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.

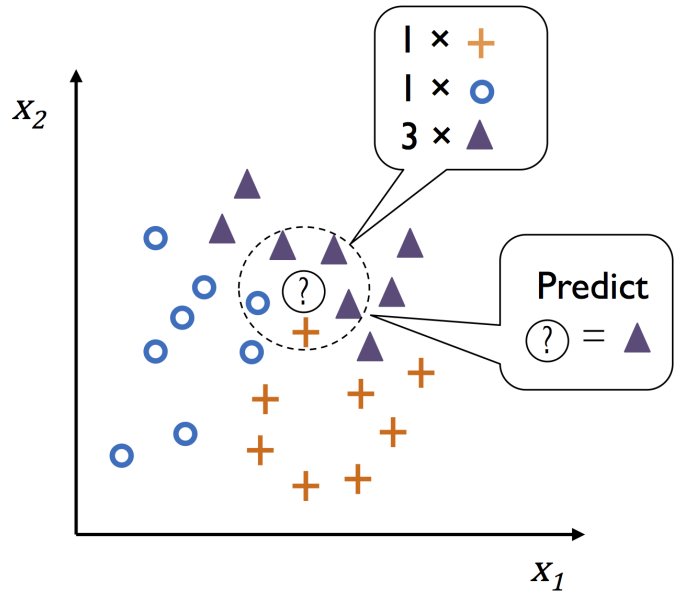

Figure 5: Illustration of kNN for a 3-class problem with k=5.

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,

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

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

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

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*[7]. In machine learning, the curse of dimensionality refers to scenarios with a fixed size 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

[7] 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.

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

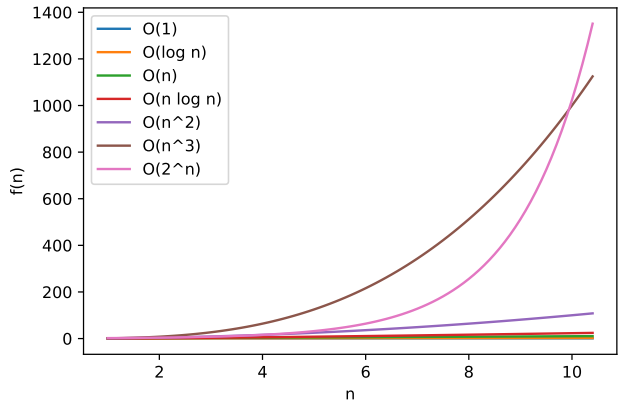

Figure 6: An illustration of the growth rates of common functions.

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

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

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 *k*NN

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 *k*NN 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 $O(n \log(k))$. For instance, we could implement a priority queue using a heap data structure [8].

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"[9]. 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[10], 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.[11]

Ball-Tree

In contrast to the KD-Tree approach, the Ball-Tree[12] partitioning algorithms are based on the construction of hyperspheres instead of cubes. While Ball-Tree algorithms are generally

[8] 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

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

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

[11] 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.

[12] 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 KD-Tree 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.

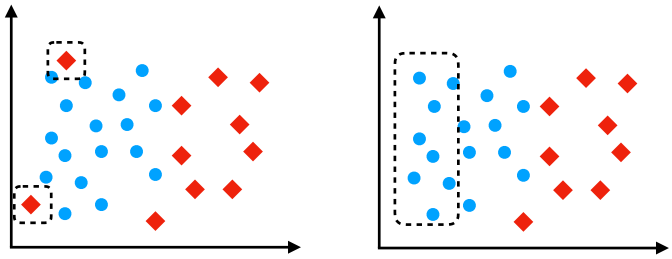

Figure 7: Illustration of kNN editing, where we can remove points from the training set that do not influence the predictions. For example, consider a 3-NN model. On the left, the two points enclosed in dashed lines would not affect the decision boundary as "outliers." Similarly, points of the "right" 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.

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.

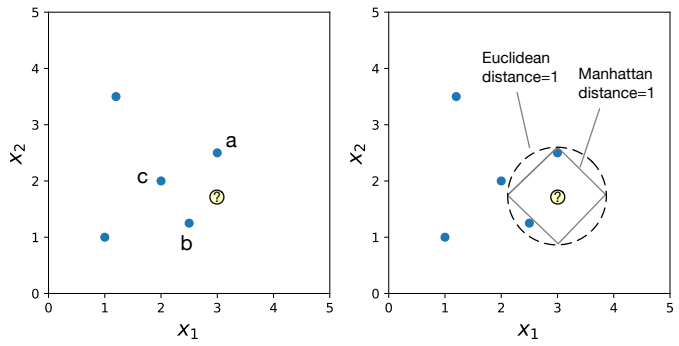

Figure 8: The phrase "nearest" is ambiguous and depends on the distance metric we use.

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