K-Nearest Neighbours from Scratch

Bhanu Prasanna Koppolu

Table of contents

> **Note**
>
> Full implementation available at GitHub - ML from Scratch

K-Nearest Neighbours (KNN)

KNN can work both as a classification and as a regression algorithm for supervised tasks.

What is KNN?

There is no specific training that is required, because KNN just looks at: **close points in the input space should have similar outputs**.

Distance Metrics

For this we need a Distance equation and there are many choices:

1. **Euclidean** - Most commonly used
2. **Manhattan**
3. **Minkowski**
4. And many more

The Algorithm

1. For each training point $i = 1, \dots, m$, compute the distance:

$$d_i = D(x_{\text{query}}, x^{(i)})$$

2. Sort the training points by distance d_i
3. Take indices of the k nearest neighbours:

$$N_k(x_{\text{query}}) = \{i_1, i_2, \dots, i_k\}$$

   Such that $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$

4. Do classification or regression using their $y^{(i)}$ values

Classification

We have $y_i \in \{1, 2, \dots, C\}$. For a query x we find its k neighbours:

$$N_k(x) = \{i_1, \dots, i_k\}$$

Unweighted Majority Vote (Classic KNN)

1. For each class c, count how many of the k neighbours have label c:

$$n_c(x) = \sum_{i \in N_k(x)} \mathbf{1}\{y_i = c\}$$

   where $\mathbf{1}$ is the indicator function (1 if true, 0 if false)

2. Choose the class with the highest count:

$$\hat{y} = \arg \max_{c \in \{1,\dots,C\}} n_c(x)$$

So, the prediction is simply the most frequent class among the k neighbours.

Also can be interpreted as:

$$\hat{P}(y = c|x) = \frac{n_c(x)}{k}$$

Distance Weighted KNN Classification

We want closer neighbours to have more influence.

The weight for each neighbour:

$$w_i = \frac{1}{d(x, x_i)^p + \epsilon}$$

where p is a power, ϵ is to prevent division by zero.

Weighted count:

$$n_c(x) = \sum_{i \in N_k(x)} w_i \cdot \mathbf{1}\{y_i = c\}$$

The $\hat{y}$ remains the same (argmax).

Probability estimate:

$$\hat{P}(y = c|x) = \frac{n_c(x)}{\sum_{c'} n_{c'}(x)}$$

where c' runs over all classes.

Regression

Now, we have target values $y_i \in \mathbb{R}$.

Same neighbour set: $N_k(x) = \{i_1, \ldots, i_k\}$

Unweighted KNN Regression

$$\hat{y}(x) = \frac{1}{k} \sum_{i \in N_k(x)} y_i$$

It is to find the K Nearest target values and average them.

Distance Weighted KNN Regression

You weigh in nearer points more.

Given weights $w_i \geq 0$ for $i \in N_k(x)$:

$$w_i = \frac{1}{d(x, x_i)^p + \epsilon}$$

Then, normalize weights:

$$\bar{w}_i = \frac{w_i}{\sum_{j \in N_k(x)} w_j}$$

Predicted value:

$$\hat{y}(x) = \sum_{i \in N_k(x)} \bar{w}_i \cdot y_i$$

This is the weighted average of neighbour targets.

Note on k

When $k = 1$, weighted and unweighted versions become identical. k is a hyperparameter.

Classification Implementation

```
class knn_unweighted_classification:
    # K - Nearest Neighbours for Unweighted Classification

    def __init__(self, k):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train.copy()
        self.y_train = y_train.copy()
        return self

    def predict(self, X_test):

        y_hat = []
        for i in range(X_test.shape[0]):
            # Step 1: Compute distance to all training points
            dis_tt = np.sqrt(np.sum((self.X_train - X_test[i]) ** 2, axis=1))

            # Step 2: Compute K Nearest Neighbours indexes
            k_idx = np.argpartition(dis_tt, self.k, axis=0)[:self.k]

            # Step 3: Find the labels for the respective indexes in y_train
            labels_idx = self.y_train[k_idx]

            # Step 4: Pick the highest count from labels_idx
            count_highest_label = np.bincount(labels_idx)
            count_highest_label = np.argmax(count_highest_label)

            # Step 5: Append the highest count label to y_hat
            y_hat.append(count_highest_label)

        return y_hat

class knn_weighted_classification:

    def __init__(self, k, p=2, epsilon=1e-6):
        self.k = k
        self.p = p
        self.epsilon = epsilon

    def fit(self, X_train, y_train):
        self.X_train = X_train
```

```
        self.y_train = y_train
        return self

    def predict(self, X_test):

        y_hat = []
        for i in range(X_test.shape[0]):
            # Step 1: Calculate the distance to all training points
            dis_tt = np.sqrt(np.sum((self.X_train - X_test[i]) ** 2, axis=1))

            # Step 2: Find the indexes k nearest neighbours
            k_idx = np.argpartition(dis_tt, self.k, axis=0)[:self.k]

            # Step 3: Extract the distances of only those k neighbours
            dis_tt = dis_tt[k_idx]

            # Step 4: Calculate the weights only for those k neighbours
            w = 1 / (np.power(dis_tt, self.p) + self.epsilon)

            # Step 5: Weighted voting
            labels_idx = self.y_train[k_idx]
            wv = np.bincount(labels_idx, weights=w) # I didn't know this, awesome.

            # Step 6: Argmax
            label = np.argmax(wv)

            y_hat.append(label)

        return y_hat
```

Regression Implementation

```
class knn_unweighted_regression:

    def __init__(self, k=5):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
```

```
        return self

    def predict(self, X_test):

        y_hat = []
        for i in range(X_test.shape[0]):
            # Step 1: Calculate the distance from all training points
            dis_tt = np.sqrt(np.sum((self.X_train - X_test[i]) ** 2, axis=1))

            # Step 2: Select k nearest neighbours
            ks_idx = np.argpartition(dis_tt, self.k)[:self.k]

            # Step 3: Get the values from y_train for the k idx
            values_idx = self.y_train[ks_idx]

            # Step 4: Append the mean of each row containing the values of k nearest neighbou
            y_hat.append(np.mean(values_idx))

        return y_hat


class knn_weighted_regression:

    def __init__(self, k=5, p=2, epsilon=1e-6):
        self.k = k
        self.p = p
        self.epsilon = epsilon

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
        return self

    def predict(self, X_test):

        y_hat = []
        for i in range(X_test.shape[0]):
            # Step 1: Compute the distance from all training points
            dis_tt = np.sqrt(np.sum((self.X_train - X_test[i]) ** 2, axis=1))

            # Step 2: Find the indexes of k nearest neighbours
            ks_idx = np.argpartition(dis_tt, self.k)[:self.k]
```

```
            dis_tt = dis_tt[ks_idx]
            values_idx = self.y_train[ks_idx]

            # Step 3: Calculate the weight
            w = 1 / (((dis_tt) ** self.p) + self.epsilon)

            # Step 4: Calculate the weighted average of their target values
            wa = np.average(values_idx, weights=w)

            # Step 5: Append the wa to y_hat
            y_hat.append(wa)

        return y_hat
```