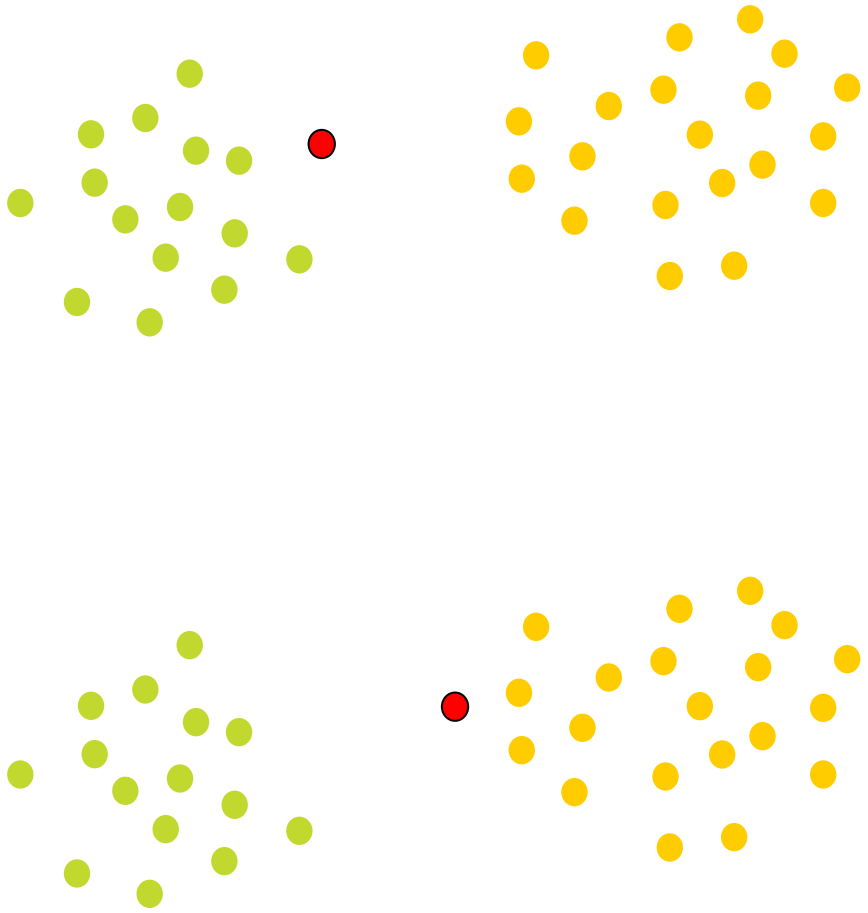
COSI 104a Introduction to machine learning

Chapter 8 – KNN

Instructor: Dr. Hongfu Liu
Email: hongfuliu@brandeis.edu

Main source: https://www-users.cs.umn.edu/~kumar001/dmbook/index.php



Nearest Neighbor Decision Rule

Given a dataset $D = \{(\vec{x}_i, y_i)\}_{i=1,\ldots,n}$

Predict the label of a new sample $\vec{x}_{new}$ as

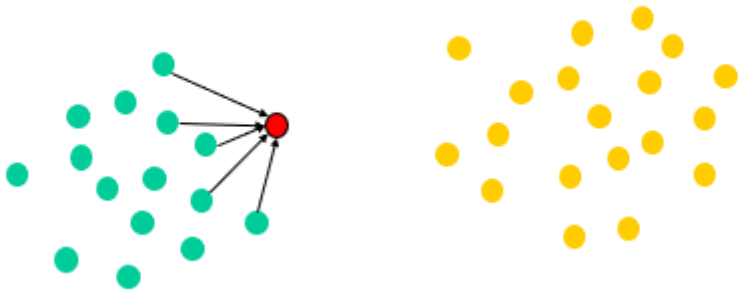
$$y_{new} = KNN_d(\vec{x}_{new}|D)$$

$d(\vec{x}_a, \vec{x}_b)$ measures the distance between two samples $\vec{x}_a$ and $\vec{x}_b$

- Find the nearest sample of $\vec{x}_{new}$ in D
- Assign $\vec{x}_{new}$ to the class of its nearest sample

K Nearest Neighbor (*K*-NN)

Assign $\vec{x}_{new}$ to a class that is the most frequently represented among its *k*-nearest samples.



$K = 5$

Distance Weighted *K*-NN

Find the *K*-nearest neighbors of $\vec{x}_{new}$: $\mathbb{N} = \{(\vec{x}_m, y_m)\}_{m=1,\dots,K} \subseteq D$

Predict the label of a new sample $\vec{x}_{new}$ as

$$c^* = \underset{c}{\operatorname{argmax}} \sum_{(x_m, y_m) \in \mathbb{N}} \delta(y_m, c)\, w(d(\vec{x}_{new}, \vec{x}_m))$$

$$\delta(y_m, c) = \begin{cases} 1, & y_m = c \\ 0, & y_m \neq c \end{cases}$$

$w\big(d(\vec{x}_{new}, \vec{x}_m)\big)$ converts the distance between $\vec{x}_{new}$ and $\vec{x}_m$ to a weight

TKNN

- Find K-nearest neighbor within a fixed range.

Time Complexity

- What is the time complexity of KNN?

1.6. Nearest Neighbors

`sklearn.neighbors` provides functionality for unsupervised and supervised neighbors-based learning methods. Unsupervised nearest neighbors is the foundation of many other learning methods, notably manifold learning and spectral clustering. Supervised neighbors-based learning comes in two flavors: classification for data with discrete labels, and regression for data with continuous labels.

The principle behind nearest neighbor methods is to find a predefined number of training samples closest in distance to the new point, and predict the label from these. The number of samples can be a user-defined constant (k-nearest neighbor learning), or vary based on the local density of points (radius-based neighbor learning). The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as *non-generalizing* machine learning methods, since they simply "remember" all of its training data (possibly transformed into a fast indexing structure such as a Ball Tree or KD Tree).

Despite its simplicity, nearest neighbors has been successful in a large number of classification and regression problems, including handwritten digits and satellite image scenes. Being a non-parametric method, it is often successful in classification situations where the decision boundary is very irregular.

The classes in `sklearn.neighbors` can handle either NumPy arrays or `scipy.sparse` matrices as input. For dense matrices, a large number of possible distance metrics are supported. For sparse matrices, arbitrary Minkowski metrics are supported for searches.

There are many learning routines which rely on nearest neighbors at their core. One example is kernel density estimation, discussed in the density estimation section.

KNeighborsClassifier

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, *, weigh
algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=
n_jobs=None)
```

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters:

n_neighbors : *int, default=5*

Number of neighbors to use by default for `kneighbors` queries.

weights : *{'uniform', 'distance'}, callable or None, default='uniform'*

Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted
- 'distance' : weight points by the inverse of their distance. in this case, close query point will have a greater influence than neighbors which are further
- [callable] : a user-defined function which accepts an array of distances, anc of the same shape containing the weights.

Refer to the example entitled Nearest Neighbors Classification showing the imp `weights` parameter on the decision boundary.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : *int, default=30*

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : *float, default=2*

Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used. This parameter is expected to be positive.

metric : *str or callable, default='minkowski'*

Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2. See the documentation of scipy.spatial.distance and the metrics listed in `distance_metrics` for valid metric values.

If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

If metric is a callable function, it takes two arrays representing 1D vectors as inputs and must return one value indicating the distance between those vectors. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

metric_params : *dict, default=None*

Additional keyword arguments for the metric function.

n_jobs : *int, default=None*

The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel backend` context. `-1` means using all processors. See Glossary for more details. Doesn't affect `fit` method.

Attributes:

classes_ : *array of shape (n_classes,)*

Class labels known to the classifier

effective_metric_ : *str or callble*

The distance metric used. It will be same as the `metric` parameter or a synonym of it, e.g. 'euclidean' if the `metric` parameter set to 'minkowski' and `p` parameter set to 2.

effective_metric_params_ : *dict*

Additional keyword arguments for the metric function. For most metrics will be same with `metric_params` parameter, but may also contain the `p` parameter value if the `effective_metric_` attribute is set to 'minkowski'.

n_features_in_ : *int*

Number of features seen during fit.

> *Added in version 0.24.*

feature_names_in_ : *ndarray of shape (`n_features_in_`,)*

Names of features seen during fit. Defined only when `X` has feature names that are all strings.

> *Added in version 1.0.*

n_samples_fit_ : *int*

Number of samples in the fitted data.

outputs_2d_ : *bool*

False when `y`'s shape is (n_samples,) or (n_samples, 1) during fit otherwise True.


?
Q
&
&
A
!