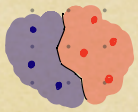
Vectorized KNN



How to solve KNN without loops

Pavel Larionov

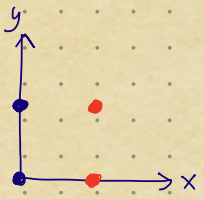
Train Set

features

$$\begin{matrix} a_1 \rightarrow \\ a_2 \rightarrow \\ a_3 \rightarrow \\ a_4 \rightarrow \end{matrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

labels

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$



Test Set

features

$$\begin{matrix} b_1 \rightarrow \\ b_2 \rightarrow \end{matrix} \begin{bmatrix} -1 & 0 \\ 1 & 0.5 \end{bmatrix}$$

labels

$$\begin{bmatrix} ? \\ ? \end{bmatrix}$$

- Instead of calculating $|a_i - b_j|$ for each pair, we will calculate $(a_i - b_j)^2$ for each pair.

  for example $[0, 0]$ ← a_i , b_j → for example $[-1, 0]$

- $(a_i - b_j)^2$ for vectors is $(a_i - b_j)^T (a_i - b_j)$

- $(a_i - b_j)^T (a_i - b_j) = a_i^T a_i - b_j^T a_i - a_i^T b_j + b_j^T b_j$

  $a_i^T a_i$: vector i from train data

  $-b_j^T a_i - a_i^T b_j = -2 b_j a_i$

  $b_j^T b_j$: vector j from test data

- Notice that for each test vector $(b_j ; \forall j)$ we use the same term $a_i^T a_i$. We want to calculate it once and re-use.

  → for example: $a_4^T a_4 = [1, 1]^T [1, 1] = 1 \cdot 1 + 1 \cdot 1 = 2$

- Let's calculate $a_i^T a_i$ for all i's at once:

A (train data)

$$a_1 \to \begin{bmatrix} 0 & 0 \end{bmatrix} \to [0,0]^T[0,0] = 0^2+0^2$$
$$a_2 \to \begin{bmatrix} 0 & 1 \end{bmatrix} \to [0,1]^T[0,1] = 0^2+1^2 \Rightarrow \begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \end{bmatrix}$$
$$a_3 \to \begin{bmatrix} 1 & 0 \end{bmatrix} \to [1,0]^T[1,0] = 1^2+0^2$$
$$a_4 \to \begin{bmatrix} 1 & 1 \end{bmatrix} \to [1,1]^T[1,1] = 1^2+1^2$$

Basically we apply element-wise square function and then sum the values of a row.

(keeps it as a column vector)

numpy code: `np.sum(np.square(A), axis=1, keepdims=True)`

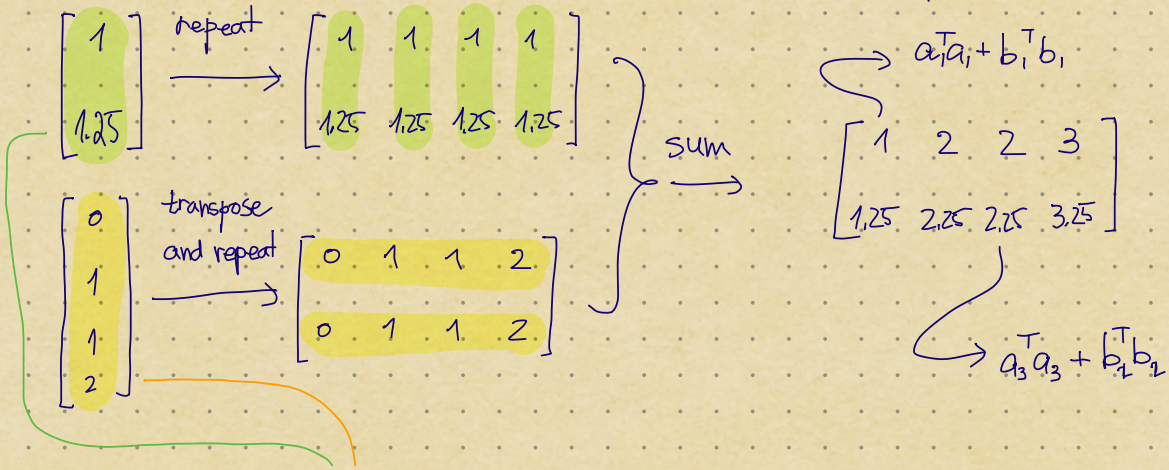
torch code: `torch.sum(torch.square(A_torch), dim=1, keepdims=True)`

- The same way calculate $b_j^T b_j$ for all j's in the test matrix:

B

$$b_1 \to \begin{bmatrix} -1 & 0 \end{bmatrix} \to \begin{bmatrix} 1 \\ 1.25 \end{bmatrix}$$
$$b_2 \to \begin{bmatrix} 1 & 0.5 \end{bmatrix}$$

- Reminder: we want to calculate $a_i^T a_i - 2 a_i^T b_j + b_j^T b_j$ for every possible pair of i and j. First let's focus on calculating $a_i^T a_i + b_j^T b_j$ for every possible i, j.



numpy code:

```
aTa = np.sum(np.square(A), axis=1, keepdims=True)
bTb = np.sum(np.square(B), axis=1, keepdims=True)
aTa.T + bTb
```

How does this [4×1] + [1×4] work?

Broadcasting! [4×1] + [1×4] → [4×4]

torch:

```
aTa_torch = torch.sum(torch.square(A_torch), dim=1, keepdims=True)
bTb_torch = torch.sum(torch.square(B_torch), dim=1, keepdims=True)
aTa_torch.T + bTb_torch
```

- Now calculate the last term $-2a_i^T b_j$ for each possible pair of i, j. It's done by matrix multiplication

$$\underbrace{\begin{bmatrix} -1 & 0 \\ 1 & 0.5 \end{bmatrix}}_{B\ (test)\ 2\times 2} \times \underbrace{\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}}_{A^T\ (train)\ 2\times 4} = \underbrace{\begin{bmatrix} 0 & 0 & -1 & -1 \\ 0 & 0.5 & 1 & 1.5 \end{bmatrix}}_{BA^T}$$

$a_1^T b_4$ → top-right entry; $a_3^T b_2$ → bottom-left entry

```
np.matmul(B, A.T)
torch.matmul(B_torch, A_torch.T)
```

- Now multiply by -2 and sum those two matrices:

numpy:

```
aTa.T + bTb -2*np.matmul(B, A.T)
```

torch:

```
aTa_torch.T + bTb_torch -2*torch.matmul(B_torch, A_torch.T)
```

$$\begin{bmatrix} 1 & 2 & 2 & 3 \\ 1.25 & 2.25 & 2.25 & 3.25 \end{bmatrix} - 2 \times \begin{bmatrix} 0 & 0 & -1 & -1 \\ 0 & 0.5 & 1 & 1.5 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 & 5 \\ 1.25 & 1.25 & 0.25 & 0.25 \end{bmatrix}$$

Matix of squared distances of all possible pairs

Now, when we have the squared distances matrix, we want to find k-smallest distances for each row.

Actually, we want to find the indexes of the k-smallest values in each row.

dist →

$$\begin{bmatrix} 1 & 2 & 4 & 5 \\ 1.25 & 1.25 & 0.25 & 0.25 \end{bmatrix}$$

↑

You may apply element-wise sqrt to convert it to distances.

numpy:

```
k_nearest_idx = np.argsort(dist)[:,:k]
k_nearest = labels[k_nearest_idx]
```

torch:

```
k_nearest_idx = torch.argsort(dist_torch)[:,:k]
k_nearest = labels[k_nearest_idx]
```

↳ an array with the labels of the k-nearest neighbors.

you can use

```
from scipy.stats import mode
mode(k_nearest, axis=1).mode
```

to make the final prediction