K-Nearest Neighbors (and Evaluating ML Methods)

David I. Inouye

Outline

- K-Nearest Neighbors (KNN) simple algorithm
- Evaluating methods (i.e., generalization error)
  - Train vs test data
  - Cross validation
- Hyperparameter tuning (choosing k)
- Curse of dimensionality

The naïve KNN algorithm requires computing the distance to **all training points**

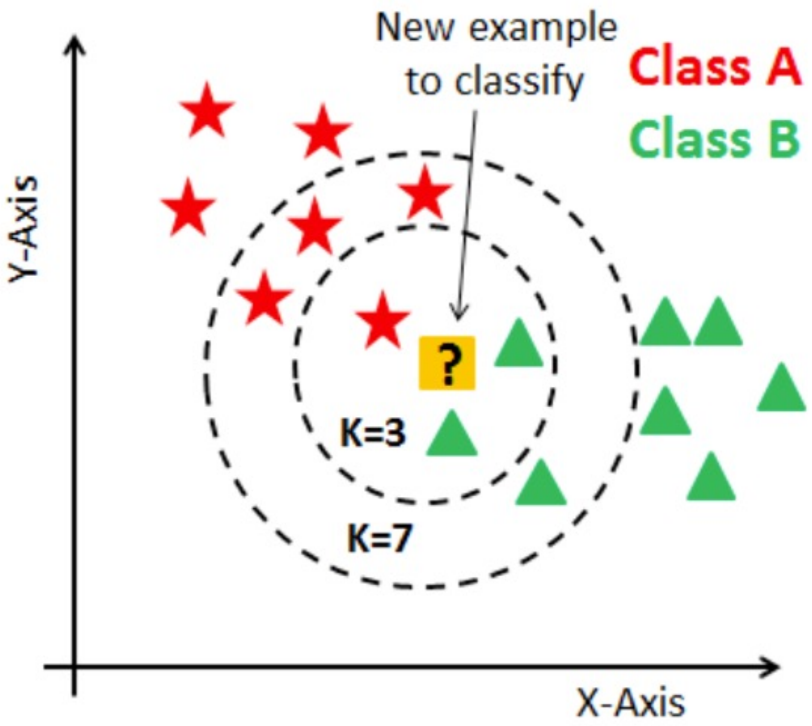
Input: Test point x_0, training data $\{x_i, y_i\}_{i=1}^n$
Output: Predicted class y_0

1. Compute distance to all training points:
$$d_i = d(x_0, x_i), \forall i$$
2. Sort distances where π is a permutation:
(e.g., $\pi(1)$ is the index of the closest point)
$$d_{\pi(1)} \leq d_{\pi(2)} \leq \cdots \leq d_{\pi(n)}$$
3. Return the most common class of the top k
$$y_0 = \text{mode}\left\{y_{\pi(j)}\right\}_{j=1}^k$$

K-nearest neighbors (KNN) is a very simple and intuitive supervised learning algorithm

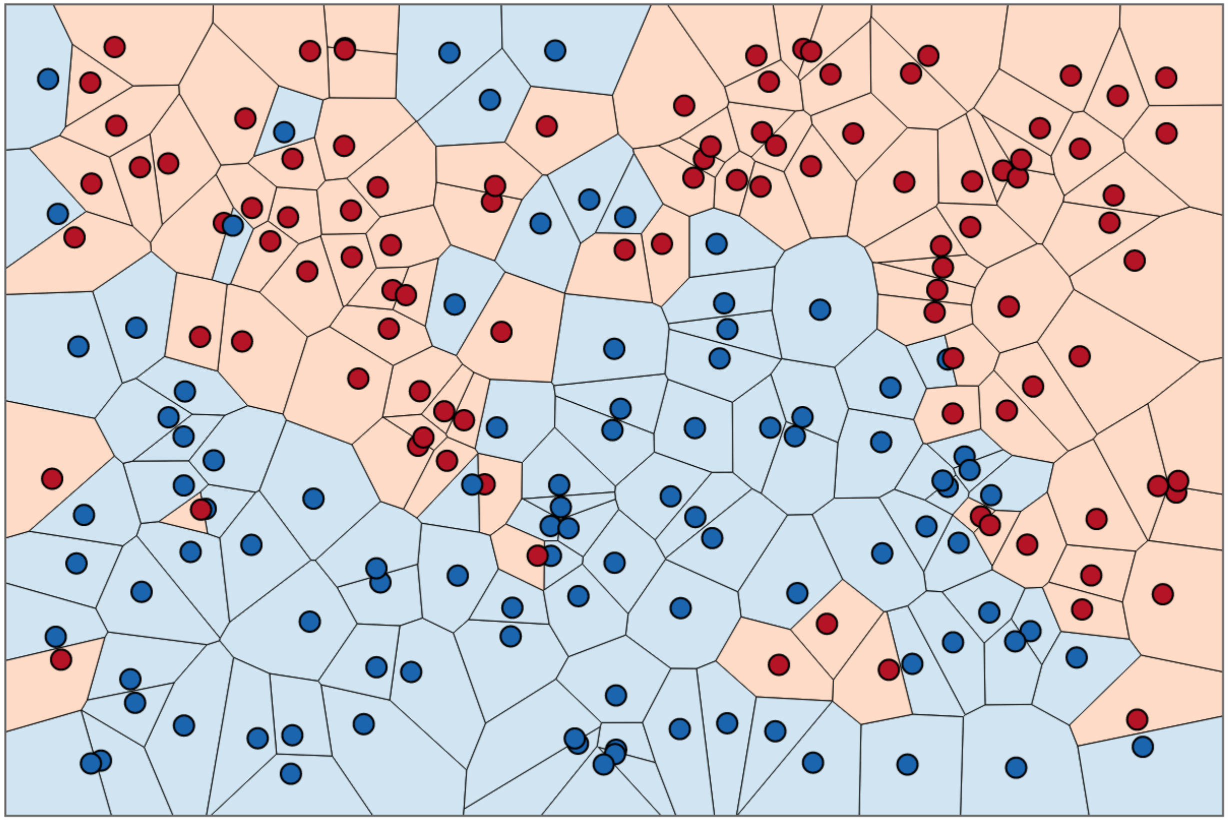
1. Find the k nearest neighbors
   - Equivalently, expand circle until it includes k points

2. Select most common class



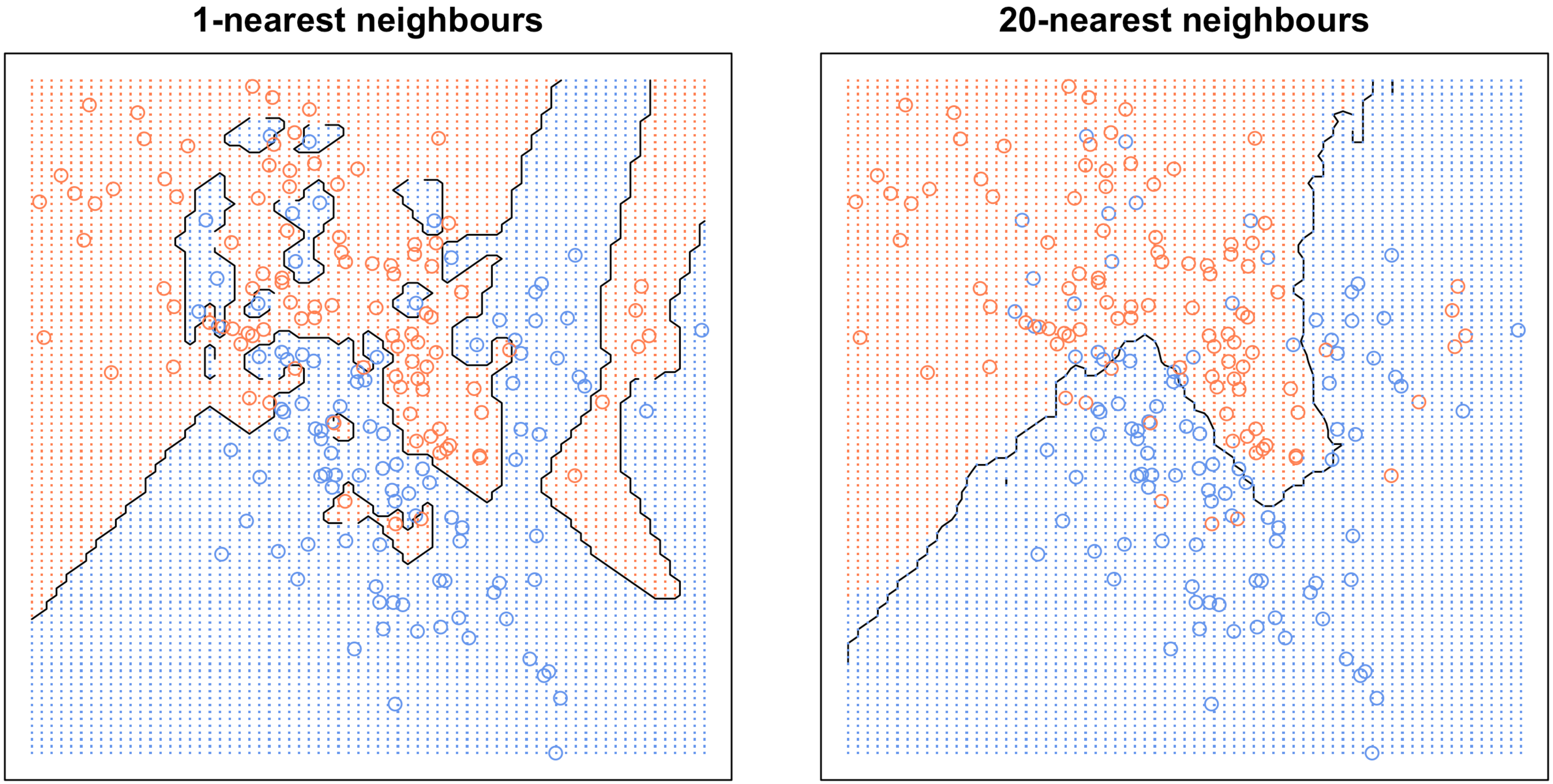
https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn

1-NN partitions the space into Voronoi cells based on the training data



http://scott.fortmann-roe.com/docs/BiasVariance.html

The KNN boundary gets "smoother" as k increases



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

How should method performance be estimated?

- Demo on using KNN with training data

How should method performance be estimated? It should be evaluated on **unseen test data**

- If we train and evaluate on the same data, **the model may not generalize well**.

- Analogy to class
  - **Training data** is like homeworks, sample problems, and sample exams
  - **Testing data** is like the real exam

We actually care about the method's performance on <u>new unseen data</u>

Data we have

What we want

Medical domain

Disease records for past patients

Predict disease for **new patients**

Photos domain

Human-labeled images

Predict object in **new photos**

Business domain

Historical stock prices

Predict **future stock prices**

Estimating **generalization** on unseen data is important for model evaluation and model selection

1. Model evaluation
    - Is the model accurate enough to deploy?
    - Example: The business department may decide that the ML predictions will be worthwhile if the accuracy in the real world is above 90% on average.

2. Model selection
    - Which of many possible models should be used?
    - Example: Which value of k is best for KNN?

<u>Generalization error</u> measures how much error the model makes on **unseen data**

- How do we measure generalization error since (by definition) we don't have new unseen data?

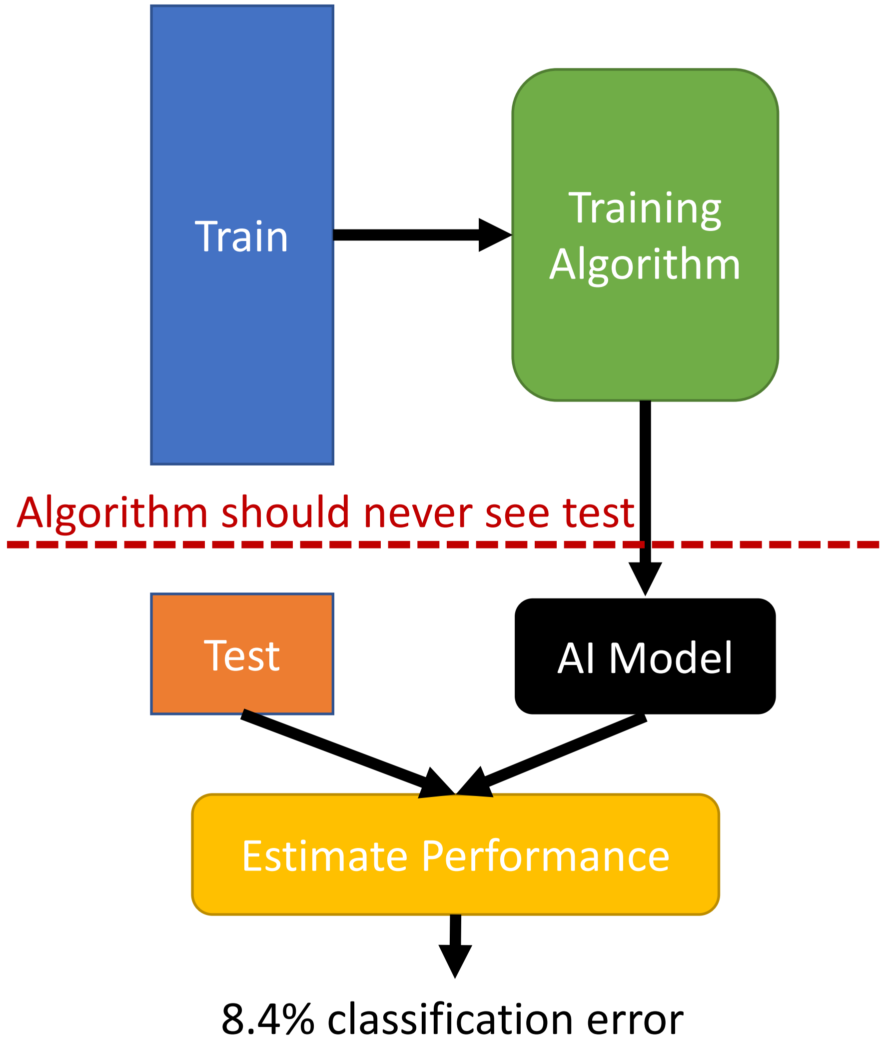
Act like we do! ☺

Generalization error can be estimated by splitting the known dataset

- Split the dataset

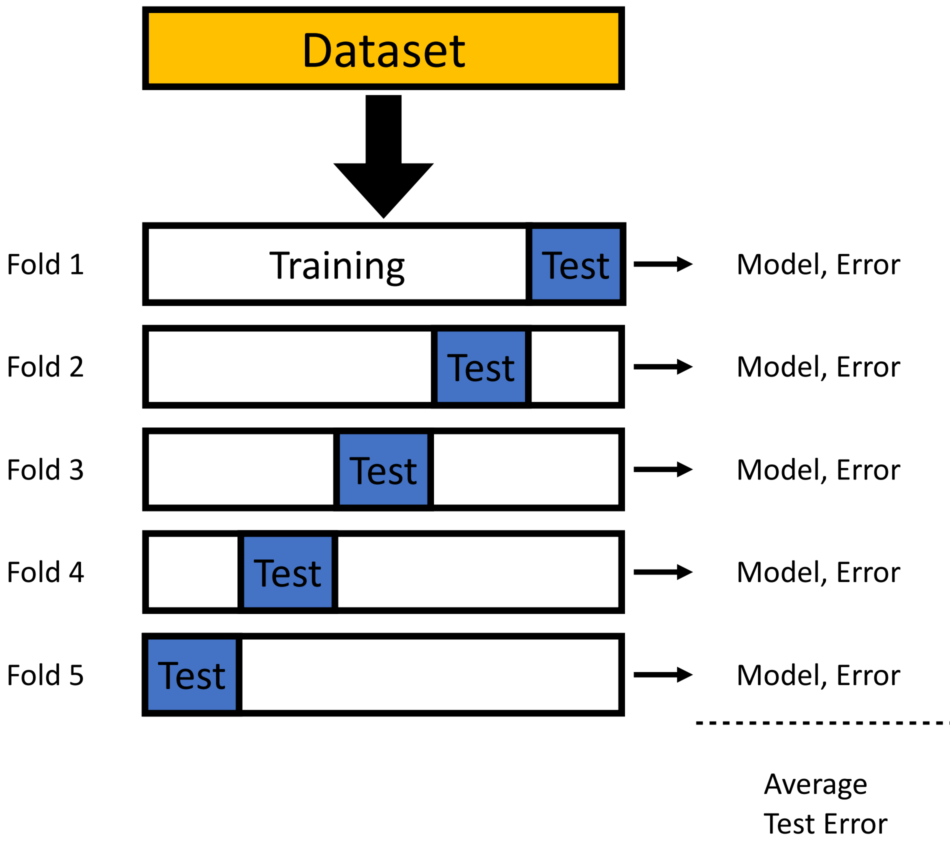
1. The <u>training dataset</u> is used to estimate the model

2. The <u>test dataset</u> (or <u>held-out dataset</u>) is used to estimate generalization error.



Cross-validation (CV) generalizes the simple train/test split to M disjoint splits (effectively reusing data)

- Repeat the split process M times
  - Fit new model on train
  - Evaluate model on test
- Note: M models are fitted throughout process
- Final error estimate is average over all folds

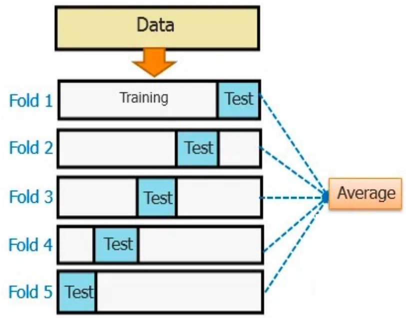


$M = 3, M = 5, M = 10$ are common

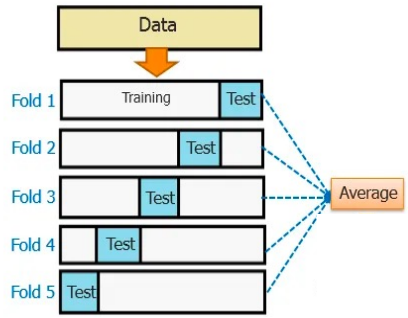
Generalization error via CV can aid in <u>**model selection**</u> (or hyperparameter selection)

(1) Run CV (to estimate generalization) for multiple k

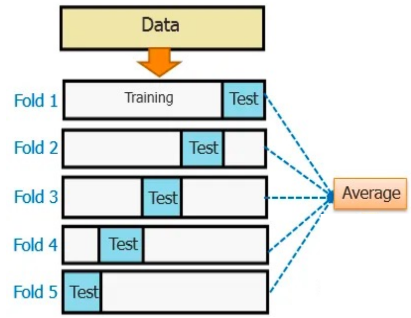
Train all folds
with $k = 1$



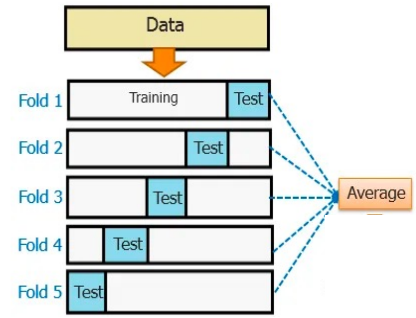
Train all folds
with $k = 3$



Train all folds
with $k = 5$



Train all folds
with $k = 7$



(2) Choose $k^\star$ whose CV performance is the best

$$k^\star = \arg\min_k \ \mathrm{CVGenError}(k; X)$$

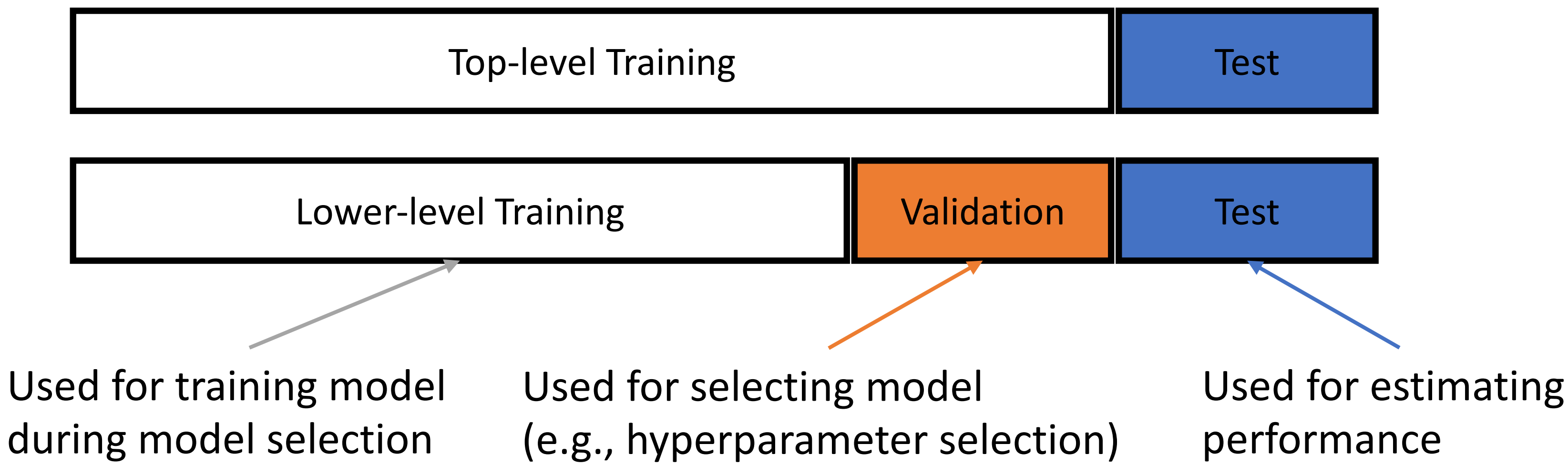
(3) For final model, train model with all data using $k^\star$

Data → Training with $k^\star$ → Final model

Back to demo for using cross validation for KNN

But what if we want to select a model AND estimate the model's performance?

- Inception!
- Nested train/test split (most common)



- Nested CV (better but expensive)

Real-world caveat:
Even CV performance estimates are only good if real-world distribution is like the training data

- Training images in the daytime,
  but real-world images may be at night
  - (Domain generalization tackles this problem)

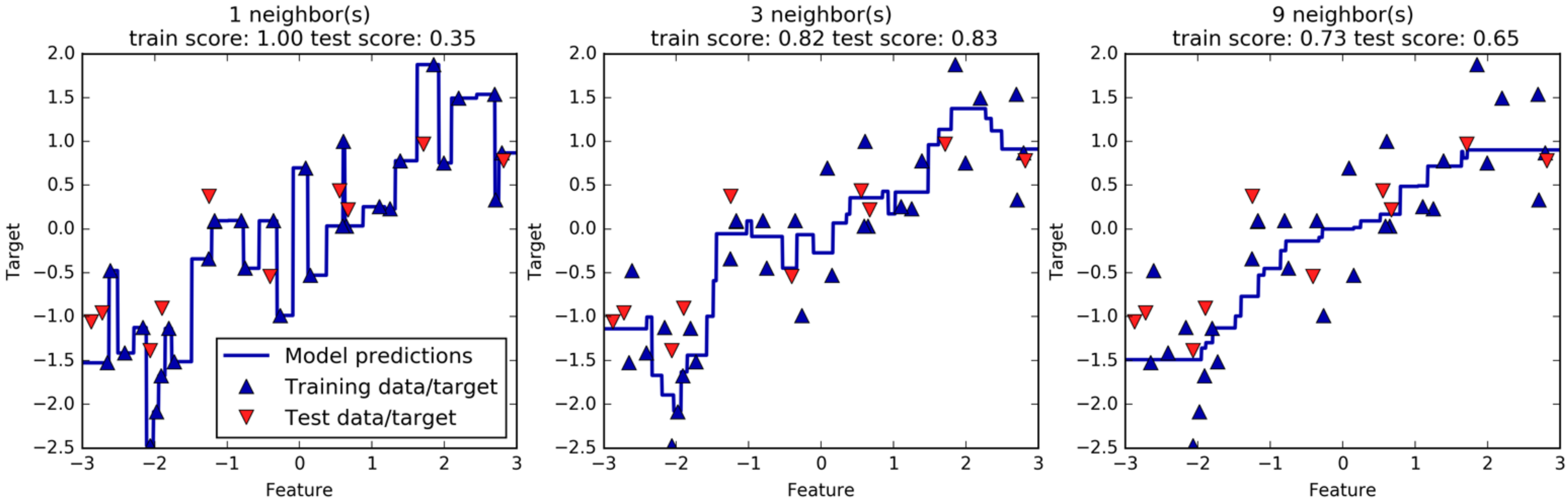
- Training based on historical court cases that are biased against minorities,
  but real-world court cases should be unbiased
  - (Fairness in AI/ML is a recent popular topic)

- Training based on historical stock market data,
  but real-world stock market has changed

Okay, back to KNN… ☺

KNN regression can be used to predict continuous values

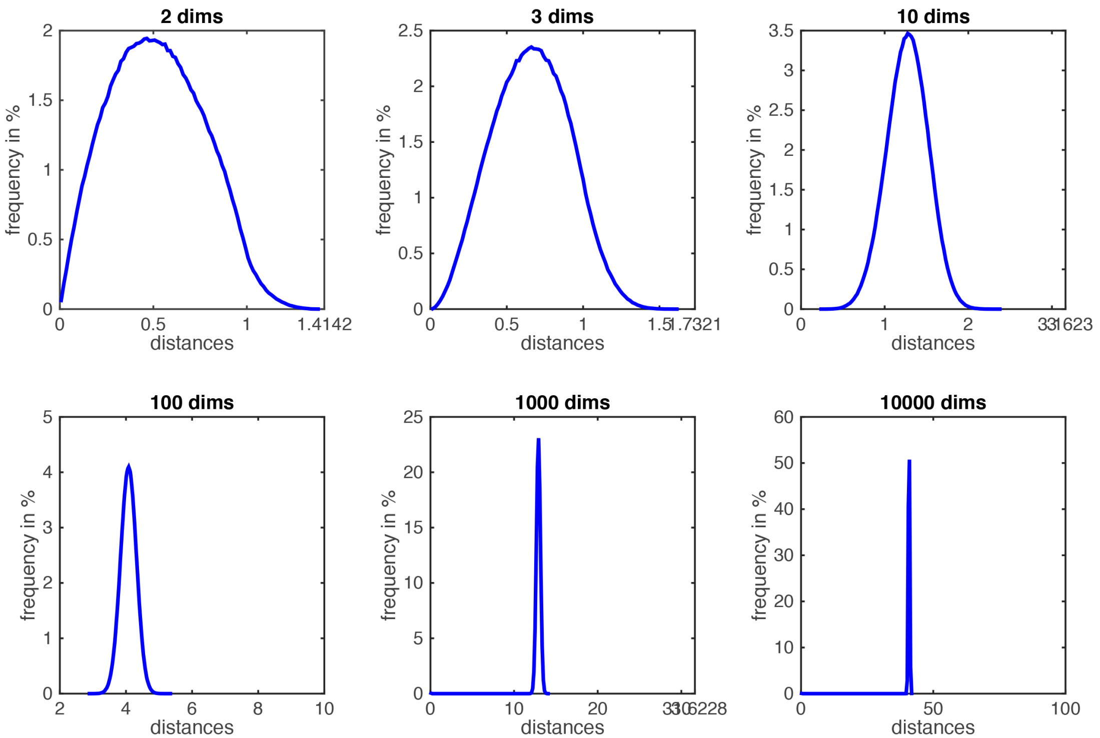
1. Find k nearest neighbors
2. Predict average of k nearest neighbors (possibly weighted by distance)



https://medium.com/analytics-vidhya/k-neighbors-regression-analysis-in-python-61532d56d8e4

The performance and intuitions of KNN degrade significantly in high dimensions (one consequence of the curse of dimensionality)

- The distances between **any two points** in high dimensions is nearly the same

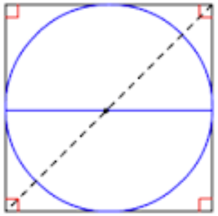


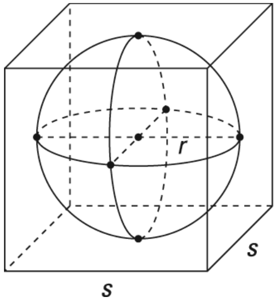
Distance between two **random points** concentrate around a single value

https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html

The curse of dimensionality is *unintuitive*
Example: Most space is in the "corners"

- Ratio between unit hypersphere to unit hypercube
  - 1D : $2/2 = 1$
  - 2D : $\frac{\pi}{4} = 0.7854$
  - 3D : $\frac{\frac{4}{3}\pi}{8} = 0.5238$
  - d-dimensions: $V_d(r) = \frac{\pi^{\frac{n}{2}}}{\Gamma\left(\frac{n}{2}+1\right)} r^d$
    - Thus, for 10-D: 2.55/2^10 = 2.55/1024 = 0.00249

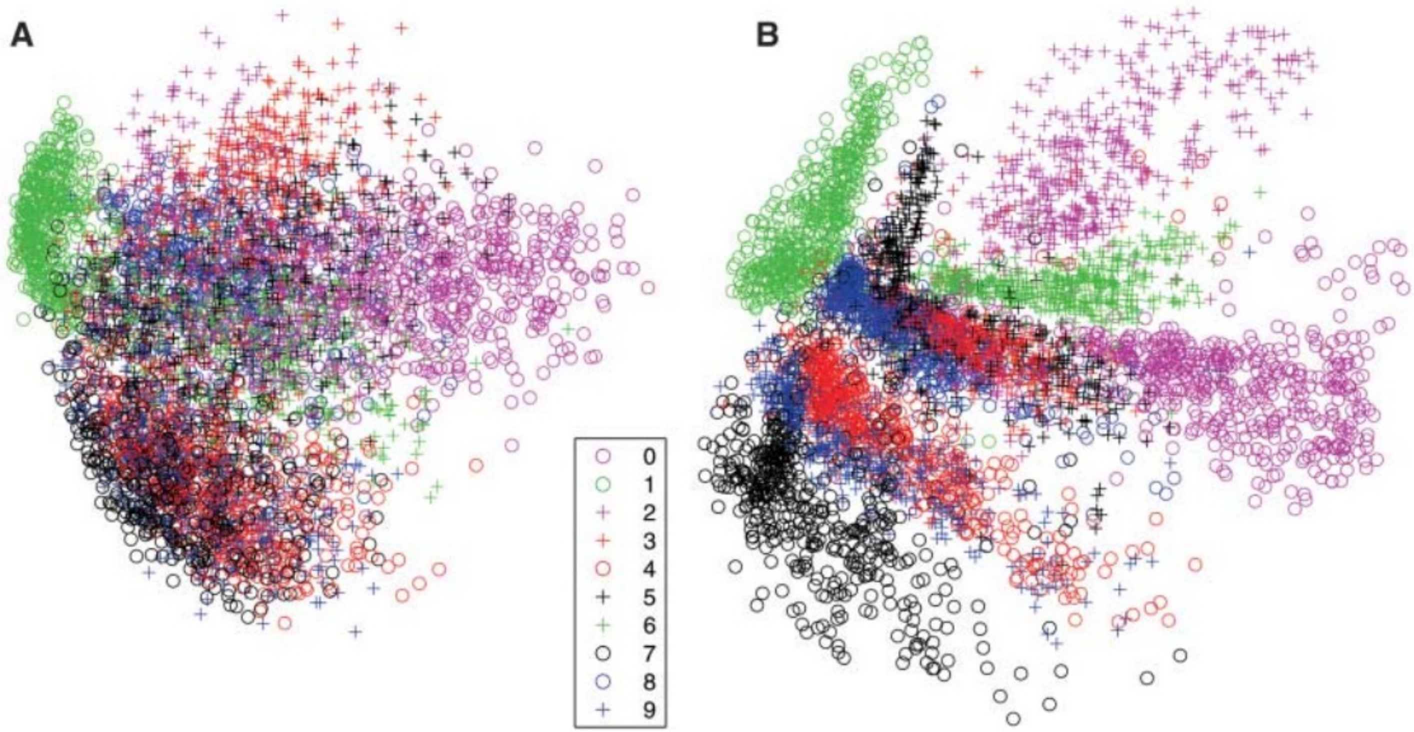




Solution 1:
Reduce the dimensionality and then use KNN

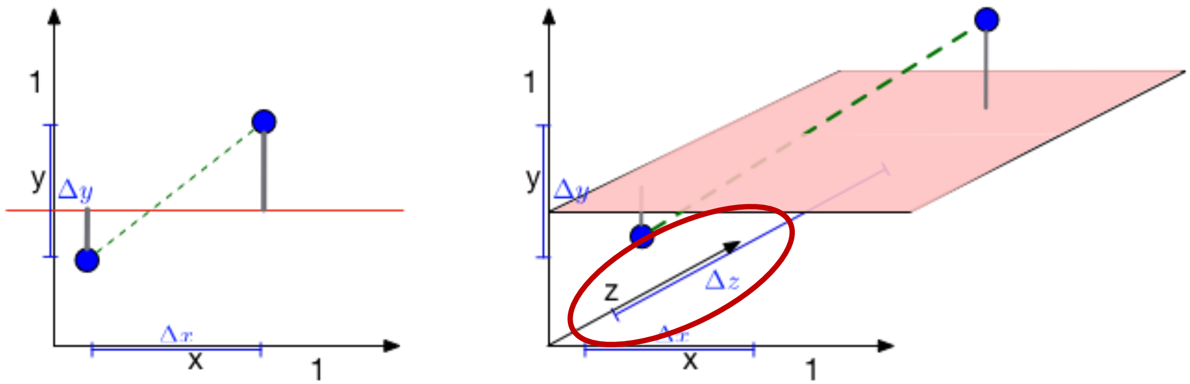
MNIST Digits

Fig. 3. (**A**) The two-dimensional codes for 500 digits of each class produced by taking the first two principal components of all 60,000 training images. (**B**) The two-dimensional codes found by a 784-1000-500-250-2 autoencoder. For an alternative visualization, see (*8*).

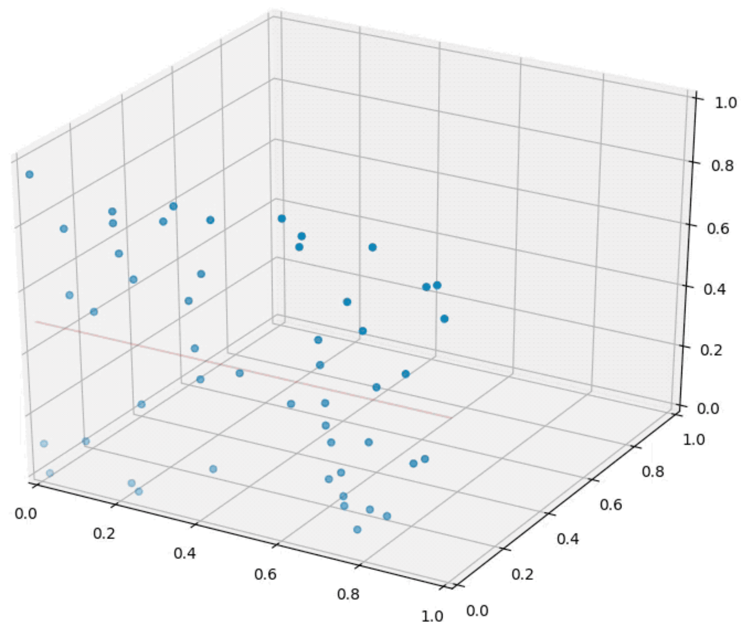


Reducing the Dimensionality of Data with Neural Networks, G. E. Hinton and R. R. Salakhutdinov, Science, 2006, https://www.cs.toronto.edu/~hinton/science.pdf

Solution 2 (non-KNN): Compute distance to hyperplane instead



Distance to hyperplane is **constant** but pairwise distances between points grows as dimensionality increase.



How do we compute distance to hyperplane?

Dot product with unit normal vector plus constant!

$$\boldsymbol{x}^T\boldsymbol{n} + c$$

<u>One view of linear classifiers:</u>
1D projection and then classification

Excellent illustrations from: https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html

Related reading and source for KNN curse of dimensionality illustrations

- https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html