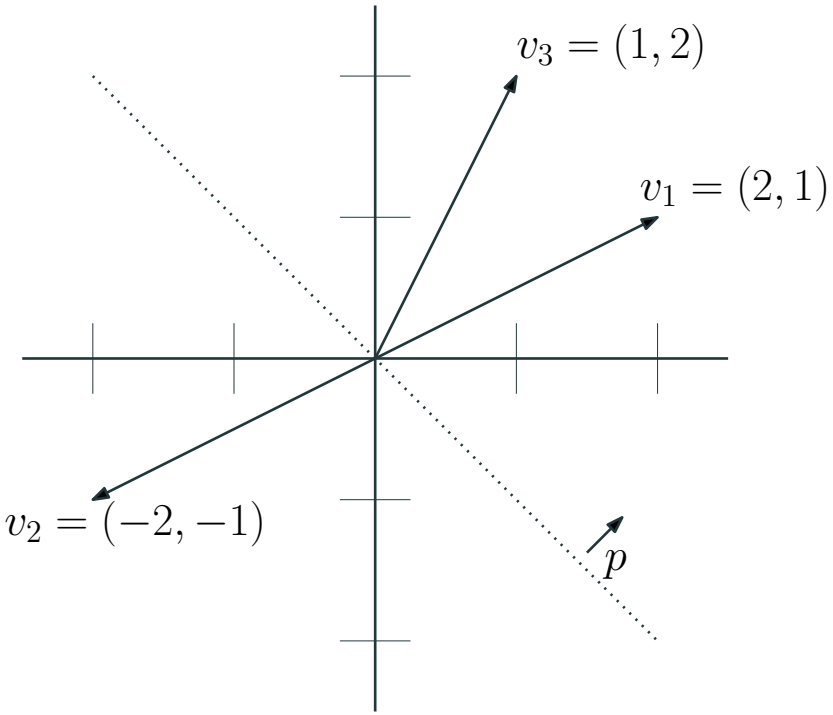
Key Idea for LSH: Relative Position to Hyperplanes

- We can group points in the feature space by assigning an identifier built 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 oposite side where x is located

Remembering the Scalar Product

- Given the following plane and three vectors:

$$p = (1, 1)$$
$$v_1 = (2, 1)$$
$$v_2 = (-2, -1)$$
$$v_3 = (1, 2)$$



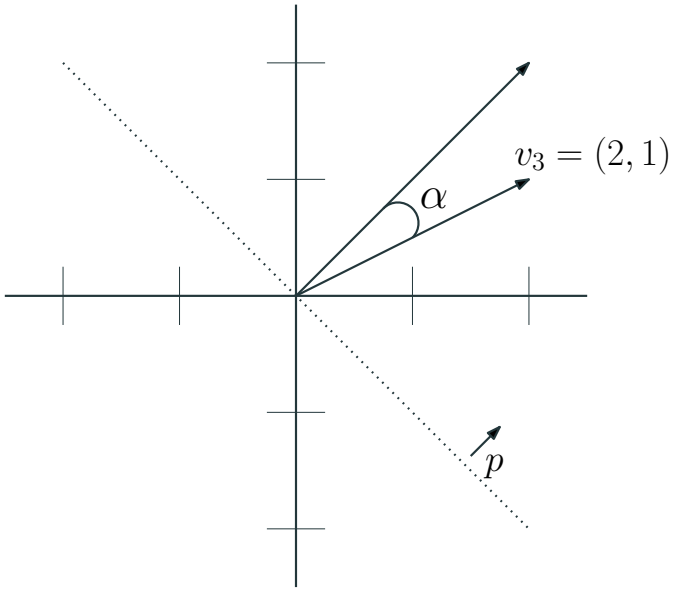
- Result of the dot products

$$p \cdot v_1 = (1, 1) \cdot (2, 1)^T = 2 + 1 = 3$$
$$p \cdot v_2 = (1, 1) \cdot (-2, -1)^T = -2 - 1 = -3$$
$$p \cdot v_3 = (1, 1) \cdot (1, 2)^T = 1 + 2 = 3$$

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

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



Relative Position Computed with a Scalar Product

- The sign of the dot product $p \cdot x^T$ tells us:
  - $p \cdot x^T > 0$ implies that x is in the right 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 side of the plane defined by p

```
def side_of_plane(p,x):
    dot_product = np.dot(p, x.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 $p_1, ..., p_K \in \mathbb{R}$ 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:

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

Example

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

$$
\begin{aligned}
p_1 \cdot x^T &= (-1, 1) \cdot (3, 0) = -3 \\
p_2 \cdot x^T &= (-1, -1) \cdot (3, 0) = -3 \\
\theta(x, \{p_1, p_2\}) &= 2^0 \cdot 0 + 2^1 \cdot 0 = 0
\end{aligned}
$$

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 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

def hash_multiple_planes(p_list, v):
    hash_value = 0
    for i, p in enumerate(p_list):
        sign = side_of_plane(p_list)
        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

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

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

- 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

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

- Let us consider $T : \mathbb{N} \longrightarrow (\mathbb{R}^D)^*$ a hash table for universe $u \in \{1, ..., U\}$. Given an integer (that identifies a bucket value) this function will return all vectors assigned to that bucket
- Given a query vector x 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, producing a set of vectors:

$$\bigcup_{u=1}^{U} T^u(\theta(x^q; P^u))$$