LEC1: Instance-based classifiers

Guangliang Chen

September 5, 2018

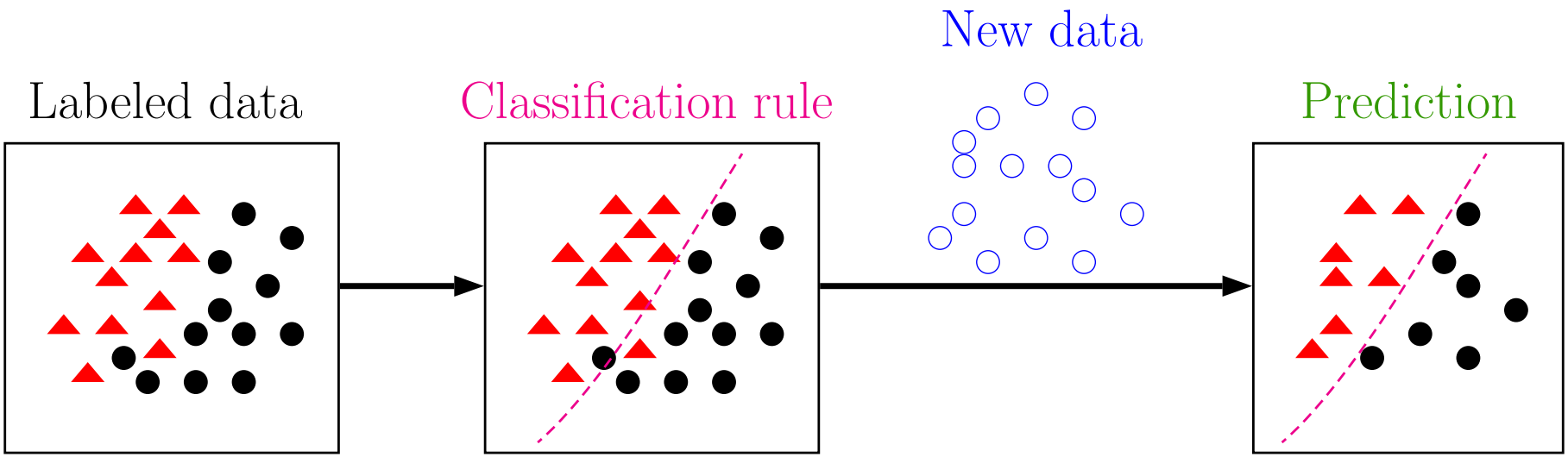
Outline

- kNN: algorithm, software, theory, and variants
- Nearest centroid classifier
- Summary

Recall

...the machine learning classification problem: Given training data and their labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{1, \ldots, k\}$, learn a classification rule (under certain criterion)

$$f : \mathbf{x}_0 \in \mathbb{R}^d \longrightarrow y_0 \in \{1, \ldots, k\}$$



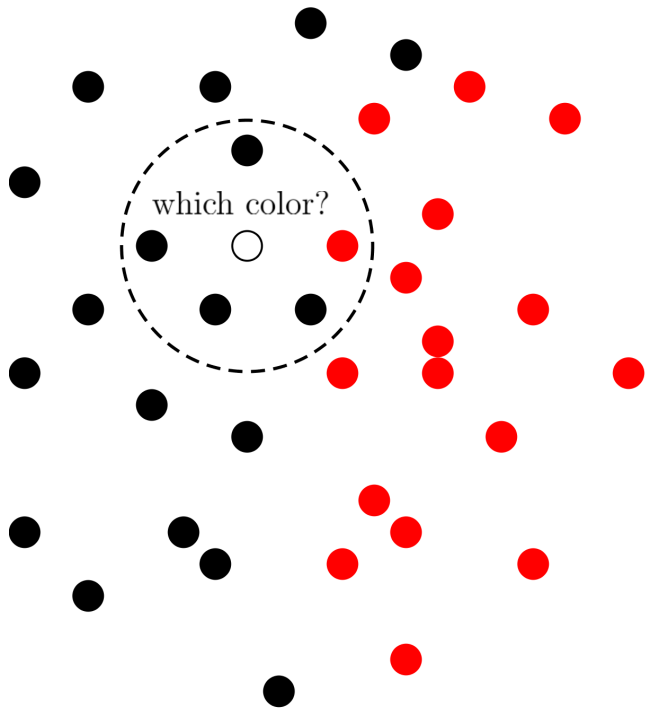
The k nearest neighbors (kNN) classifier

Main idea: Determine the label of a new point $\mathbf{x}_0$ based on those of k closest training points through majority voting:

$$\hat{y}_0 = \arg\max_j \sum_{\mathbf{x}_i \in \mathcal{N}_k(\mathbf{x}_0)} I(y_i = j),$$

where $\mathcal{N}_k(\mathbf{x}_0)$ denotes the set of k nearest neighbors of $\mathbf{x}_0$ in the training set.

It is one of the simplest and most commonly used classifiers in practice.



Critical questions in kNN classification

- What do we mean by "close"? $\longleftarrow$ distance metric
- How large should k be? $\longleftarrow$ bias-variance tradeoff
- How fast is it? $\longleftarrow$ complexity
- How accurate is it? $\longleftarrow$ model
- What do we know about its theoretical properties (e.g., linear/nonlinear, advantages, and limitations)? $\longleftarrow$ analysis

Common distance metrics (for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$)

- **Minkowski** (ℓ_p for any $p > 0$): $\|\mathbf{x} - \mathbf{y}\|_p = \left(\sum |x_i - y_i|^p\right)^{1/p}$
  - **Euclidean** ($p = 2$): $\sqrt{\sum (x_i - y_i)^2}$
  - **Manhattan/ City-block** ($p = 1$): $\sum |x_i - y_i|$
  - **Chebyshev** ($p = \infty$): $\max |x_i - y_i|$
- **Cosine of the angle**: $1 - \left\langle \frac{\mathbf{x}}{\|\mathbf{x}\|_2}, \frac{\mathbf{y}}{\|\mathbf{y}\|_2} \right\rangle = 1 - \cos\theta$
- **Correlation coefficient**: $1 - \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2} \sqrt{\sum (y_i - \bar{y})^2}}$

Complexity of kNN classification

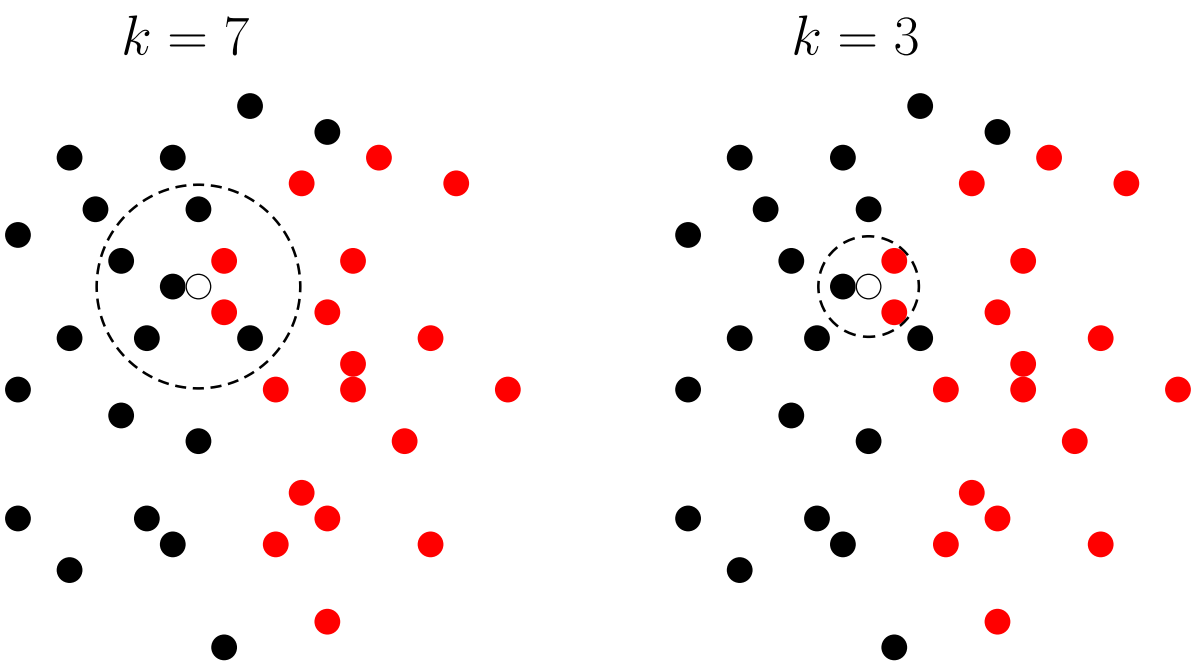
Memory: $\mathcal{O}(nd)$, as it needs to store all the training data

Speed: $\mathcal{O}(n(d+k))$ for a single test point (through direct implementation). For m test points, the overall complexity is $\mathcal{O}(mn(d+k))$.

Computationally, this is a heavy burden for large data sets in high dimensions (e.g., for MNIST handwritten digits, $n = 60,000, m = 10,000, d = 784$).

How to select k

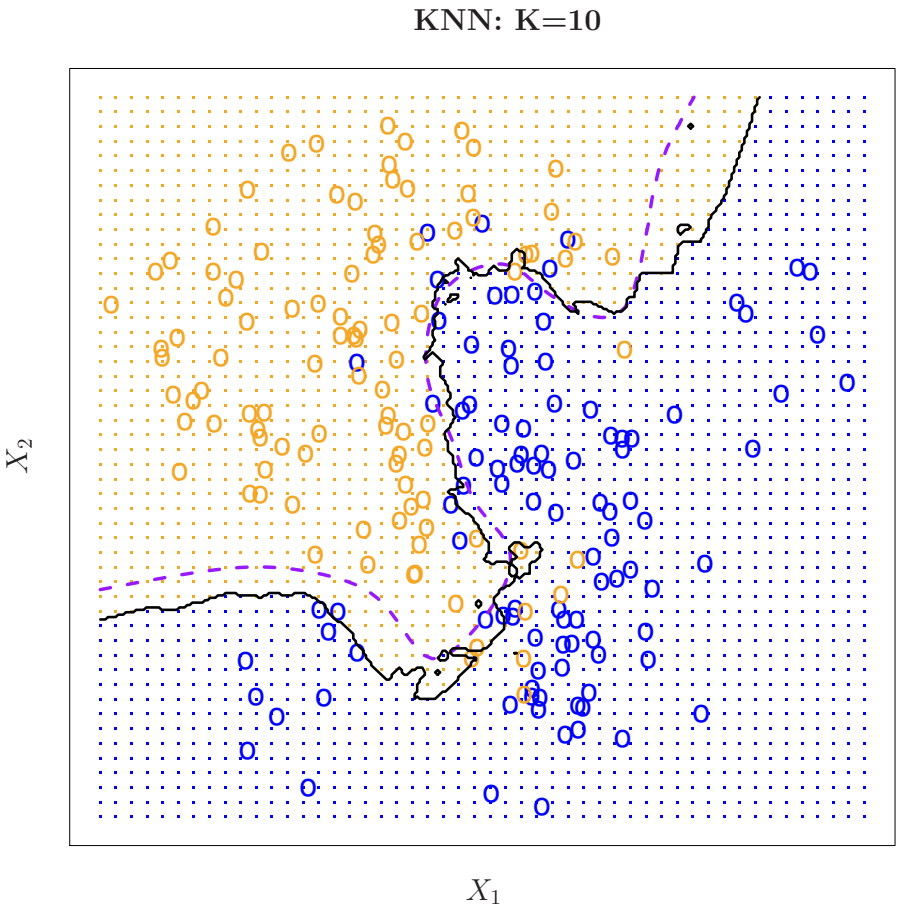
Once we have chosen a proper distance metric, we can train a kNN classifier with any value of k. However, different choices of k may lead to different predictions for the same test point.



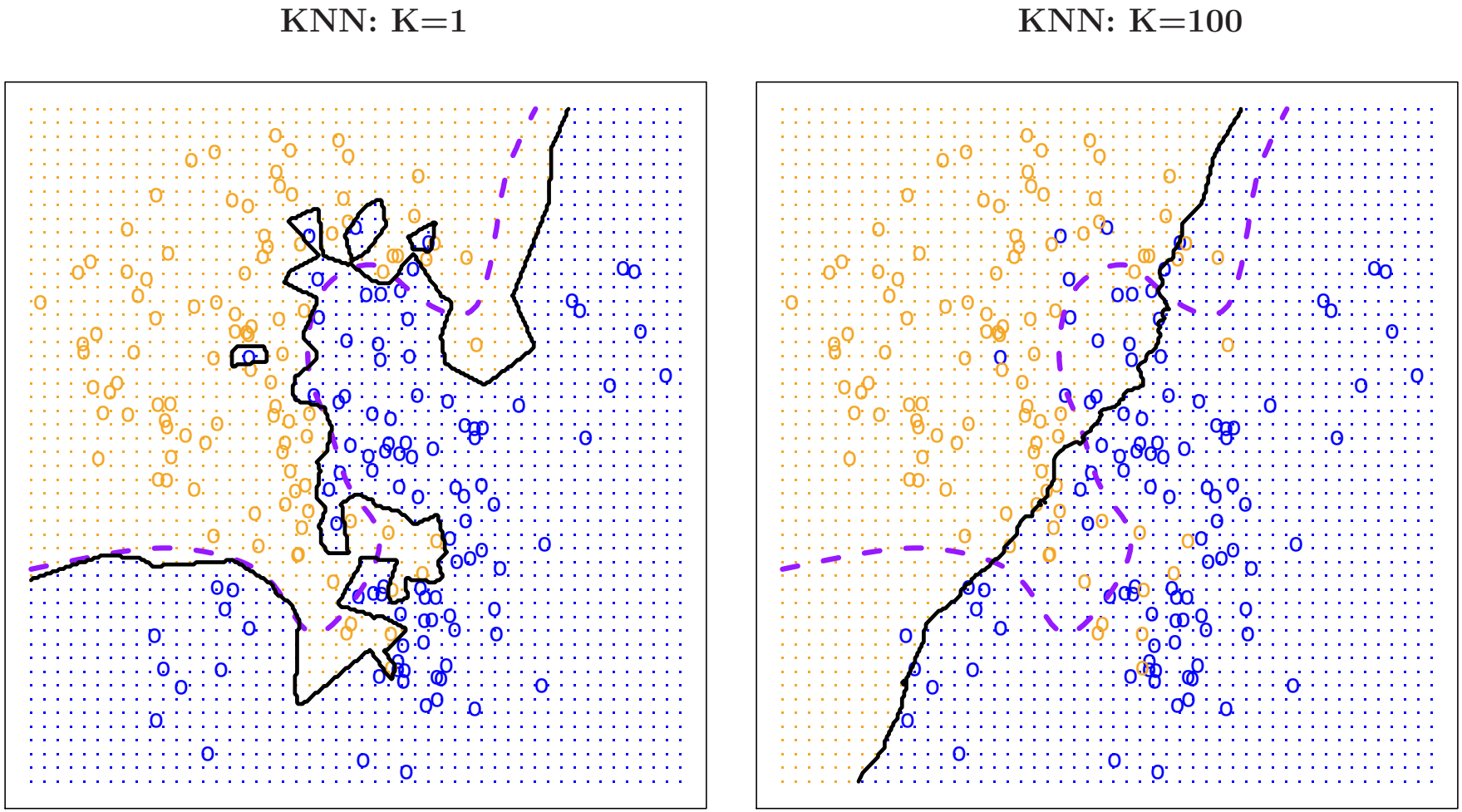
We can also study the influence of the parameter k on the kNN classifier through decision maps.

The plot on the right (taken from text-book) shows a simulated data set with the **model boundary** (in purple) and the **decision boundary** (in black) of the 10NN classifier.

What about other values of k?



Instance-based classifiers



Clearly, the choice of k is crucial and different values of k lead to different kinds of decision maps.

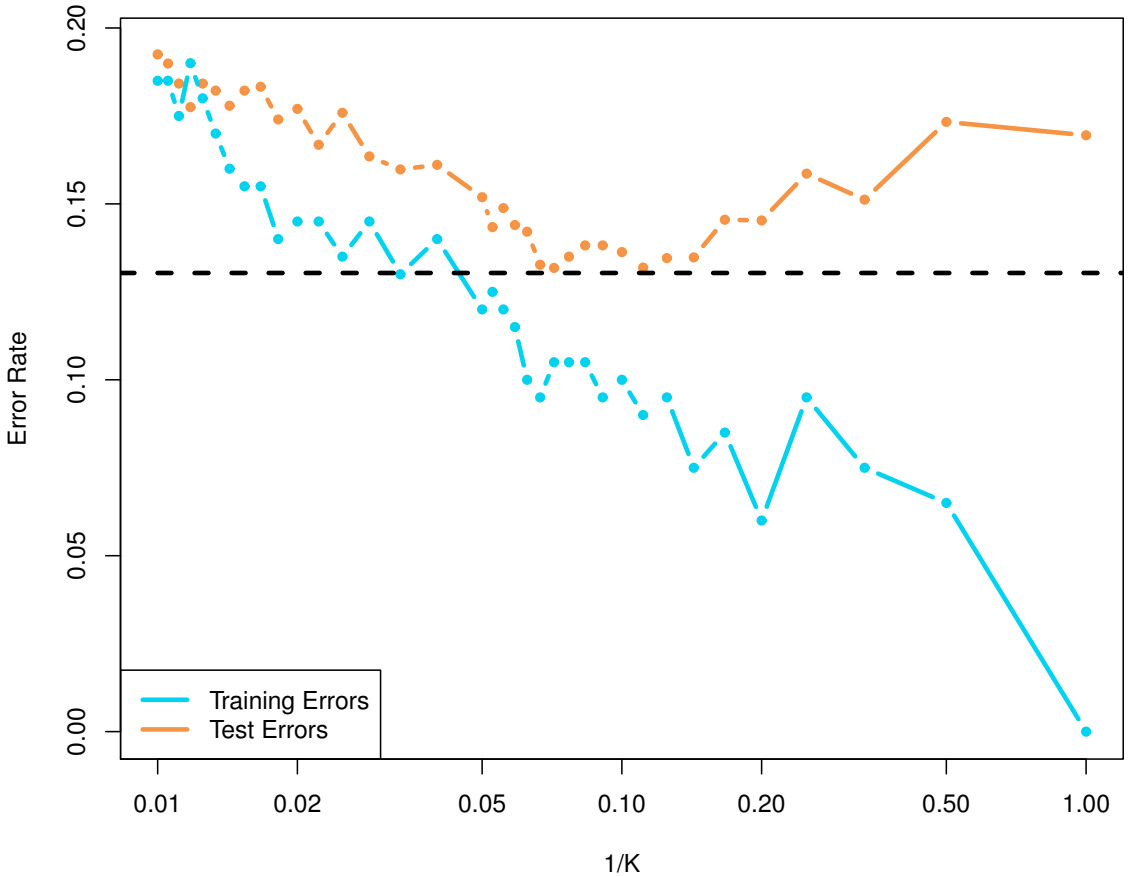
For test data which in reality has no labels available, we cannot compare different values of k and determine which one is optimal.

So the only possibility is to use the training data itself to tune the parameter k.

However, the decision map is only a qualitative criterion and also expensive to produce.

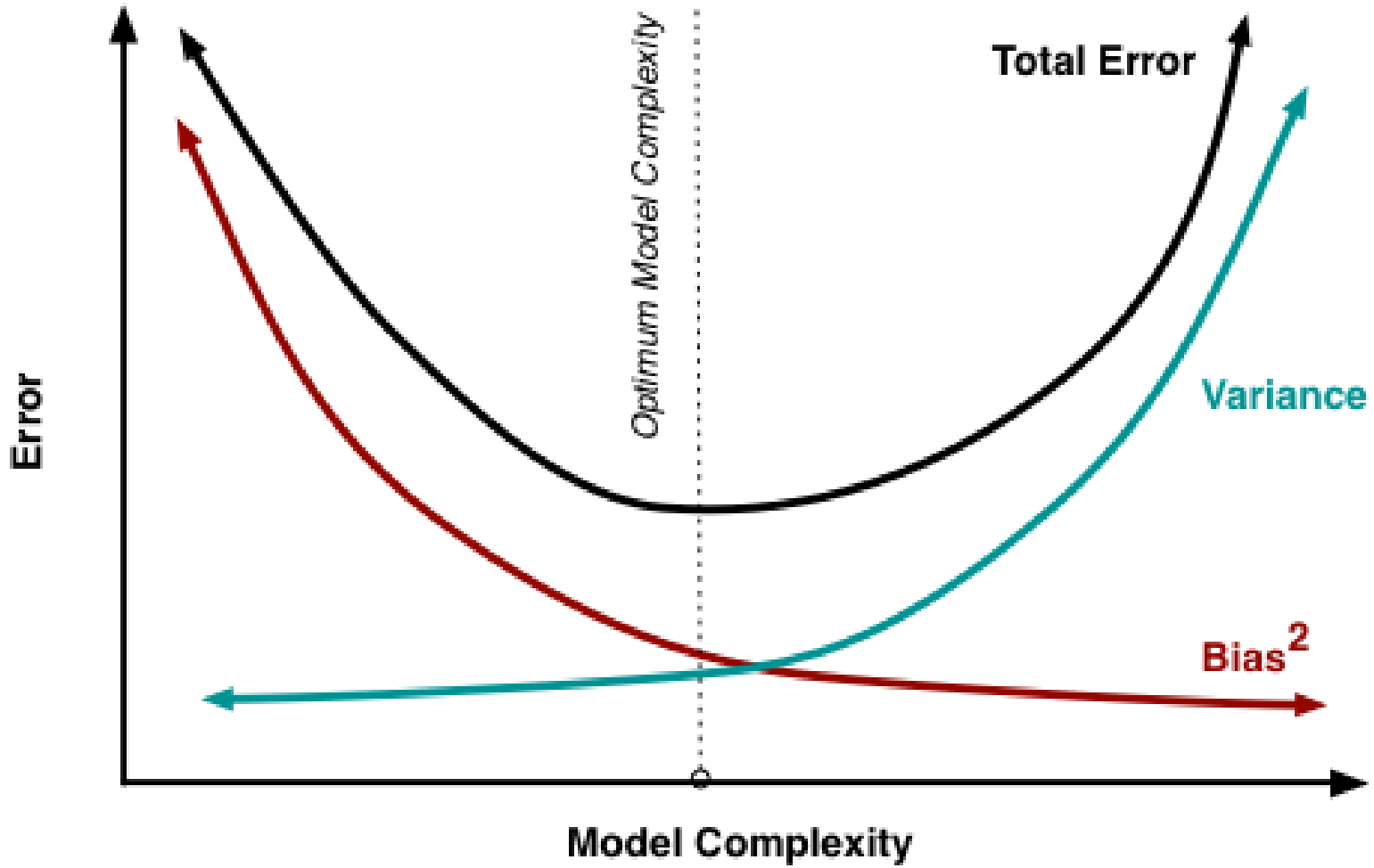
Is there a (better) quantitative way?

First idea: For each $k = 1, 2, \ldots$, apply the kNN classifier to the training data and compute the corresponding training error: $e_{\text{tr}}(k) = \frac{1}{n} \sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$.



A few notes and remarks:

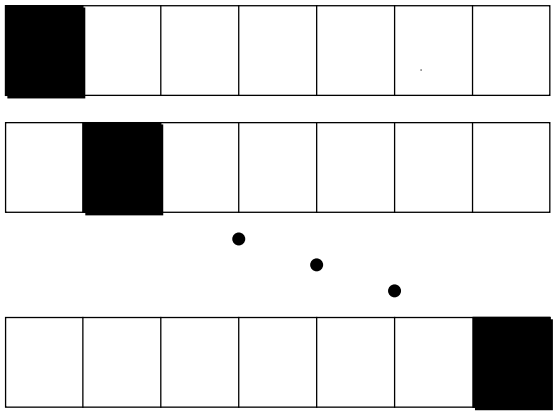
- The experiment used 200 training points and $5,000$ test points.
- $1/k$ measures model complexity/ flexibility.
- $k = 1$ minimizes the training error, but the model generalizes poorly to test data. $\longleftarrow$ We will thus abandon this tuning technique for selecting k.
- The optimal k is around 10, representing a good tradeoff between model fidelity (bias) and complexity (variance).
- Smaller k overfits the data while bigger k underfits the data.
- Such a phenomenon also occurs in the setting of regression (see next slide).


Error
Optimum Model Complexity
Total Error
Variance
Bias^2
Model Complexity

Second idea: Use separate parts of training data for training and validation.

Algorithm. Fix an integer m (e.g. 10). For each value of k do the following:

- *Partition the training set randomly into m equal sized-subsets (called folds).*
- *Of the m subsets, each one is retained as validation data once and the remaining $m-1$ are used as training data*



A combined validation error for each k may be computed:

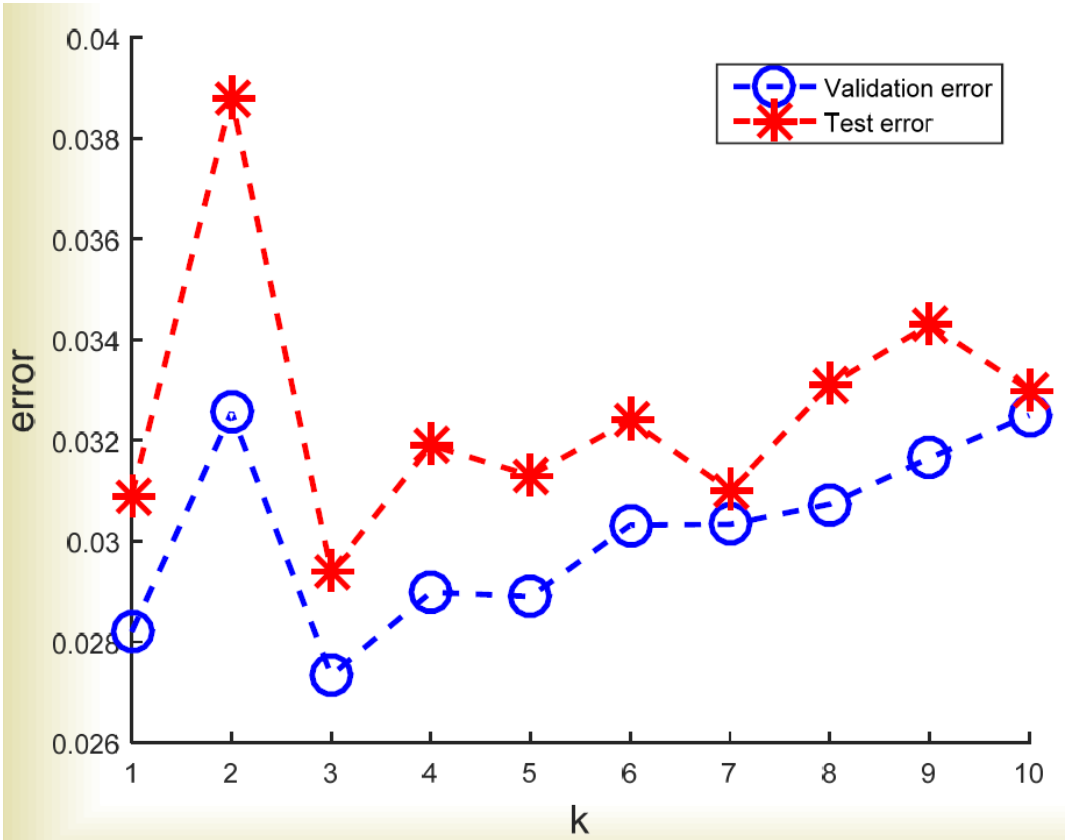
$$e_{\mathrm{CV}}(k) = \frac{1}{m}\sum_{i=1}^{m} e(i).$$

This full procedure is called (m-fold) **cross validation (CV)**.

When $m = n$ (i.e., leave-one-out CV), this is also called resubstitution error.

Cross validation on the MNIST digits

This idea really works!



Summary: evaluation criteria for a classifier

- On test data
  - Time
  - Test error (if true labels available)
  - Confusion matrix (if true labels available)
- On training data
  - Validation error
  - Decision map

MATLAB commands for kNN classification

% fit a kNN classification model

mdl = fitcknn(trainX, trainLabels, 'NumNeighbors', 3);

% apply the model to test data for prediction

pred = predict(mdl, testX);

Evaluating the kNN classifier in MATLAB

```
% compute test error
testError = 1 - sum(testLabels ==pred)/numel(testLabels)

% examine confusion matrix
cMat = confusionmat(testLabels, pred)

% calculate 10-fold cross validation error for the above k
cvmdl = crossval(mdl,'kfold',10);
kloss = kfoldLoss(cvmdl)

% can also calculate the resubstitution error
rloss = resubLoss(mdl)
```

kNN classification with other distance metrics

mdl = fitcknn(trainX, trainLabels, 'NumNeighbors', 3, 'Distance', DISTANCE);

Common choices of the string variable DISTANCE are:

'euclidean' - Euclidean distance (default)

'cityblock' - City Block distance

'chebychev' - Chebychev distance (maximum coordinate difference)

'cosine' - One minus the cosine of the included angle between observations (treated as vectors)

'correlation' - One minus the sample linear correlation between observations (treated as sequences of values).

Python and R codes for kNN classification

- R: See textbook (Section 4.6.5)
- Python:
  - kNN in Python[1]
  - Kevin Zakka's blog[2].
  - Ryan's lecture[3]

[1] https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

[2] https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/

[3] https://github.com/ryanshiroma/Python-Intro-for-Math-285

The probabilistic model for kNN classification

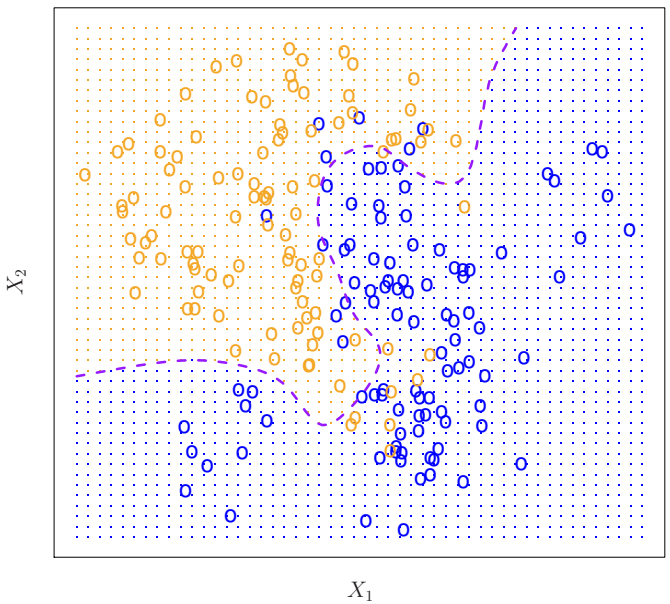
Suppose training and test data come from the same distribution where the probabilities of observing points from k different classes are:

$$\{(p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})) \mid \mathbf{x} \in \mathbb{R}^d\}$$

and the observed label at each location is a multinomial random variable:

$$Y \mid \mathbf{x} \sim \text{MN}(p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})).$$

For 2 classes (as shown in the figure), the purple curve corresponds to locations with equal probability (i.e., $p_1(\mathbf{x}) = p_2(\mathbf{x})$) while on each side, points with the dominant color have probabilities larger than $\frac{1}{2}$.



The Bayes classifiers

If one knows the model probabilities $p_1(\mathbf{x}), \ldots, p_k(\mathbf{x})$ everywhere in space, then one may form the so-called **Bayes classifier**

$$\hat{y}_0 = \arg \max_{1 \le j \le k} P(Y = j \mid \mathbf{x}_0)$$

with **Bayes error rate**

$$1 - E_{\mathbf{x}_0} \left(\max_{1 \le j \le k} P(Y = j \mid \mathbf{x}_0) \right)$$

The Bayes classifier is optimal in the sense that this is the lowest possible test error rate that may be achieved by a classifier.

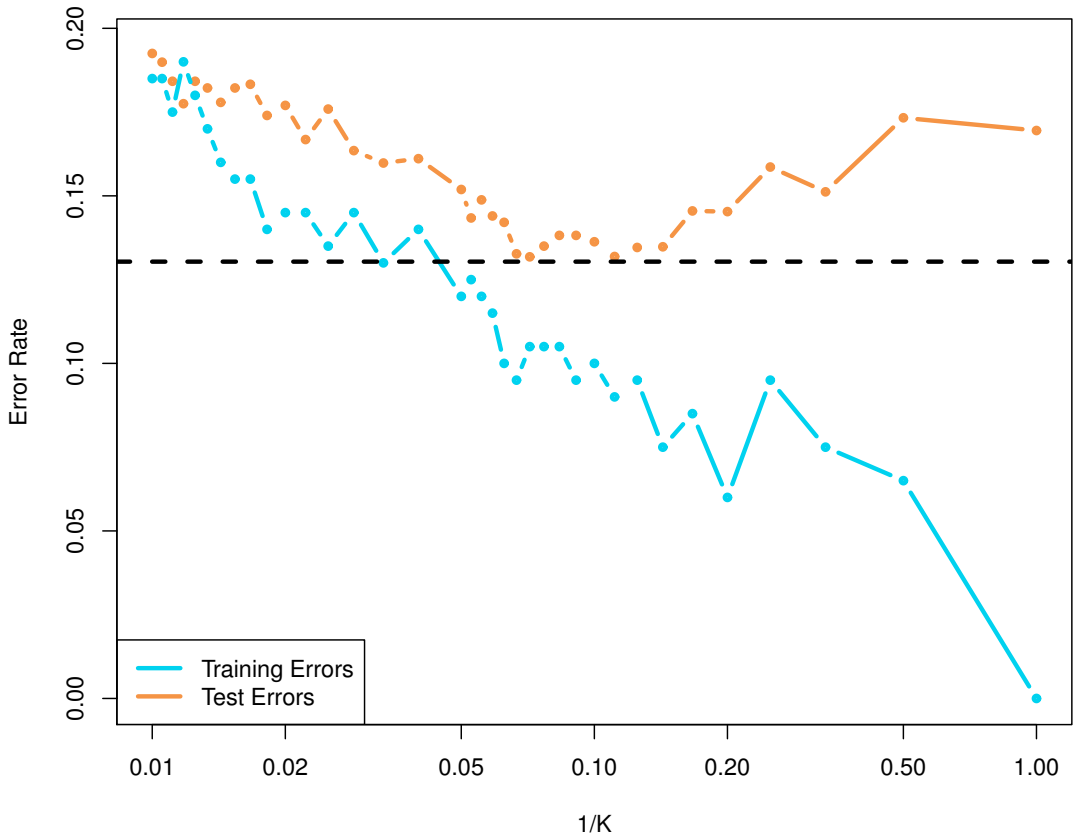
Recall that

$$P(Y = j \mid \mathbf{x}_0) \propto f(\mathbf{x}_0 \mid Y = j) \cdot P(Y = j) = f_j(\mathbf{x}_0) \cdot \pi_j$$

where

- $f_j(\mathbf{x}_0) = f(\mathbf{x}_0 \mid Y = j)$: **density function** of the jth class;
- $\pi_j = P(Y = j)$: **prior probability** that a new observation would come from class j (before seeing it)
- $P(Y = j \mid \mathbf{x}_0)$: **posterior probability** that $\mathbf{x}_0$ came from class j (after seeing it)

In the previous simulation, the Bayes error rate was .1304 (indicated by the black dashed line).

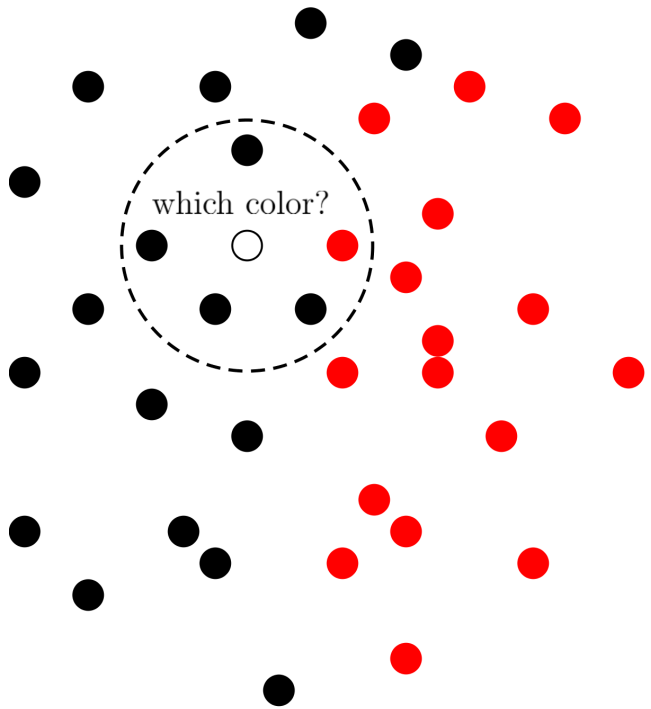


In reality, it is impossible to construct the Bayes classifier since we do not know the conditional distribution of Y (label) given each $\mathbf{x}$ (location).

kNN classification can be regarded as a way to approximate the posterior probabilities at each location $\mathbf{x}$:

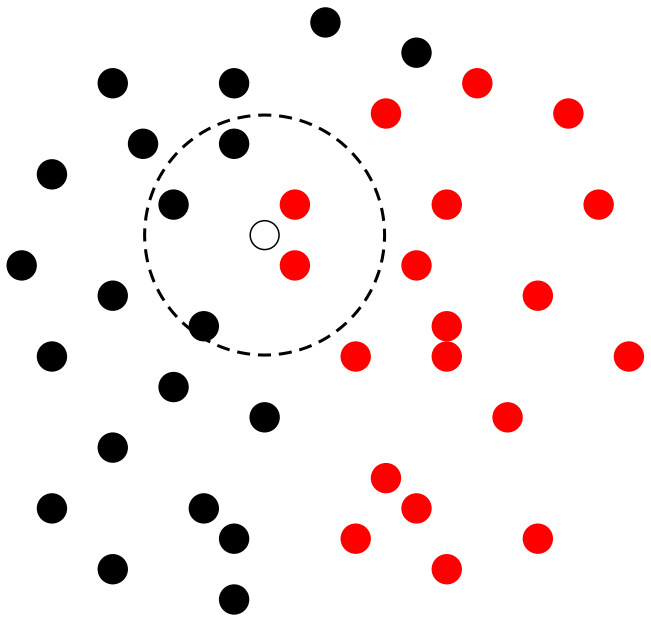
$$P(Y = j \mid \mathbf{x}) \approx \frac{1}{k} \sum_{\mathbf{x}_i \in \mathcal{N}_k(\mathbf{x})} I(y_i = j)$$

This explains why the kNN classifier works very well when given "many" training examples.



Weighted kNN

Consider the following example:



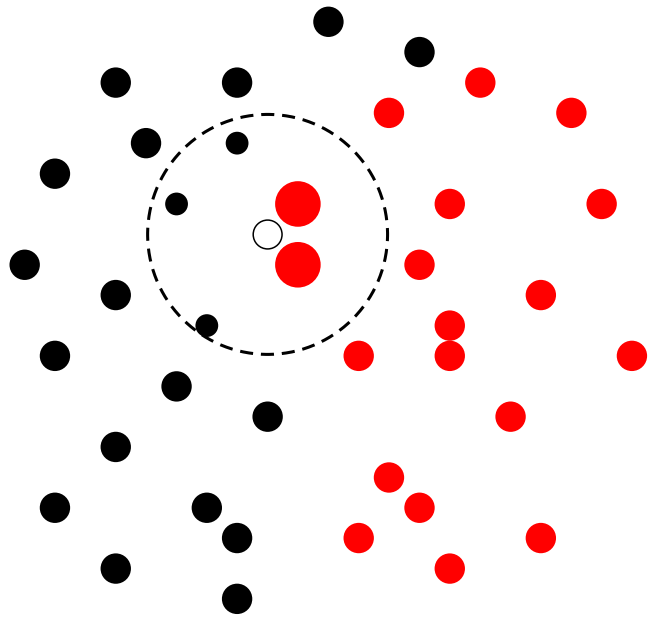
What is your prediction for the label of the test point?

To color it in red (even with a wrong choice of k), one strategy is to give closer neighbors larger voting weights.

Mathematically, this is achieved as follows:

$$\hat{y} = \arg\max_j \sum_{\mathbf{x}_i \in \mathcal{N}_k} w_i \cdot I(y_i = j)$$

where w_i's are the weights assigned to the nearest neighbors.

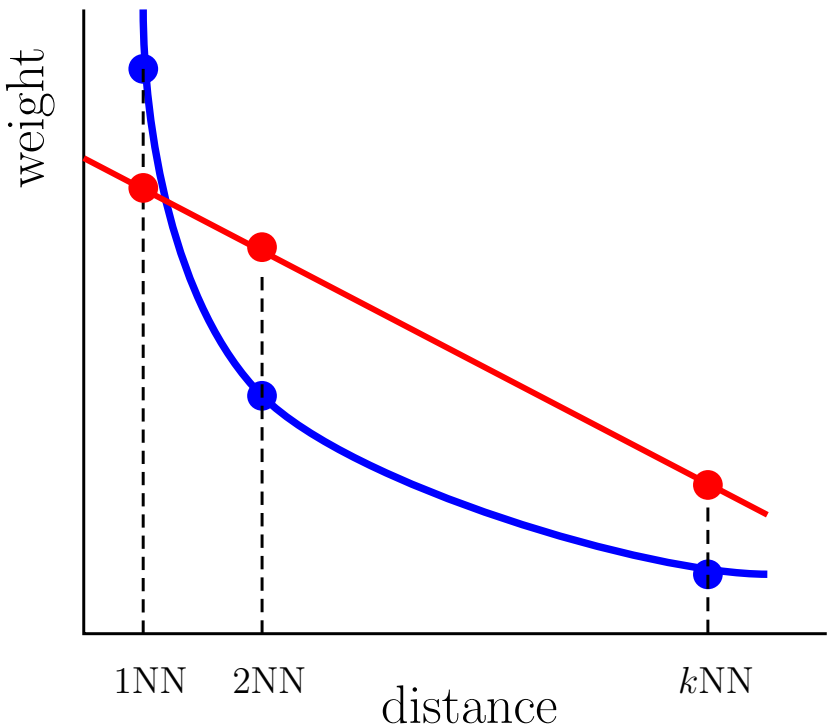


How to choose weights

Generally, the weights w_i should be a decreasing function of the distances from the test point to the nearest neighbors in the training set.

Common weights:

- Linear weights
- (Squared) inverse weights
- Exponentially decaying weights
- Normal weights



Weighted kNN in MATLAB

mdl = fitcknn(trainX, trainLabels, 'DistanceWeight', weight);

weight: A string or a function handle specifying the distance weighting function. The choices of the string are:

- '**equal**': Each neighbor gets equal weight (default).

- '**inverse**': Each neighbor gets weight $1/d$, where d is the distance between this neighbor and the point being classified.

- '**squaredinverse**': Each neighbor gets weight $1/d^2$, where d is the distance between this neighbor and the point being classified.

- **A distance weighting function specified using @.** A distance weighting function must be of the form:

function DW = DISTWGT(D)

taking as argument a matrix D and returning a matrix of distance weight DW. D and DW can only contains non-negative numerical values. DW must have the same size as D. DW(I,J) is the weight computed based on D(I,J).

mdl = fitcknn(trainX, trainLabels, 'DistanceWeight', @DISTWGT);

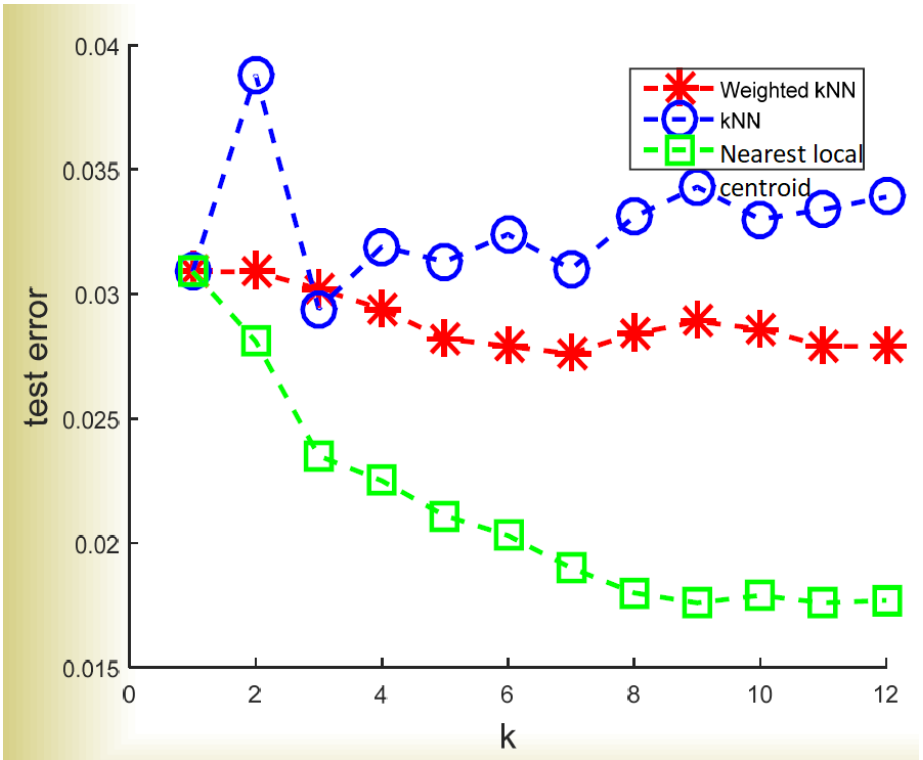
Weighted kNN on MNIST

Recall that the smallest error obtained by a plain kNN classifier on the MNIST handwritten digits is 2.94% (when $k = 3$).

We fix $k = 3$ and experiment with the following kinds of weights:

- Inverse: 2.83%
- Squared inverse: 2.83%
- Linear: 3.02%

Smaller test error rates (with linear weights) may be achieved for larger k; the minimum error rate is 2.76% (when $k = 7$).

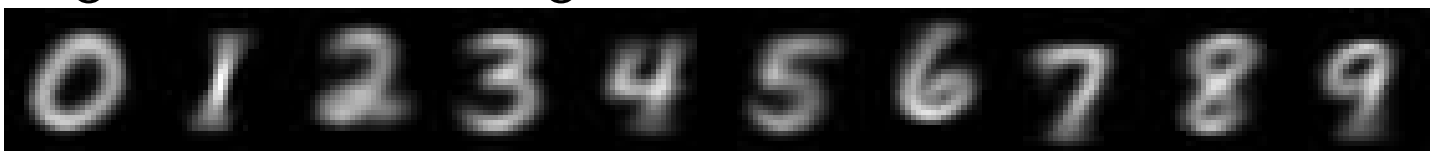


Comments on kNN classification

- Instance-based learning (or lazy learning)
- Model free (nonparametric)
- Simple to implement, but powerful
- Localized and nonlinear (trying to approximate the Bayes classifier)
- Algorithmic complexity only depends nearest neighbors search (memory and CPU cost)
- The choice of k is critical (need cross validation)
- Lots of variations (there could be a final project here)

The nearest centroid classifier

Assign labels to test images based on the most similar centers.



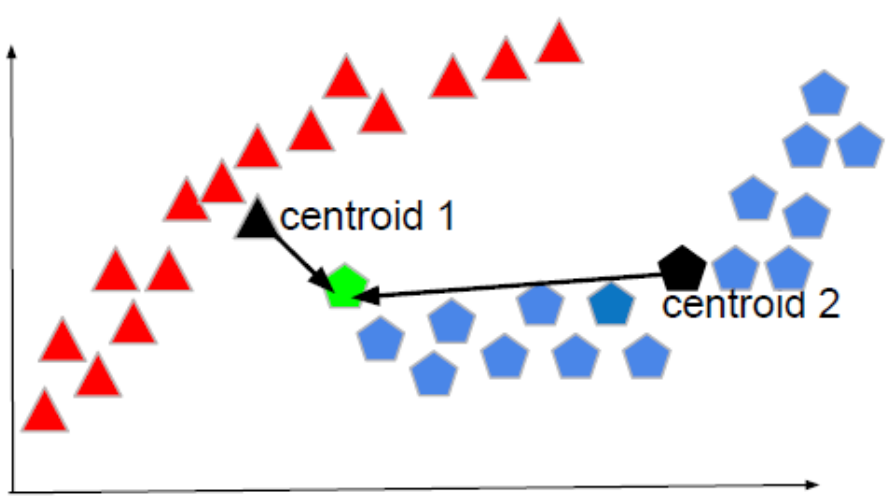
Recall that it has a 18.0% error rate.

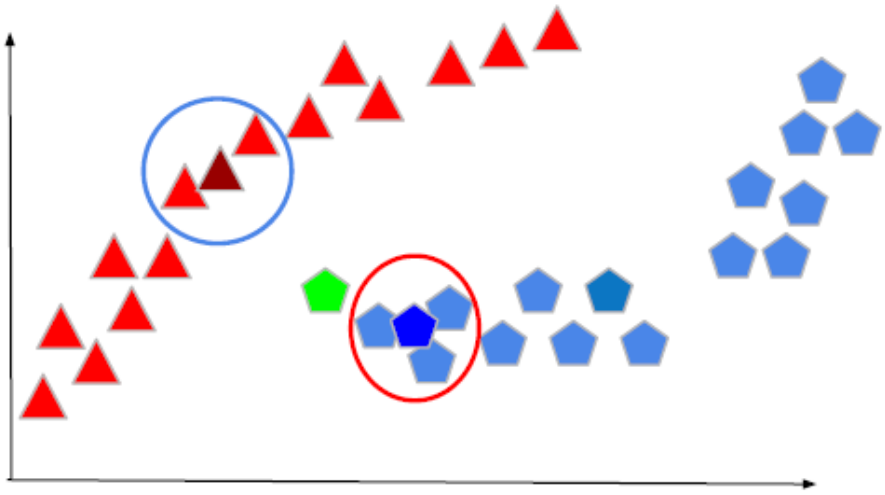
It can also be thought of as an example of a "nearest subset" classifier

$$\hat{y} = \text{argmin}_j \ \text{dist}(\mathbf{x}, \mathcal{C}_j)$$

in which $\text{dist}(\mathbf{x}, \mathcal{C}_j)$ denotes the distance from the test point to the training class.

Why does it not work well?



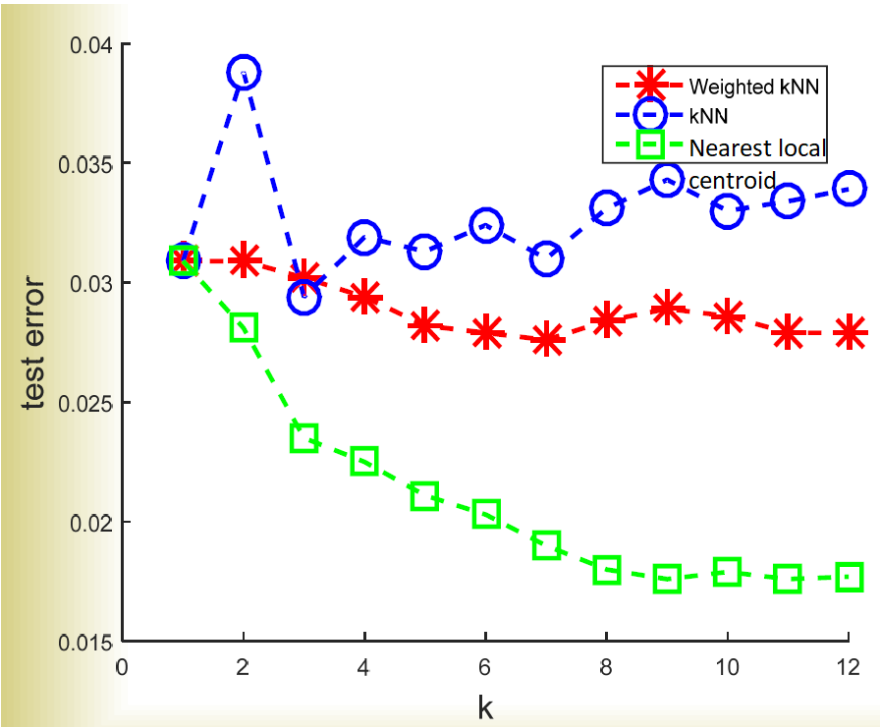


The global centers are often insufficient to summarize the training classes.

A fix is to only look at the most "relevant region" of each class (with respect to the given test point), and use their local centroids to represent the training classes.

Nearest local centroid (NLC) classifier

Main idea: For each test point, find its k nearest neighbors inside each training class and assign the label for the test point based on the nearest local centroid.



The lowest test error rate is 1.76% (when $k = 9$)!

But it may take a long time to perform the nearest local centroid classification.

What is the complexity of classifying one test point?

How to implement the NLC classifier

This classifier is not implemented by any software, so you will have to implement it from scratch.

The following two MATLAB functions (and their counterparts in R/Python) will be very useful to you:

- $pdist2$: Pairwise distances between two sets of observations
- $knnsearch$: k nearest neighbor search

The $pdist2$ function in MATLAB

Pairwise distances between two sets of observations.

- For full usage, type '**help pdist2**' in MATLAB command window.
- **D = pdist2(X,Y);**
  Returns a matrix D containing the Euclidean distances between each pair of observations in the MX-by-N data matrix X and MY-by-N data matrix Y. Rows of X and Y correspond to observations, and columns correspond to variables. D is an MX-by-MY matrix, with the (I,J) entry equal to distance between observation I in X and observation J in Y.

- **D = pdist2(X,Y,DISTANCE)**. Common choices of DISTANCE are:

  'euclidean' - Euclidean distance (default)

  'cityblock' - City Block distance

  'chebychev' - Chebychev distance (maximum coordinate difference)

  'cosine' - One minus the cosine of the included angle between observations (treated as vectors)

  'correlation' - One minus the sample linear correlation between observations (treated as sequences of values).

  'hamming' - Hamming distance, percentage of coordinates that differ. This is the default if all predictors are categorical.

The MATLAB *knnsearch* function

IDX = knnsearch(X,Y);

This is the most basic way of using this function which finds the nearest neighbor in X for each point in Y.

IDX is a column vector. Each row in IDX contains the index of nearest neighbor in X for the corresponding row in Y.

To find more than one nearest neighbor using a metric other than Euclidean, use

IDX = knnsearch(X,Y, 'k', 8, 'Distance', 'cityblock');

For other usage, such as outputting also the distances, type '**help knnsearch**' in MATLAB command window.

Assignment 1 (due 9/19, Wednesday)

Read the following instructions carefully:

- From this homework on, please prepare your homework in presentation format (through Powerpoint or LaTex). Once you are done, convert your file to pdf and submit through Canvas.
- In the first few slides of your presentation you should give a summary of the results from your experiment for each question.
  - You should report the experimental results (such as accuracy and running time) through meaningful visuals such as figures, tables, etc. Please also briefly discuss your results regarding what you learned from them.

  - For each question, you must provide all necessary details of your implementation such as software and parameter values used.

- You must append your scripts at the end of the presentation to support your results (make sure you indicate clearly which script corresponds to which question)
- Lastly, for samples of such homework, you may refer to the following
  - `http://www.math.sjsu.edu/~gchen/Math285S16/HW2-Terry.pdf`
  - `http://www.math.sjsu.edu/~gchen/Math285S16/HW5-Shioushiou.pdf`

Reminder: Some of the programming assignments might take very long time (hours) to finish. Therefore, it is advisable to start doing the homework as early as possible, in order to leave enough time for the experiments to finish.

Questions (Unless specified, distance is default to Euclidean, and weight to none).

1. Download the *iris* data set from `https://archive.ics.uci.edu/ml/datasets/iris` only as the training data and then compute, for each $k = 1, \ldots, 10$, the 5-fold, 10-fold and leave-one-out cross validation errors of the kNN classifier. Plot all the errors against k (as separate curves) in one plot, and also interpret your plot.

2. Download the USPS digits data set from the bottom of `https://web.stanford.edu/~hastie/ElemStatLearn//data.html` and perform kNN classification with each $1 \leq k \leq 10$ and each of the three distance metrics: Euclidean, City-block, and Cosine. Report the test error rates by plotting three curves (one corresponding to a metric) against k. Which distance metric seems to work best on this data set and what is the lowest error rate you got overall?

3. Apply the weighted kNN classifier with the inverse weights, for each $k = 1, \ldots, 10$, to the USPS data set. Repeat the experiment with squared inverse and linear weights. Plot the test errors corresponding to the three different weights against k and interpret the results.

4. Apply the nearest local centroid classifier to the USPS digits data with each $k = 1, \ldots, 10$. Plot the test error curve and interpret your results. What is the confusion matrix corresponding to the optimal k?

5. Find a new data set on the internet (with at least 500 observations) that is suitable for classification, and describe clearly what it is like. Afterwards, explore the data set by trying the classifiers learned in class (with various options and parameter values). What is the best accuracy you can get and for which combination of options and parameter values you got those?