Ten algorithms

Daniel Walther Berns

December 4, 2024

Chapter 1

KNN

1.1 Introduction

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point.

1. Non-parametric models do not make assumptions about the underlying distribution of the data, which can make them more flexible but also potentially more computationally expensive.

2. Supervised learning means that the algorithm is trained on labeled data, where the correct classifications are already known, and uses this training to make predictions on new, unlabeled data.

3. In KNN, proximity refers to the distance between data points in a feature space, where features are the measurable characteristics of the data points. The k in KNN refers to the number of nearest neighbors that are considered when making a prediction. For example, if k=3, the algorithm would consider the three closest data points to the one being classified, and assign the class label that is most frequent among those three.

4. While it is true that KNN can be used for both classification and regression problems, it is worth mentioning that the algorithm is more commonly used for former working off the assumption that similar points can be found near one another. In regression problems, KNN would predict a numerical value for the target variable, rather than a class label.

5. For classification problems, a class label is assigned on the basis of a majority vote, i.e. the label that is most frequently represented around a given data point is used. While this is technically considered "plurality voting", the term, "majority vote" is more commonly used in literature. The distinction between these terminologies is that "majority voting" technically requires a majority of greater than 50%, which primarily works when there are only two categories. When you have multiple classes, four categories, you don't necessarily need 50% of the vote to make a conclusion about a class; you could assign a class label with the greatest vote.

6. Regression problems use a similar concept as classification problem, but in this case, the average of the k nearest neighbors is taken to make a prediction about a classification. The main distinction here is that classification is used for discrete values, whereas regression is used with continuous ones.

7. It's also worth noting that the KNN algorithm is also part of a family of "lazy learning" models, meaning that it only stores a training dataset versus undergoing a training stage. This also means that all the computation occurs when a prediction is being made. Since it heavily relies on memory to store all its training data, it is also referred to as an instance-based or memory-based learning method.

8. Evelyn Fix and Joseph Hodges are credited with the initial ideas around the KNN model in this 1951 paper while Thomas Cover expands on their concept in his article "Nearest Neighbor Pattern Classification." While it's not as popular as it once was, it is still one of the first algorithms one learns in data science due to its simplicity and accuracy. However, as a dataset grows, KNN becomes increasingly inefficient, compromising overall model performance. It is commonly used for simple recommendation systems, pattern recognition, data mining, financial market predictions, intrusion detection, and more.

1.2 KNN Code

```
import numpy as np

class KNN:
    """
    This implementation defines a KNN class
    that takes a value k as a parameter
    during initialization.
    The fit method is used to train the model
    on a training set X and corresponding labels y.

    The predict method takes a set of
    data points X and returns predicted labels
    for each data point.

    In this implementation, we take advantage
    of numpy's ability to perform operations
    on arrays element-wise to vectorize
    the calculation of distances between X
    and self.X_train.

    First, we reshape X and self.X_train using
    np.newaxis to create a new axis so that we
    can perform broadcasting, which allows
    numpy to perform operations on arrays
    with different shapes.
```

```
    Then, we calculate the squared differences
    between each data point in X and each
    training example in self.X_train
    using the ** operator, sum over the
    feature dimension using np.sum, and take
    the square root using np.sqrt to obtain
    the distances.

    Next, we use np.argsort to obtain the
    indices of the k smallest distances
    along the second axis (axis 1), corresponding
    to the k nearest neighbors for each
    data point in X.

    We then extract the labels of the
    k nearest neighbors from self.y_train
    using numpy's array indexing syntax,
    and calculate the mode of the labels
    for each data point using np.apply_along_axis
    with a lambda function that applies np.bincount
    and argmax along the first axis (axis 1).

    Finally, we return the predicted labels y_pred,
    which is a numpy array with shape (n_samples,)
    containing the predicted label for each
    data point in X.
    """
    def __init__(self, k):
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        distances =
        np.sqrt(
            np.sum((X[:, np.newaxis, :] - self.X_train) ** 2,
                   axis=2))
        neighbors = np.argsort(distances, axis=1)[:, :self.k]
        labels = self.y_train[neighbors]
        y_pred = np.apply_along_axis(
            lambda x: np.bincount(x).argmax(),
            axis=1,
            arr=labels)
        return y_pred
```

Chapter 2

DBScan

2.1 Introduction

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. Proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu in 1996, it is a density-based clustering algorithm. DBSCAN assumes that clusters are dense regions separated by areas of lower point density.

It groups 'densely grouped' data points into a single cluster. It can identify clusters in large spatial datasets by looking at the local density of the data points. The most exciting feature of DBSCAN clustering is that it is robust to outliers. It also does not require the number of clusters to be told beforehand, unlike K-Means, where we have to specify the number of centroids.

DBSCAN requires only two parameters: `epsilon` and `min_points`:

1. `epsilon` is the radius of the circle to be created around each data point to check the density;

2. `min_points` is the minimum number of data points required inside that circle for that data point to be classified as a Core point.

In higher dimensions the circle becomes hypersphere, `epsilon` becomes the radius of that hypersphere, and `min_points` is the minimum number of data points required inside that hypersphere.

DBSCAN creates a circle of `epsilon` radius around every data point and classifies them into Core point, Border point, and Noise. A data point is a Core point if the circle around it contains at least '`min_points`' number of points. If the number of points is less than `min_points`, then it is classified as Border Point, and if there are no other data points around any data point within `epsilon` radius, then it treated as Noise.

For locating data points in space, DBSCAN uses Euclidean distance, although other methods can also be used (like great circle distance for geographical data). It also needs to scan through the entire dataset once, whereas in other algorithms we have to do it multiple times.

2.1.1 Reachability and Connectivity

These are the two concepts to understand before moving further. Reachability states if a data point can be accessed from another data point directly or indirectly, whereas Connectivity states whether two data points belong to the same cluster or not. In terms of reachability and connectivity, two points in DBSCAN can be referred to as:

1. Directly Density-Reachable: A point X is directly density-reachable from point Y with respect to `epsilon`, `min_points` if,
   (a) X belongs to the neighborhood of Y, i.e, dist(X, Y) <= `epsilon`
   (b) Y is a Core point
2. Density-Reachable: A point X is density-reachable from point Y w.r.t `epsilon`, `min_points` if there is a chain of points $p_1, p_2, p_3, ..., p_n$, $p_1 = X$ and $p_n = Y$ such that p_{i+1} is directly density-reachable from p_i.
3. Density-Connected: A point X is density-connected from point Y w.r.t `epsilon` and `min_points` if there exists a point O such that both X and Y are density-reachable from O w.r.t to `epsilon` and `min_points`.

Parameter Selection in DBSCAN Clustering DBSCAN is very sensitive to the values of `epsilon` and `min_points`. Therefore, it is very important to understand how to select the values of `epsilon` and `min_points`. A slight variation in these values can significantly change the results produced by the DBSCAN algorithm.

The value of `min_points` should be at least one greater than the number of dimensions of the dataset, i.e.,

`min_points`>=Dimensions+1.

It does not make sense to take `min_points` as 1 because it will result in each point being a separate cluster. Therefore, it must be at least 3. Generally, it is twice the dimensions. But domain knowledge also decides its value.

The value of `epsilon` can be decided from the K-distance graph. The point of maximum curvature (elbow) in this graph tells us about the value of `epsilon`. If the value of `epsilon` chosen is too small then a higher number of clusters will be created, and more data points will be taken as noise. Whereas, if chosen too big then various small clusters will merge into a big cluster, and we will lose details.

2.2 Abstract algorithm

The DBSCAN algorithm can be abstracted into the following steps:[4]

1. Find the points in the `epsilon` neighborhood of every point, and identify the Core points with more than `min_points` neighbors.
2. Find the connected components of Core points on the neighbor graph, ignoring all non Core points.

3. Assign each non Core point to a nearby cluster if the cluster is an `epsilon` neighbor, otherwise assign it to noise. A naive implementation of this requires storing the neighborhoods in step 1, thus requiring substantial memory. The original DBSCAN algorithm does not require this by performing these steps for one point at a time.

Chapter 3

Linear regression

Linear regression

Chapter 4

Logistic regression

Logistic regression

Chapter 5

Decision trees

Decision trees can be used for both regression and classification task. They are simple to understand, interpret and implement.

Trees are directed acyclic graph where any two vertices are connected by one path only.

Chapter 6

Random forest

Random Forest

Chapter 7

Naive Bayes

Naive Bayes

Chapter 8

PCA

PCA

Chapter 9

Perceptron

Perceptron

Chapter 10

SVM

Chapter 11

K means

Kmeans

Chapter 12

Bibliography

Write