STAT 451: Introduction to Machine Learning
Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat451-fs2020/

Fall 2020

Contents

STAT 451: Introduction to Machine Learning Lecture Notes

Sebastian Raschka
Department of Statistics
University of Wisconsin–Madison

http://stat.wisc.edu/~sraschka/teaching/stat451-fs2020/

Fall 2020

2 Nearest Neighbor Methods

2.1 Introduction

Nearest neighbor algorithms are among the “simplest” supervised machine learning algorithms and have been well studied in the field of pattern recognition over the last century. While nearest neighbor algorithms are not as popular as they once were, they are still widely used in practice, and I highly recommend that you are at least considering the k-Nearest Neighbor algorithm in classification projects as a predictive performance benchmark – especially, when you are trying to develop more sophisticated models.

In this lecture, we focus mostly on two algorithms, the Nearest Neighbor (NN) algorithm and the k-Nearest Neighbor (kNN) algorithm. NN is just a special case of kNN where $k = 1$. To avoid making this text unnecessarily convoluted, we only use the abbreviation NN if we talk about concepts that do not apply to kNN in general. Otherwise, we will use kNN to refer to nearest neighbor algorithms in general, regardless of the value of k.

2.1.1 Key concepts

While kNN is a universal function approximator under certain conditions, the underlying concept is relatively simple. kNN is an algorithm for supervised learning that simply *stores* the labeled training examples,

$$\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D} \quad (|\mathcal{D}| = n), \tag{1}$$

during the training phase. For this reason, kNN is also called a **lazy** learning algorithm.

What it means to be a *lazy* learning algorithm is that the processing of the training examples is postponed until making predictions [1] – again, the training consists literally of just storing the training data.

[1] When you are reading recent literature, note that the *prediction* step is now often called ”inference” in the machine learning community. Note that in the context of machine learning, the term ”inference” is not to be confused with how we use the term ”inference” in statistics, though.

Then, to make a prediction (class label or continuous target), a trained *kNN* model finds the k nearest neighbors of a query point and computes the class label (classification) or continuous target (regression) based on the k nearest (most "similar") points. The exact mechanics will be explained in the next sections. However, the overall idea is that instead of approximating the target function $f(\mathbf{x}) = y$ globally, during each prediction, kNN approximates the target function locally. In practice, it is easier to learn to approximate a function locally than globally.

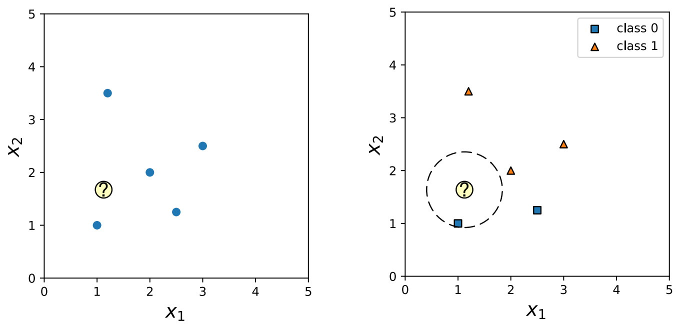

Figure 1: Illustration of the nearest neighbor classification algorithm in two dimensions (features x_1 and x_2). In the left subpanel, the training examples are shown as blue dots, and a query point that we want to classify is shown as a question mark. In the right subpanel, the class labels are annotated via blue squares and red triangle symbols. The dashed line indicates the nearest neighbor of the query point, assuming a Euclidean distance metric. The predicted class label is the class label of the closest data point in the training set (here: class 0, i.e., blue square).

2.1.2 Nearest Neighbor Classification In Context

The previous lecture notes document introduced different kinds of categorization schemes, which may be helpful for understanding and distinguishing different types of machine learning algorithms.

To recap, the categories we discussed were

- eager vs lazy;
- batch vs online;
- parametric vs nonparametric;
- discriminative vs generative.

Since kNN does not have an explicit training step and defers all of the computation until prediction, we already determined that kNN is a *lazy* algorithm.

Further, instead of devising one global model or approximation of the target function, for each different data point, there is a different local approximation, which depends on the data point itself as well as the training data points. Since the prediction is based on a comparison of a query point with data points in the training set (rather than a global model), kNN is also categorized as **instance-based** (or "memory-based") method. While kNN is a lazy instance-based learning algorithm, an example of an eager instance-based learning algorithm would be the support vector machine (which is not covered in this course due to time constraints)

Lastly, because we do not make any assumption about the functional form of the kNN algorithm, a kNN model is also considered a **nonparametric** model. However, categorizing

kNN as either discriminative or generative is not as straightforward as for other algorithms. Under certain assumptions, we can estimate the conditional probability that a given data point belongs to a given class as well as the marginal probability for a feature given a training dataset (more details are provided in the section on "kNN from a Bayesian Perspective" later). However, since kNN does not explicitly try to model the data generating process but models the posterior probabilities, $p(f(\mathbf{x}) = i|\mathbf{x})$, directly, kNN is usually considered a discriminative model.

2.1.3 Common Use Cases of kNN

While neural networks are gaining popularity in the computer vision and pattern recognition field, one area where k-nearest neighbors models are still commonly and successfully being used is in the intersection between computer vision, pattern classification, and biometrics (e.g., to make predictions based on extracted geometrical features[2]).

Other common use cases include recommender systems (via collaborative filtering[3]) and outlier detection[4].

2.2 Nearest Neighbor Algorithm

After introducing the overall concept of the nearest neighbor algorithms, this section provides a more formal or technical description of the 1-nearest neighbor (NN) algorithm.

Training algorithm:

```
for $i = 1, ..., n$ in the $n$-dimensional training dataset $\mathcal{D}$ ($|\mathcal{D}| = n$):
    • store training example $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle$
```

Prediction algorithm [5]:

```
closest_point := None
closest_distance := $\infty$
    • for $i = 1, ..., n$:
        – current_distance := $d(\mathbf{x}^{[i]}, \mathbf{x}^{[q]})$
        – if current_distance < closest_distance:
            ∗ closest_distance := current_distance
            ∗ target_value_of_closest_point := $y^{[i]}$
```

The prediction produced by the 1NN model,$h(\mathbf{x}^{[q]})$, is the target value of the closest point.

Unless noted otherwise, the default distance metric (in the context of this lecture) of nearest neighbor algorithms is the Euclidean distance (also called L^2 distance), which computes the

[2]Asmaa Sabet Anwar, Kareem Kamal A Ghany, and Hesham Elmahdy. "Human ear recognition using geometrical features extraction". In: *Procedia Computer Science* 65 (2015), pp. 529–537.

[3]Youngki Park et al. "Reversed CF: A fast collaborative filtering algorithm using a k-nearest neighbor graph". In: *Expert Systems with Applications* 42.8 (2015), pp. 4022–4028.

[4]Guilherme O Campos et al. "On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study". In: *Data Mining and Knowledge Discovery* 30.4 (2016), pp. 891–927.

[5]We use ":=" as an assignment operator.

distance between two points, $\mathbf{x}^{[a]}$ and $\mathbf{x}^{[b]}$:

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sqrt{\sum_{j=1}^{m} \left(x_j^{[a]} - x_j^{[b]} \right)^2}. \tag{2}$$

2.3 Nearest Neighbor Decision Boundary

In this section, we build some intuition for the decision boundary of the NN classification model. Assuming a Euclidean distance metric, the decision boundary between any two training examples a and b is a straight line. If a query point is located on the decision boundary, this means its equidistant from both training example a and b.

While the decision boundary between a pair of points is a straight line, the decision boundary of the NN model on a global level, considering the whole training set, is a set of connected, convex polyhedra. All points within a polyhedron are closest to the training example inside, and all points outside the polyhedron are closer to a different training example.

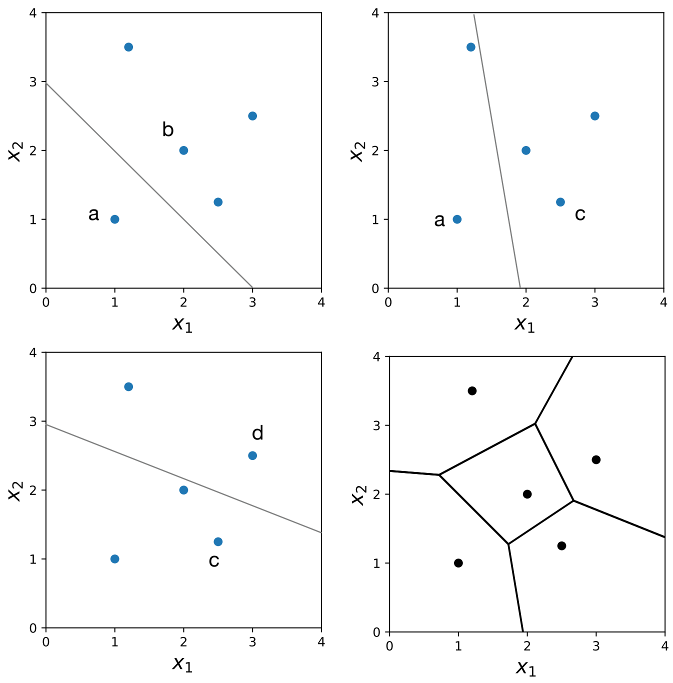

Figure 2: Illustration of the plane-partitioning of a two-dimensional dataset (features x_1 and x_2) via linear segments between two training examples (a & b, a & c, and c & d) and the resulting Voronoi diagram (lower right corner).

This partitioning of regions on a plane in 2D is also called "Voronoi diagram" or "Voronoi tessellation." (You may remember from geometry classes that given a discrete set of points, a Voronoi diagram can also be obtained by a process known as Delaunay triangulation[6], i.e., by connecting the centers of the circumcircles.)

While each linear segment is equidistant from two different training examples, a vertex (or node) in the Voronoi diagram is equidistant to three training examples. Then, to draw the decision boundary of a two-dimensional nearest neighbor classifier, we take the union of the pair-wise decision boundaries of instances of the same class.

[6]**https://en.wikipedia.org/wiki/Delaunay˙triangulation**.

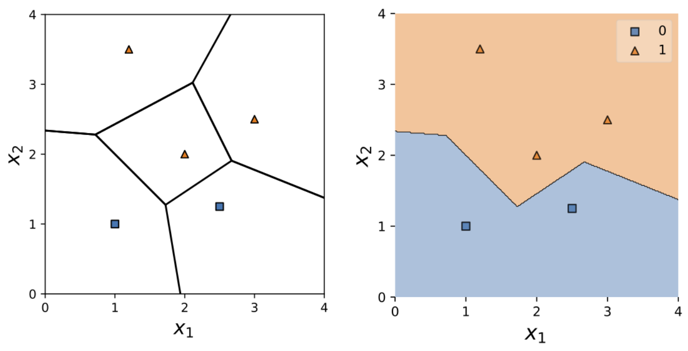

Figure 3: Illustration of the nearest neighbor decision boundary as the union of the polyhedra of training examples belonging to the same class.

2.4 k-Nearest Neighbor Classification and Regression

Previously, we described the NN algorithm, which makes a prediction by assigning the class label or continuous target value of the most similar training example to the query point (where similarity is typically measured using the Euclidean distance metric for continuous features).

Instead of basing the prediction of the single, most similar training example, kNN considers the k nearest neighbors when predicting a class label (in classification) or a continuous target value (in regression).

2.4.1 Classification

In the classification setting, the common kNN model is to predicts the target class label as the class label that is most often represented among the k most similar training examples for a given query point. In other words, the class label can be considered as the “mode” of the k training labels or the outcome of a “plurality voting.” Note that in literature, kNN classification is often described as a “majority voting.” While the authors usually mean the right thing, the term “majority voting” usually refers to a reference value of >50% for making a decision[7]. In the case of binary predictions (classification problems with two classes), there is always a majority or a tie. Hence, a majority vote is also automatically a plurality vote. However, in multi-class settings, we do not require a majority to make a prediction via kNN. For example, in a three-class setting a frequency $> \frac{1}{3}$ (approx 33.3%) could already enough to assign a class label.

[7]Interestingly, the term “plurality vote” (in North America) is called ”relative majority” in the United Kingdom, according to https://en.wikipedia.org/wiki/Plurality_(voting)

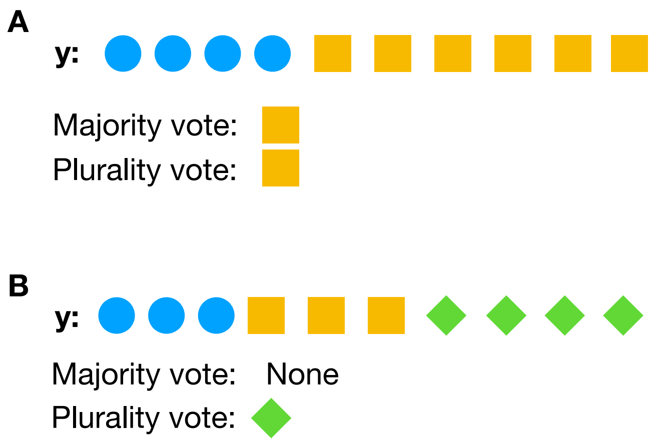

Figure 4: Illustration of plurality and majority voting.

Remember that the NN prediction rule (recall that we defined NN as the special case of kNN with $k = 1$) is the same for both classification or regression. However, in kNN we have two distinct prediction algorithms:

- Plurality voting among the k nearest neighbors for classification.
- Averaging the continuous target variables of the k nearest neighbors for regression.

More formally, assume we have a target function $f(\mathbf{x}) = y$ that assigns a class label $y \in \{1, \dots, t\}$ to a training example,

$$f : \mathbb{R}^m \rightarrow \{1, ..., t\}. \tag{3}$$

(Usually, we use the letter k to denote the number of classes in this course, but in the context of kNN, it would be too confusing.)

Assuming we identified the k nearest neighbors ($\mathcal{D}_k \subseteq \mathcal{D}$) of a query point $\mathbf{x}^{[q]}$,

$$\mathcal{D}_k = \{\langle \mathbf{x}^{[1]}, f\big(\mathbf{x}^{[1]}\big)\rangle, \dots, \langle \mathbf{x}^{[k]}, f\big(\mathbf{x}^{[k]}\big)\rangle\}, \tag{4}$$

we can define the kNN *hypothesis* as

$$h(\mathbf{x}^{[q]}) = \arg \max_{y \in \{1,\dots,t\}} \sum_{i=1}^{k} \delta(y, f(\mathbf{x}^{[i]})). \tag{5}$$

Here, δ denotes the Kronecker Delta function

$$\delta(a, b) = \begin{cases} 1, & \text{if } a = b, \\ 0, & \text{if } a \neq b. \end{cases} \tag{6}$$

Or, in simpler notation, if you remember the "mode" from introductory statistics classes:

$$h(\mathbf{x}^{[t]}) = \text{mode}\big(\{f\big(\mathbf{x}^{[1]}\big), \dots, f\big(\mathbf{x}^{[k]}\big)\}\big). \tag{7}$$

A common distance metric to identify the k nearest neighbors $\mathcal{D}_k$ is the Euclidean distance measure,

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sqrt{\sum_{j=1}^{m} \left(x_j^{[a]} - x_j^{[b]}\right)^2}, \tag{8}$$

which is a pairwise distance metric that computes the distance between two data points $\mathbf{x}^{[a]}$ and $\mathbf{x}^{[b]}$ over the m input features.

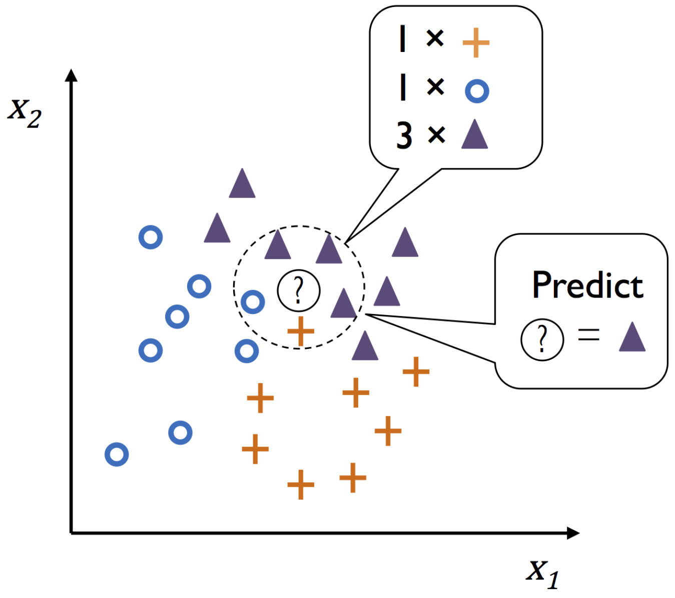

Figure 5: Illustration of kNN for a 3-class problem with $k = 5$.

2.4.2 Regression

The general concept of kNN for regression is the same as for classification: first, we find the k nearest neighbors in the dataset; second, we make a prediction based on the labels of the k nearest neighbors. However, in regression, the target function is a real- instead of discrete-valued function,

$$f : \mathbb{R}^m \rightarrow \mathbb{R}. \tag{9}$$

A common approach for computing the continuous target is to compute the mean or average target value over the k nearest neighbors,

$$h\big(\mathbf{x}^{[t]}\big) = \frac{1}{k} \sum_{i=1}^{k} f\big(\mathbf{x}^{[i]}\big). \tag{10}$$

As an alternative to averaging the target values of the k nearest neighbors to predict the label of a query point, it is also not uncommon to use the median instead.

2.5 Curse of Dimensionality

The kNN algorithm is particularly susceptible to the *curse of dimensionality*[8]. In machine learning, the curse of dimensionality refers to scenarios with a fixed number of training examples but an increasing number of dimensions and range of feature values in each dimension in a high-dimensional feature space.

In kNN an increasing number of dimensions becomes increasingly problematic because the more dimensions we add, the larger the volume in the hyperspace needs to be to capture a

[8]David L Donoho et al. "High-dimensional data analysis: The curses and blessings of dimensionality". In: *AMS math challenges lecture* 1.2000 (2000), p. 32.

fixed number of neighbors. As the volume grows larger and larger, the "neighbors" become less and less "similar" to the query point as they are now all relatively distant from the query point considering all different dimensions that are included when computing the pairwise distances.

For example, consider a single dimension with unit length (range [0, 1]). Now, if we consider 100 training examples that are uniformly distributed, we expect one training example located at each 0.01th unit along the [0, 1] interval or axis. So, to consider the three nearest neighbors of a query point, we expect to cover 3/100 of the feature axis. However, if we add a second dimension, the expected interval length that is required to include the same amount of data (3 neighbors) now increases to $0.03^{1/2}$ (we now have a unit rectangle). In other words, instead of requiring $0.03 \times 100\% = 3\%$ of the space to include 3 neighbors in 1D, we now need to consider $0.03^{1/2} \times 100\% = 17.3\%$ of a 2D space to cover the same amount of data points – the density decreases with the number of dimensions. In 10 dimensions, that's now $0.03^{1/10} = 70.4\%$ of the hypervolume we need to consider to include three neighbors on average. You can see that in high dimensions we need to take a large portion of the hypervolume into consideration (assuming a fixed number of training examples) to find k nearest neighbors, and then these so-called "neighbors" may not be particularly "close" to the query point anymore.

2.6 Computational Complexity and the Big-O Notation

The Big-O notation is used in both mathematics and computer science to study the asymptotic behavior of functions, i.e., the asymptotic upper bounds. In the context of algorithms in computer science, the Big-O notation is most commonly used to measure the time complexity or runtime of an algorithm for the worst case scenario. (Often, it is also used to measure memory requirements.)

Since Big-O notation and complexity theory, in general, are areas of research in computer science, we will not go into too much detail in this course. However, you should at least be familiar with the basic concepts, since it is an essential component for the study of machine learning algorithms.

Notation	**Name**
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^c)$	Higher-level polynomial
$O(2^n)$	Exponential

Table 1: Most common Big-O runtime notations listed in increasing order (slowest-growing functions first).

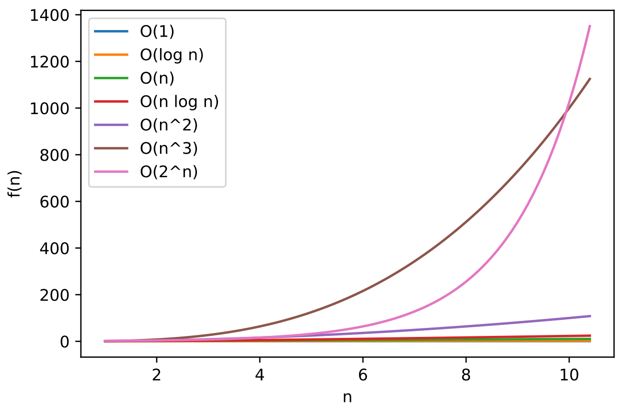

Figure 6: An illustration of the growth rates of common functions.

Note that in "Big-O" analysis, we only consider the most dominant term, as the other terms and constants become insignificant asymptotically. For example, consider the function

$$f(x) = 14x^2 - 10x + 25. \tag{11}$$

The worst case complexity of this function is $O(x^2)$, since x^2 is the dominant term.

Next, consider the example

$$f(x) = (2x + 8)\log_2(x + 9). \tag{12}$$

In "Big-O" notation, that is $O(x \log x)$. Note that it does not need to distinguish between different bases of the logarithms, e.g., $\log_{10}$, or $\log_2$, since we can regard these just as a scalar factor given the conversion

$$\log_2(x) = \log_{10}(x)/\log_{10}(2), \tag{13}$$

where $\frac{1}{log_{10}(2)}$ is just a scaling factor.

Lastly, consider this naive example of implementing matrix multiplication in Python:

```
A = [[1, 2, 3],
     [2, 3, 4]]

B = [[5, 8],
     [6, 9],
     [7, 10]]

def matrixmultiply (A, B):

    C = [[0 for row in range(len(A))]
          for col in range(len(B[0]))]

    for row_a in range(len(A)):
        for col_b in range(len(B[0])):
            for col_a in range(len(A[0])):
                C[row_a][col_b] += \
                    A[row_a][col_a] * B[col_a][col_b]
    return C

matrixmultiply(A, B)
```

Result:

```
[[38, 56],
 [56, 83]]
```

Due to the three nested *for*-loops, the runtime complexity of this function is $O(n^3)$.

2.6.1 Big-O of kNN

For the brute-force neighbor search of the kNN algorithm, we have a time complexity of $O(n \times m)$, where n is the number of training examples and m is the number of dimensions in the training set. For simplicity, assuming $n \gg m$, the complexity of the brute-force nearest neighbor search is $O(n)$. In the next section, we will briefly go over a few strategies to improve the runtime of the kNN model.

2.7 Improving Computational Performance

2.7.1 Naive kNN Algorithm in Pseudocode

Below are two naive approaches (Variant A and Variant B) for finding the k nearest neighbors of a query point $\mathbf{x}^{[q]}$.

Variant A

```
$\mathcal{D}_k := \{\}$
while $|\mathcal{D}_k| <$ k:
  • closest_distance $:= \infty$
  • for $i = 1, ..., n, \quad \forall i \notin \mathcal{D}_k$:
      – current_distance $:= d(\mathbf{x}^{[i]}, \mathbf{x}^{[q]})$
      – if current_distance < closest_distance:
          ∗ closest_distance := current_distance
          ∗ closest_point $:= \mathbf{x}^{[i]}$
  • add closest_point to $\mathcal{D}_k$
```

Variant B

```
$\mathcal{D}_k := \mathcal{D}$
while $|\mathcal{D}_k| > k$:
  • largest_distance $:= 0$
  • for $i = 1, ..., n \quad \forall i \in \mathcal{D}_k$:
      – current_distance $:= d(\mathbf{x}^{[i]}, \mathbf{x}^{[q]})$
      – if current_distance > largest_distance:
          ∗ largest_distance := current_distance
          ∗ farthest_point $:= \mathbf{x}^{[i]}$
```

- remove `farthest_point` from $\mathcal{D}_k$

Using a Priority Queue

Both Variant A and Variant B are expensive algorithms, $O(k \times n)$ and $O((n-k) \times n)$, respectively . However, with a simple trick, we can improve the nearest neighbor search to $O(k \log(n))$. For instance, we could implement a priority queue using a heap data structure [9].

We initialize the heap with the k arbitrary points from the training dataset based on their distances to the query point. Then, as we iterate through the dataset to find the first nearest neighbor of the query point, at each step, we make a comparison with the points and distances in the heap. If the point with the largest stored distance in the heap is farther away from the query point that the current point under consideration, we remove the farthest point from the heap and insert the current point. Once we finished one iteration over the training dataset, we now have a set of the k nearest neighbors.

2.7.2 Data Structures

Different data structures have been developed to improve the computational performance of kNN during prediction. In particular, the idea is to be smarter about identifying the k nearest neighbors. Instead of comparing each training example in the training set to a given query point, approaches have been developed to partition the search space most efficiently.

The details of these data structures are beyond the scope of this lecture since they require some background in computer science and data structures, but interested students are encouraged to read the literature referenced in this section.

Bucketing

The simplest approach is "bucketing"[10]. Here, we divide the search space into identical, similarly-sized cells (or buckets), that resemble a grid (picture a 2D grid 2-dimensional hyperspace or plane).

KD-Tree

A KD-Tree[11], which stands for k-dimensional search tree, is a generalization of binary search trees. KD-Trees data structures have a time complexity of $O(\log(n))$ on average (but $O(n)$ in the worst case) or better and work well in relatively low dimensions. KD-Trees also partition the search space perpendicular to the feature axes in a Cartesian coordinate system. However, with a large number of features, KD-Trees become increasingly inefficient, and alternative data structures, such as Ball-Trees, should be considered.[12]

Ball-Tree

In contrast to the KD-Tree approach, the Ball-Tree[13] partitioning algorithms are based on the construction of hyperspheres instead of cubes. While Ball-Tree algorithms are generally

[9]A heap is a special case of a binary search tree with a structure that makes lookups more efficient. You are not expected to now how heaps work in the exam, but you are encouraged to learn more about this data structure. A good overview is provided on Wikipedia with links to primary sources: https://en.wikipedia.org/wiki/Heap_%28data_structure%29

[10]Ronald L Rivest. "On the Optimality of Elia's Algorithm for Performing Best-Match Searches." In: *IFIP Congress*. 1974, pp. 678–681.

[11]Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[12]Note that software implementations such as the ighborsClassifier in the Scikit-learn library has a "method='auto'" default setting that chooses the most appropriate data structure automatically.

[13]Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

more expensive to run than KD-Trees, the algorithms address some of the shortcomings of the KD-Tree datastructure and are more efficient in higher dimensions.

Note that these data structures or space partitioning algorithms come each with their own set of hyperparameters (e.g., the leaf size, or settings related to the leaf size). Detailed discussions of the different data structures for efficient data structures are beyond the scope of this class.

2.7.3 Dimensionality Reduction

Next, to help reduce the effect of the curse of dimensionality, dimensionality reduction strategies are also useful for speeding up the nearest neighbor search by making the computation of the pair-wise distances "cheaper." There are two approaches to dimensionality reduction:

- Feature Selection (e.g., Sequential Forward Selection)
- Feature Extraction (e.g., Principal Component Analysis)

We will cover both feature selection and feature extraction as separate topics later in this course.

2.7.4 Faster Distance Metric/Heuristic

kNN is compatible with any pairwise distance metric. However, the choice of the distance metric affects the runtime performance of the algorithm. For instance, computing the Mahalanobis distance is much more expensive than calculating the more straightforward Euclidean distance.

2.7.5 "Pruning"

There are different kinds of "pruning" approaches that we could use to speed up the kNN algorithm. For example, *editing* and *prototype selection.*

Editing

In *edited* kNN, we permanently remove data points that do not affect the decision boundary. For example, consider a single data point (aka "outlier") surrounded by many data points from a different class. If we perform a kNN prediction, this single data point will not influence the class label prediction in plurality voting; hence, we can safely remove it.

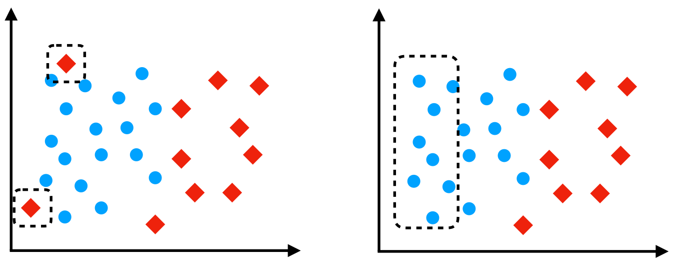

Figure 7: Illustration of kNN editing, where we can remove points from the training set that do not influence the predictions. For example, consider a 3NN model. On the left, the two points enclosed in dashed lines would not affect the decision boundary as "outliers." Similarly, points of the "right" class that are very far away from the decision boundary, as shown in the right subpanel, do not influence the decision boundary and hence could be removed for efficiency concerning data storage or the number of distance computations.

Prototypes

Another strategy (somewhat related to KMeans, a clustering algorithm that we will cover towards the end of this course), is to replace selected data points by prototypes that summarize multiple data points in dense regions.

2.7.6 Parallelizing kNN

kNN is one of these algorithms that are very easy to *parallelize*. There are many different ways to do that. For instance, we could use distributed approaches like map-reduce and place subsets of the training datasets on different machines for the distance computations. Further, the distance computations themselves can be carried out using parallel computations on multiple processors via CPUs or GPUs[14].

2.8 Distance measures

There are many distance metrics or measures we can use to select k nearest neighbors. There is no "best" distance measure, and the choice is highly context- or problem-dependent.

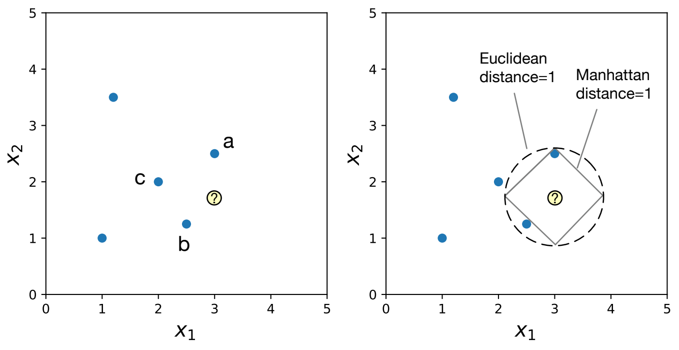

Figure 8: The phrase "nearest" is ambiguous and depends on the distance metric we use.

For continuous features, the probably most common distance metric is the Euclidean dis-

[14]For example, the RapidsAI ecosystem provided implementations of machine learning algorithms that are parallelized and executed faster on GPUs; https://medium.com/rapids-ai/accelerating-k-nearest-neighbors-600x-using-rapids-cuml-82725d56401e

tance. Another popular choice is the Manhattan distance,

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sum_{j=1}^{m} \left| x^{[a]} - x^{[b]} \right|, \tag{14}$$

which emphasizes differences between "distant" feature vectors or outliers less than the Euclidean distance.

A generalization of the Euclidean or Manhattan distance is the so-called Minkowski distance,

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \left[\sum_{j=1}^{m} \left(\left| x^{[a]} - x^{[b]} \right| \right)^{p} \right]^{\frac{1}{p}}, \tag{15}$$

which is equal to the Euclidean distance if $p = 2$ and equal to the Manhattan distanced if $p = 1$.

The Mahalanobis distance would be another good choice for a distance metric as it considers the variance of the different features as well as the covariance among them. However, one downside of using more "sophisticated" distance metrics is that it also typically negatively impacts computational efficiency. For instance, the Mahalanobis distance is substantially more challenging to implement efficiently, for example, if we consider running kNN in a distributed-fashion, as it requires the covariances as a scaling term.

2.8.1 Discrete Features

Minkowski-based distance metrics can be used for discrete and continuous features. One example would be the so-called Hamming distance, which really is just the Manhattan distance applied to binary feature vectors:

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sum_{j=1}^{m} \left| x^{[a]} - x^{[b]} \right|. \tag{16}$$

The Hamming distance is also called overlap metric, because it essentially measures how many positions in two vectors are different. For instance, considering two vectors

$$\mathbf{a} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{17}$$

The Hamming distance is one, because the vectors only differ in one position.

As we discussed in class, If we are working with vectors containing word counts of documents and we want to measure the similarity (or distance) between two documents, cosine similarity could be a metric that is more appropriate than, for example, the Euclidean distance. The cosine similarity[15] is defined as the the dot-product between two vectors normalized by their magnitude:

$$\cos(\theta) = \frac{\mathbf{a}^{\top}\mathbf{b}}{||\mathbf{a}|| \cdot ||\mathbf{b}||} \tag{18}$$

To provide some intuition for why this measure could be more indicative of the similarity of two document vectors, consider a document $\mathbf{a}$ in which we duplicated every sentence $\mathbf{a}'$.

[15]Please note that while cosine similarity can be useful to measure distances in *k*NN, it is not a proper distance metric as it violates the triangle-inequality, $d(\mathbf{a}, \mathbf{c}) \leq d(\mathbf{a}, \mathbf{b}) + d(\mathbf{b}, \mathbf{c})$.

Then, the cosine similarity to another document vector **b** would be the same for **a** and **a**′, which is not true for, for example, the dot product.

While document-word counts were used as an illustrative example above, please do not worry about the details at this point as we will discover text analysis in a separate lecture in this course – if time permits.

Also, an important consideration when we are talking about discrete or categorical features is whether the features "categories" are on an ordinal or nominal scale. We will discuss this in detail in a feature lecture on "Data Preprocessing."

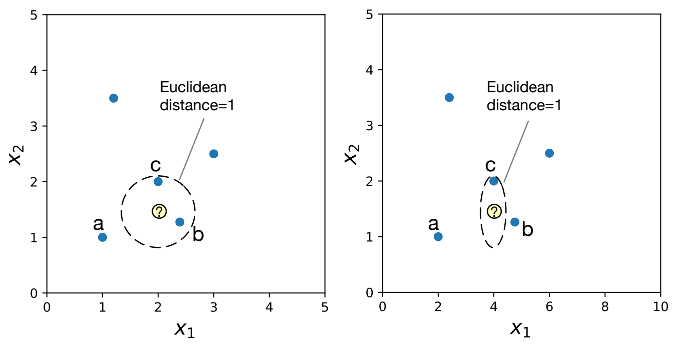

Figure 9: Next, to choosing an appropriate distance metric, feature scaling is another important consideration, which is illustrated in this figure. The right subpanel illustrates the effect of scaling the x_1 axis by a factor of 2 on finding the nearest neighbor via Euclidean distance.

2.8.2 Feature Weighting

Furthermore, we can modify distances metrics by adding a weight to each feature dimension, which is equivalent to feature scaling. In the case of the Euclidean distance, this would look as follows:

$$d_w(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sqrt{\sum_{j=1}^{m} w_j \left(x_j^{[a]} - x_j^{[b]}\right)^2}, \tag{19}$$

where $w_j \in \mathbb{R}$.

To implement this efficiently in code, we can express the weighting as a transformation matrix, where the transformation matrix is a diagonal matrix consisting of the m weight coefficients for the m features:

$$\mathbf{W} \in \mathbb{R}^{m \times m} = \text{diag}(w_1, w_2, ..., w_m). \tag{20}$$

In particular, note that we can express the standard Euclidean distance as a dot product of the distance vector x:

$$d(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sqrt{(\mathbf{x}^{[a]} - \mathbf{x}^{[b]})^T (\mathbf{x}^{[a]} - \mathbf{x}^{[b]})}. \tag{21}$$

Then, the distance-weighted Euclidean distance can be expressed as a follows:

$$d_w(\mathbf{x}^{[a]}, \mathbf{x}^{[b]}) = \sqrt{(\mathbf{x}^{[a]} - \mathbf{x}^{[b]})^T \mathbf{W} (\mathbf{x}^{[a]} - \mathbf{x}^{[b]})}. \tag{22}$$

2.9 Distance-weighted *k*NN

A variant of kNN is distance-weighted kNN. In "regular" kNN, the all k neighbors participate similarly in the plurality voting or averaging. However, especially if the radius enclosing a set of neighbors is large, we may want to give a stronger weights to neighbors that are "closer" to the query point. For instance, we can assign a weight w to the neighbors in kNN classification,

$$h(\mathbf{x}^{[t]}) = \arg \max_{j \in \{1, \dots, p\}} \sum_{i=1}^{k} w^{[i]} \delta(j, f(\mathbf{x}^{[i]})). \tag{23}$$

Simiarly, we can define the following equation for kNN regression:

$$h(\mathbf{x}^{[t]}) = \frac{\sum_{i=1}^{k} w^{[i]} f(\mathbf{x}^{[i]})}{\sum_{i=1}^{k} w^{[i]}}. \tag{24}$$

As described by Tom Mitchell[16], a popular weighting scheme is using the inverse squared distance

$$w^{[i]} = \frac{1}{d(\mathbf{x}^{[i]}, \mathbf{x}^{[t]})^2}, \tag{25}$$

where $h(\mathbf{x}) = f(\mathbf{x})$ is used for an exact match. Other strategies include adding a small constant to the denominator for avoiding zero-division errors.

Also, using a weighting scheme as described above, we can also turn the kNN algorithm into a *global* method by considering all data points instead of k, as defined by Shepard[17].

2.10 Improving Predictive Performance

There are several different ways of how we can improve the predictive performance of the kNN algorithms:

- Choosing the value of k.
- Scaling of the feature axes.
- Choice of distance measure.
- Weighting of the distance measure.

However, other techniques such as "editing" (removing noisy data points) and so forth also affect the generalization performance (i.e., the predictive performance on unseen, that is, non-training data) of kNN.

Choosing between different settings of an algorithm is also known as "hyperparameter tuning" or "model selection," which is typically performed by cross-validation.

[16]T.M. Mitchell. "Machine Learning". In: McGraw-Hill International Editions. McGraw-Hill, 1997. Chap. 8. ISBN: 9780071154673. URL: https://books.google.com/books?id=EoYBngEACAAJ.

[17]Donald Shepard. "A two-dimensional interpolation function for irregularly-spaced data". In: *Proceedings of the 1968 23rd ACM national conference.* ACM. 1968, pp. 517–524.

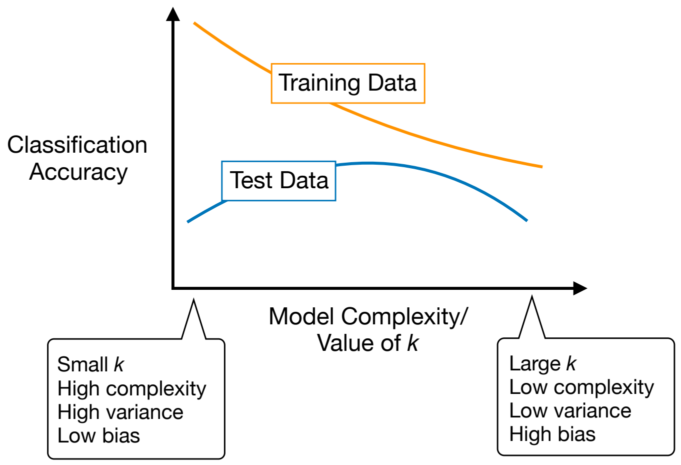

Figure 10: By changing the value of k, we affect the complexity of a kNN model. In practice, we try to find a good trade-off between high bias (the model is not complex enough to fit the data well) and high variance (the model fits the training data too closely). We will discuss overfitting and the bias-variance trade-off in more detail in future lectures.

In particular, model selection helps us with reducing the effect of the curse of dimensionality and overfitting to the training data, if done properly. We will discuss model selection and cross-validation later in this course.

In practice, values of k between 3-15 seem reasonable choices. Also, if we are working on binary classification problems, it is a good idea to avoid even numbers for k to prevent ties.

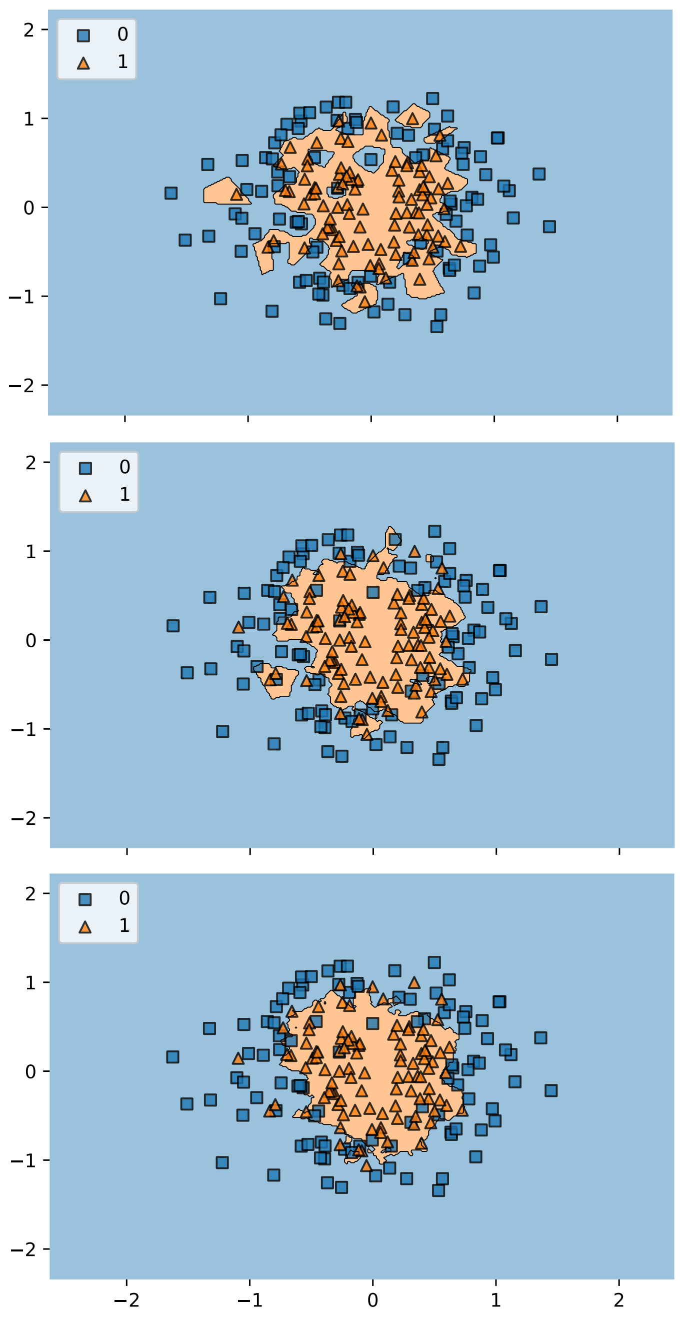

Figure 11: An illustration of the effect of changing the value of k on a simple toy dataset.

2.11 Error Bounds

Cover and Hart have proved that the 1NN algorithm is guaranteed to be no worse than twice the Bayes error[18]. The Bayes error is the minimum possible error that can be achieved; we will discuss the Bayes error in future lectures. We will not cover the proof of the error bounds in class, but interested students are encouraged to read more in Chapter 4 (Nonparametric

[18]Thomas Cover and Peter Hart. "Nearest neighbor pattern classification". In: *IEEE transactions on information theory* 13.1 (1967), pp. 21–27.

Techniques) of Duda, Hart, and Stork's *Pattern Classification* book[19].

2.12 *k*NN from a Bayesian Perspective

As a statistics student, you may be interested in looking at kNN from a Bayesian perspective. Duda, Hard, and Stork provide an excellent theoretical foundation together with helpful illustrations in their book "Pattern Classification[20]" (Chapter 4, Section 4.4) – this is out of the scope of this class but recommended if you are interested. If you are not familiar with Bayes theorem, yet (no worries if you are not, we will cover it later in this course), do not worry about it, and please feel free to skip this section.

In my attempt below, I try to describe the kNN concept from a Bayesian perspective using our familiar notation.

First, recall Bayes' theorem

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}. \tag{26}$$

In our context, that is

$$\text{Posterior Prob.} = \frac{\text{Likelihood} \times \text{Prior Prob.}}{\text{Marginal Prob.}} \tag{27}$$

or

$$P(y=i|\mathbf{x}) = \frac{p(\mathbf{x}|y=i) \times p(y=i)}{p(\mathbf{x})}. \tag{28}$$

Suppose we have a dataset $\mathcal{D}$ with n data points and t classes. $\mathcal{D}_i$ denotes the subset of data points with class label $y = i, \quad i \in \{1, ..., t\}$.

Now, to classify a new data point $\mathbf{x}$, we draw a tight sphere around the k nearest neighbors of $\mathbf{x}$. This sphere has the volume V.

Then

$$p(\mathbf{x}|y=i) = \frac{k_i}{|D_i| \times V} \tag{29}$$

is a density estimate of class i, where k_i denote the k neighbors with class label i. The marginal probability, $p(\mathbf{x})$ can then be computed as

$$p(\mathbf{x}) = \frac{k}{|\mathcal{D}| \times V}. \tag{30}$$

The class priors are then computed as

$$p(y=i) = \frac{|\mathcal{D}_i|}{|\mathcal{D}|}. \tag{31}$$

Combining the equations, we get the posterior probability that the data point $\mathbf{x}$ belongs to class i:

$$P(y=i|\mathbf{x}) = \frac{p(\mathbf{x}|y=i) \times p(y=i)}{p(\mathbf{x})} = \frac{k_i}{k}. \tag{32}$$

Stable approximation under the assumption that

$$|\mathcal{D}|, k \rightarrow \infty \tag{33}$$

and

$$\frac{k}{|\mathcal{D}|} \rightarrow 0. \tag{34}$$

[19]Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.
[20]Duda, Hart, and Stork, *Pattern classification*.

2.13 Advantages and Disadvantages of Using kNN

One of the most significant advantages of kNN is that it is relatively easy to implement and interpret. Also, with its approach to approximate complex global functions locally, it can be a powerful predictive model.

The downsides are that kNN is very sensitive to the curse of dimensionality and expensive to compute with a $O(n)$ prediction step – however, smart implementations and use of data structures such as KD-trees and Ball-trees can make kNN substantially more efficient.

Compared to other machine learning algorithms, the basic kNN algorithm has relatively few hyperparameters, namely k and the distance metric; however, the choice of an appropriate distance metric is not always obvious. Additional hyperparameters are added if we consider rescaling the feature axes and weighting neighbors by their distance from the query point, for example.

2.14 Other Forms of Instance-based Learning

2.14.1 Locally Weighted Regression

Another popular example of instance-based learning is *locally weighted regression.* The idea behind locally weighted regression is straightforward. Similar to kNN, training examples neighboring a query point are selected. Then, a regression model is fit to that selected set of training examples to predict the value of a given data point. Essentially, we are approximating the target function locally, which is easier than learning a hypothesis that approximates the target function well on a global scale.

The regression model could have any form, really; however, linear regression models are popular in the context of locally weighted regression because it is simple, computationally efficient, and performs sufficiently well for approximating the local neighborhood of a given data point.

Interestingly, the concept behind locally weighted regression has been recently used to approximate decision functions locally to help interpret decisions made by "complex" models (random forests, multi-layer networks, etc.), for example, via the LIME technique (LIME is an acronym for Local Interpretable Model-Agnostic Explanations)[21].

2.14.2 Kernel Methods

The term kernel may be a bit confusing due to its varied use, but in the context of machine learning, kernel methods are usually associated with what is known as the "kernel trick." The probably most widely used kernel method is the support vector machine (its standard form can be interpreted as a special case using a linear kernel). We will discuss this in more detail later in this course if time permits. In a nutshell, kernel methods use a kernel (as mentioned earlier, a "similarity function") to measure the similarity or distance between pairs of data points, and the "trick" refers to an implicit transformation into a higher dimensional space where a linear classifier can separate the data points. However, while SVM can be considered an instance-based learner, it is not a "lazy" learner such as kNN.

[21] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why should i trust you?: Explaining the predictions of any classifier". In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining.* ACM. 2016, pp. 1135–1144.