# 1 Learning Outcomes

- Familiarize yourself with time and space complexity of KNN algorithms.
- Describe how KNN algorithms can be used for both classification and regression tasks.
- How does the decision boundary complexity change for KNN models with a change in hyperparameters.
- List and describe appropriate distance metrics to use with KNN models. When is it beneficial to use one distant metric over another?
- What is the curse of dimensionality and how does it relate to KNN models.
- Describe advantages and disadvantages of KNN models.

# 2 Overview

The k-nearest neighbors algorithm, also known as KNN or K-NN, is a simple machine learning model that can be used for regression or classification. It assumes no specific functional form for how a feature vector is related to its label. It instead assumes that data samples that are close together (in some appropriate distance metric) will have the same classification.

To illustrate how the algorithm works, consider the example shown in figure 1. The example shows 2-dimensional training data, where each data sample can belong to one of three classes. To classify the unlabeled data (shown in gray), we look for the K closest data points (in this case k=5) and assign the majority class label to the unlabeled data sample.

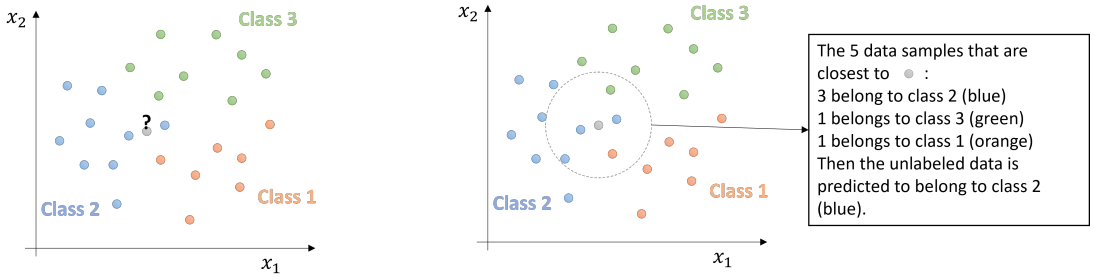



Figure 1: The KNN approach, using K = 5, is illustrated in a simple example.

The algorithm was first proposed in 1951 by Fix and Hodger (Link to the paper). Then in 1967, Thomas cover provided further analytical results for the algorithm (Link to the paper). Despite its simplicity, KNN model has been used within a variety of applications. Recommendations engine is one example of such applications: a recommendation system looks for clients with same profile in order to suggest similar products or it looks for product that are similar to a client purchase history. For a list of other KNN applications, check here.

# 3 KNN: Training and Prediction Phase

**Training phase**

KNN assumes nothing about the functional form of the relationship between a feature vector and its label. Since there's no specific function to learn, there is no training phase for this algorithm. The training data is the model itself. KNN is an example of **lazy learner** because there is no cost associated with the training phase of the algorithm, and the algorithm just needs to store the training data. This **memory-based learning** is also known as **instance-based learning** which consists of comparing new data instances with instances seen in training, whereas model-based learning tries to generalize the relationship between data samples and their labels.

When we discussed linear models, we mentioned that they are considered parametric models, since they summarize the training data with a finite set of parameters (irrespective of the number of samples in the training data). However, KNN is a **non-parametric model** since it cannot be described with a finite set of parameters. You can consider the training data set itself as the building blocks or the parameters for the KNN algorithm, which obviously grow as the size of the training data grows and hence the model cannot be described with a finite set of parameters.

**Prediction phase**

The real work of the algorithm happens during the prediction phase but it is straight-forward as illustrated in the figure 1. All what we need to identify first is K (the number of nearest neighbors whose labels will be used for prediction) and the choice of distance metric (for example, Euclidean distance).

In the prediction phase of KNN, we have a set of unlabeled data points that we're interested in predicting their label, which we also call the **query points**. The labeled or training dataset is called **the reference points**. The algorithm can be summarized by the following 5 steps:

1. Given a query point, we calculate the distance between the query point and every reference point.
2. Sort the reference points by their distance from the query point.
3. Keep the K closest reference points (K nearest neighbors).
4. Grab the labels of the K nearest neighbors.
5. Aggregate the labels of the neighbors to predict the label of the query point: in classification choose the most common label (majority vote or mode); in regression, calculate the mean or median of the labels.

And these steps should be repeated for every query point to get the predicted label for all the query points.

**Time and space complexity of the algorithm**

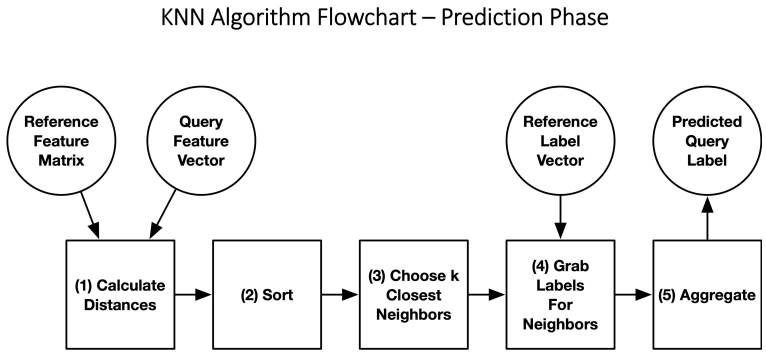



Figure 2: The flowchart of the KNN algorithm.

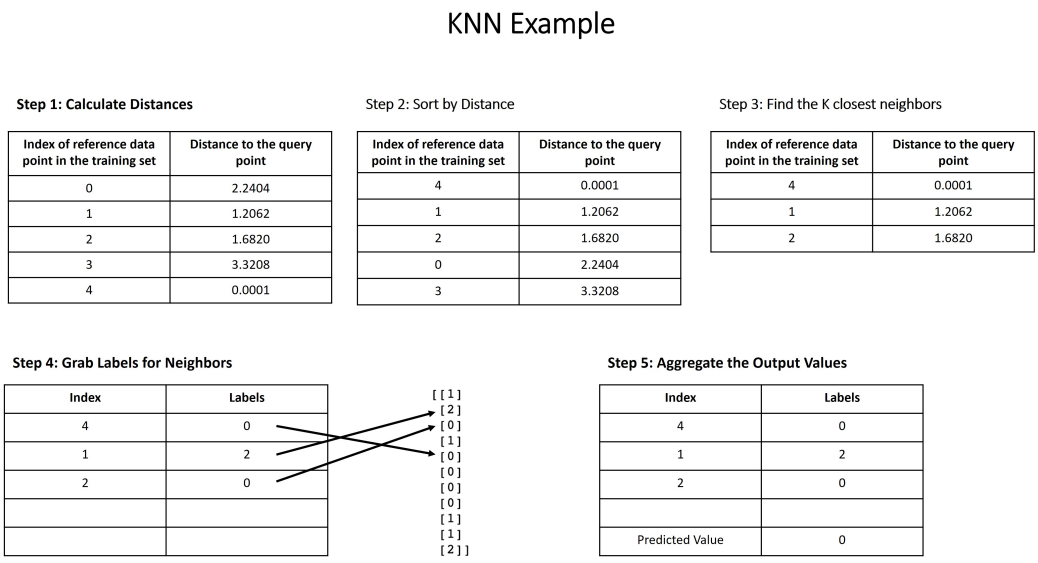



Figure 3: Example illustrating how KNN predicts the label of a given query point

Now let's look at the space and time complexity of the algorithm when predicting the label of one query point. Assume there are $n$ reference points, $m$ features, 1 query point and $k$ neighbors. Then the time complexity of each step is as follows:

- Compute distances: To compute the distance between the quey point and a point from the reference set, we need to loop through all the features $m$. We need to compute this distance between the query point and every point in the training set, so we need to repeat the computation of the distance $n$ times. Therefore, the time complexity of this step is $O(nm)$.
- Sort the neighbor list (using a heap): To get the reference points with the $K$ smallest distances could be done in more than one way.
    1. Using a min-heap: we can construct a min-heap that stores the $n$ distances and then extract the $k$ smallest values. This would require $O(n)$ to build the min-heap. (source)

and $O(k\log(n))$ to extract the $k$ smallest values. Therefore the time complexity of this step is $O(n + k\log(n))$.

2. Using a max-heap: we can construct a max-heap of only $k$ elements. We start with constructing the heap with the first $k$ distances and then after that, for each new distance we compare it with the root of the max-heap. If the new distance is greater than the root, it is ignored. Otherwise, the root is removed and the new distance is inserted in the max-heap. Therefore the time complexity of this step is $O(k + (n-k)\log(k))$

- Get known labels and aggregate them: To get the $k$ labels from the label vector and aggregate them, this step would take $O(k)$.

Thus the total time complexity is $O(nm + n + k\log(n) + k)$ if a min-heap is used or $O(nm + k + (n-k)\log(k) + k)$ if a max-heap is used. In both cases if $k$ is much smaller than $n$ and $m$, the dominating term is $O(nm)$. This time complexity is to compute the predicted label for one query point, which will increase the overall time complexity ($O(nmp)$ if $p$ is the number of query points). Different data structures have been proposed to improve the time complexity of KNN during prediction, by being more efficient about finding the $k$ nearest neighbors. These data structures try to reduce the required number of distance calculations, instead of computing the distance between each training example in the training set and a given query point (K-D Tree, Ball Tree, Sklearn guide to NN).

Regarding the space complexity, the "training" step of KNN merely involves storing the $n \times m$ feature matrix and $n$-length label for the reference points. Therefore, the space complexity is $O(nm + n) = O(nm)$.

# 4 What does affect KNN's predictive power?

The choice of $k$ (number of nearest neighbors), the choice of distance metric and feature scaling are all options that can affect the performance of KNN.

## 4.1 Choice of $k$

The choice of $k$ is critical because it determines the model's complexity. Figure 4 shows how the shape of the decision boundaries changes with changing $k$. With $k = 1$, the decision boundary is overly complex and overly specializes to the training data. As $k$ increases the decision boundary becomes smoother. With a very high $k$ ($k = 100$), the decision boundary becomes too simple and close to a linear decision boundary. Choosing the right $k$ is therefore about finding a good balance between overfitting and underfitting. For very small values of $k$, the model can be sensitive to data noise, and thus can have high variance and can be prone to overfitting. For very large values of $k$, the model considers far points as neighbors which will lead to a model with high errors, and thus high bias or underfitting. Figure 5 shows how the training and validation errors of a KNN model can change with changing $k$. So how to choose $k$?

Choosing the appropriate value of $k$ is part of the model selection that occurs during the training phase. We can perform cross-validation to choose the value of $k$ that gives the best average performance on the validation folds. Figure 6 shows for each possible value of $k$ the

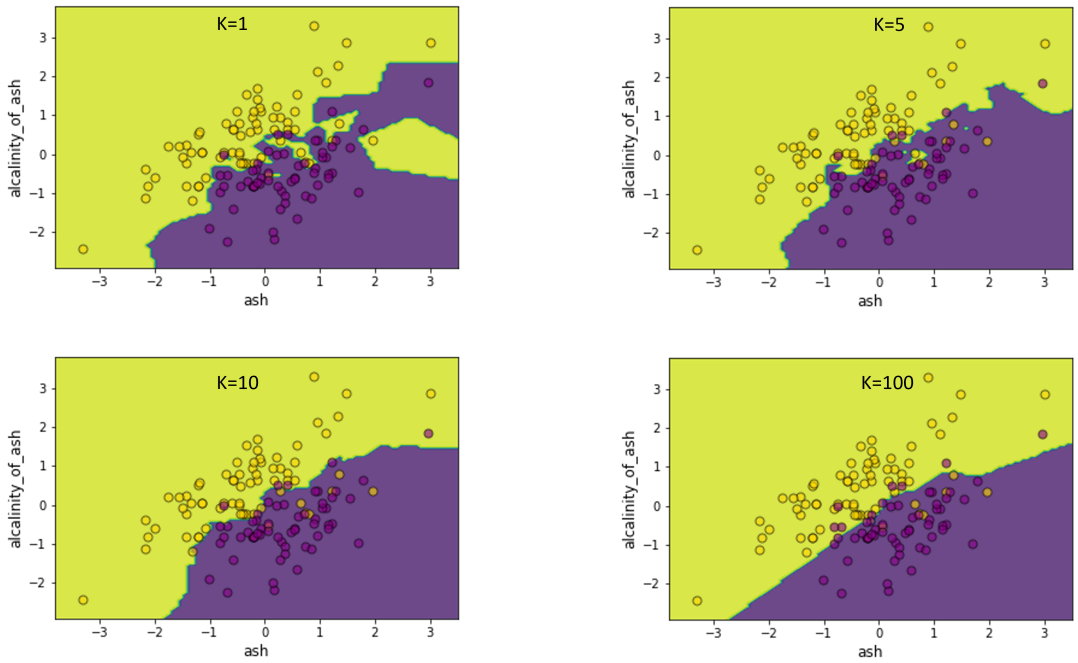



Figure 4: Complexity of KNN's decision boundary changes with changing $k$, the number of nearest neighbors.

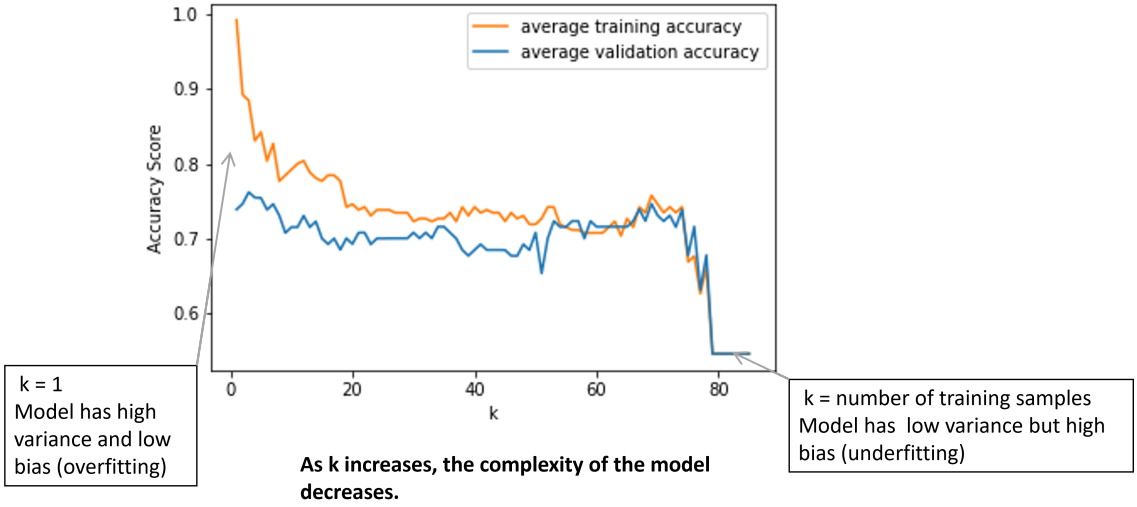



Figure 5: The complexity of a KNN model changes with changing $k$. For very large $k$, the model is too simple so that the accuracy on the validation and training sets is low. For $k = 1$, the model fits the training data too closely, this is why there is a gap between the training and validation accuracies.

average validation accuracy of the KNN model (blue plot), obtained by performing 3-fold cross validation on the dataset shown in figure 4. The best average validation accuracy was obtained for $k = 3$, so we choose 3 for $k$. Note that on the same figure, the training plot was also plot to

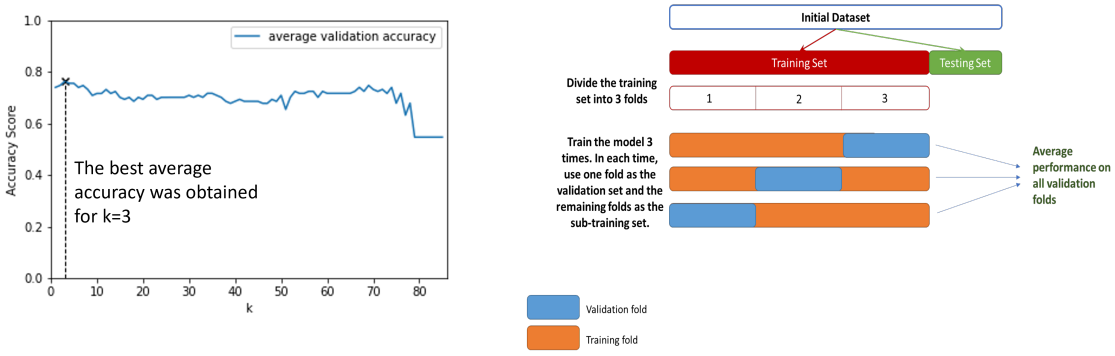



Figure 6: Choosing the best value of $k$ is performed as part of the model selection using cross-validation.

### 4.2 Choice of distance metric

The most commonly used distance metric is the Euclidean distance, which is given by:

$$\text{Euclidean distance} = d(\mathbf{x}^{(i)} - \mathbf{x}^{(q)}) = \sqrt{\sum_{j=1}^{m} \left(x_j^{(i)} - x_j^{(q)}\right)^2}$$

where $\mathbf{x}^{(i)}$ is the feature vectors of the $i^{th}$ data sample in the reference set, and $\mathbf{x}^{(q)}$ is the feature vector of a given query point. But this is not only the distance metric that could be used.

Manhattan distance is another distance that is given by:

$$\text{Manhattan distance} = d(\mathbf{x}^{(i)} - \mathbf{x}^{(q)}) = \sum_{j=1}^{m} \left|x_j^{(i)} - x_j^{(q)}\right|$$

Manhattan distance, also known as city block distance, or taxicab geometry is the distance between two data points if we only allowed to travel in a grid-like path. In figure 7, the length of the red diagonal line represents the Euclidean distance, while the other lines could represent Manhattan distance (any of them). While Euclidean distance is suitable when features are continuous, Manhattan distance can be used when the features are just binary (categorical binary features, i.e.e a vector with 0 and 1 elements). Manhattan distance is also more preferable to use when the data is high dimensional, this is due to a phenomenon known as the curse of dimensionality that will be shortly explained.

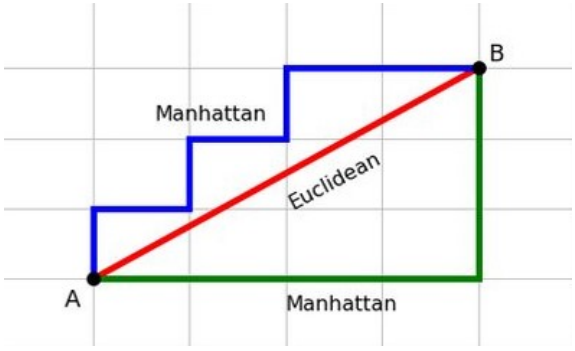



Figure 7: The difference between Manhattan and Euclidean distances.

Euclidean or Manhattan distance can be generalized as Minkowski distance, which is given by:

$$\text{Euclidean distance} = d(\mathbf{x}^{(i)} - \mathbf{x}^{(q)}) = \left[\sum_{j=1}^{m} \left(x_j^{(i)} - x_j^{(q)}\right)^p\right]^{\frac{1}{p}}$$

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

A more sophisticated metric known as the Mahalanobis distance that takes into account the variance of different features can be also used. In natural language processing, a text can be represented by a vector of features. To measure the similarity between the two texts, cosine similarity is used to measure the closeness of the two feature vectors. When selecting the distance, it is important to consider the context- or problem-dependent and choose the best appropriate distance measure.

### 4.3 Importance of feature scaling

When computing the distance between two data points, if one feature has much bigger range of values than others, this feature will end up contributing to the total distance more than the other features. If we don't scale the features, the performance of KNN model will be affected negatively, especially if we're using Euclidean distance. This is because Euclidean distance squares the difference between the values of features, so that variations in the values of the features with greater range of values will be magnified more than those of other features. This means that small variations in the feature with greater range of values will have more effect on the distance than the large variations in other features. The model will end up choosing the wrong neighbors. Figure 9 shows how the decision boundary of KNN is sensitive to feature scaling. It is therefore important to scale the features before we train A KNN model to make sure that each feature contributes equally to the overall distance.

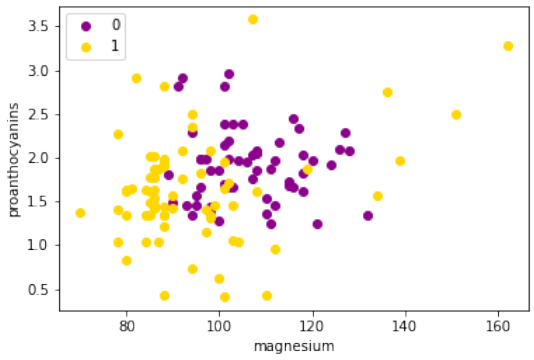
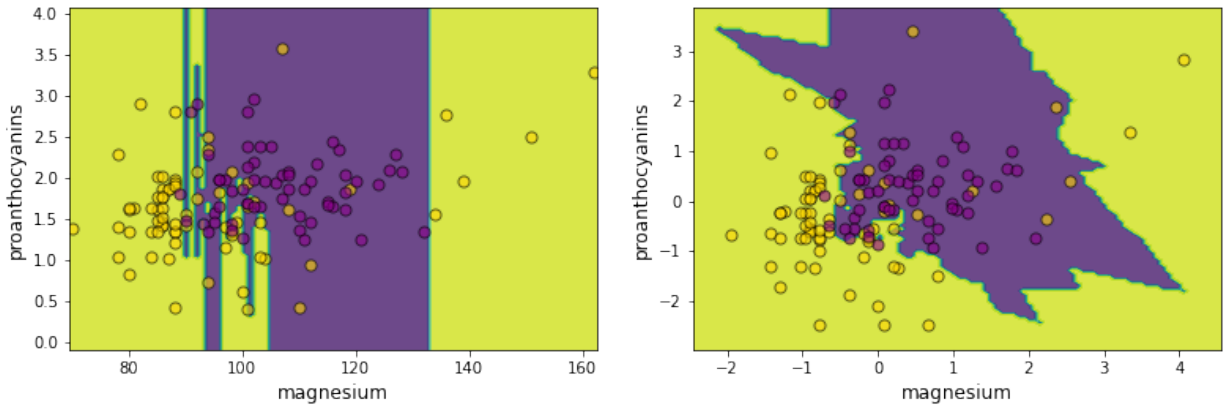



Figure 8: The two features have different range of values.




(a) KNN - No Scaling (b) KNN - Scaling

Figure 9: Knn is sensitive to feature scaling.

## 5 Curse of dimensionality

KNN works very well in low-dimensional feature space with lots of data samples: for a given query point, there are likely enough nearby data points so we can a label that is accurate enough. However, as the number of dimension increases, KNN encounters the problem of **curse of dimensionality**: the nearest neighbors in high-dimensional feature space might not be close enough, so they might not be similar enough to have similar label.

"*The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset*" - Python machine learning, page 120-121 (ebook)

In high-dimensional feature space, the data points become more sparse or scattered (especially if we don't have lots of training samples)[1], so that the nearest points to a given query point might be too far. The problem with considering neighbors that are so far away is that they may not be good predictors about the label of the query point, leading to a very poor prediction of KNN. In machine learning, we can use feature selection or dimensionality reduction techniques to avoid the curse of dimensionality.

## 6 Summary: advantages and disadvantages

Like any machine learning models, KNN has its strengths and limitations.

**Advantages**

- **Simple model**: The model has **no training step** and the prediction phase of the algorithm is **easy to implement**. The model requires **few hyperparameters**: choice of $k$ and distance

[1]To see why this is the case, we will have an exercise in the assignment about the curse of dimensionality

metric. Other models require the use of more hyperparameters.

- **Non-linear model**: The model does not assume any specific functional form, which results in more flexible decision boundaries (classification) or model's response (regression) than those of linear models.
- **Adapts easily to new training samples**: Since the training sample is the model itself, the model adjusts easily to new training samples. The new samples needs to be added to the old training samples and both stored in memory.
- **Extends easily to multi-class problem**: The prediction algorithm of KNN works no matter how any classes to classify, it does not only work for binary classification. When predicting the label of a query point, we choose the majority vote label from its nearest neighbors.

**Disadvantages**

- **Memory and computation costly**: Since KNN is a lazy learning, the training data needs to be entirely stored, which might not be efficient from memory and storage perspective. Moreover, the prediction run time is slower than many other models.
- **Curse of dimensionality**: As already explained, KNN does not performance well with high-dimensional data inputs.
- **Sensitive to feature scaling**: The performance of KNN is also affected if features have different scales, this is why it is important to make sure that features are scaled before applying KNN algorithm.
- **Interpretability**: Linear models are considered interpretable model because the weights can give us insight into the contribution of each feature to the model. However, KNN is not able to do this. When computing the distance between two data points, all features are considered equally. An unpredictive feature can introduce noise into the computation of the distances.