

K-Nearest Neighbors (and Evaluating ML Methods)

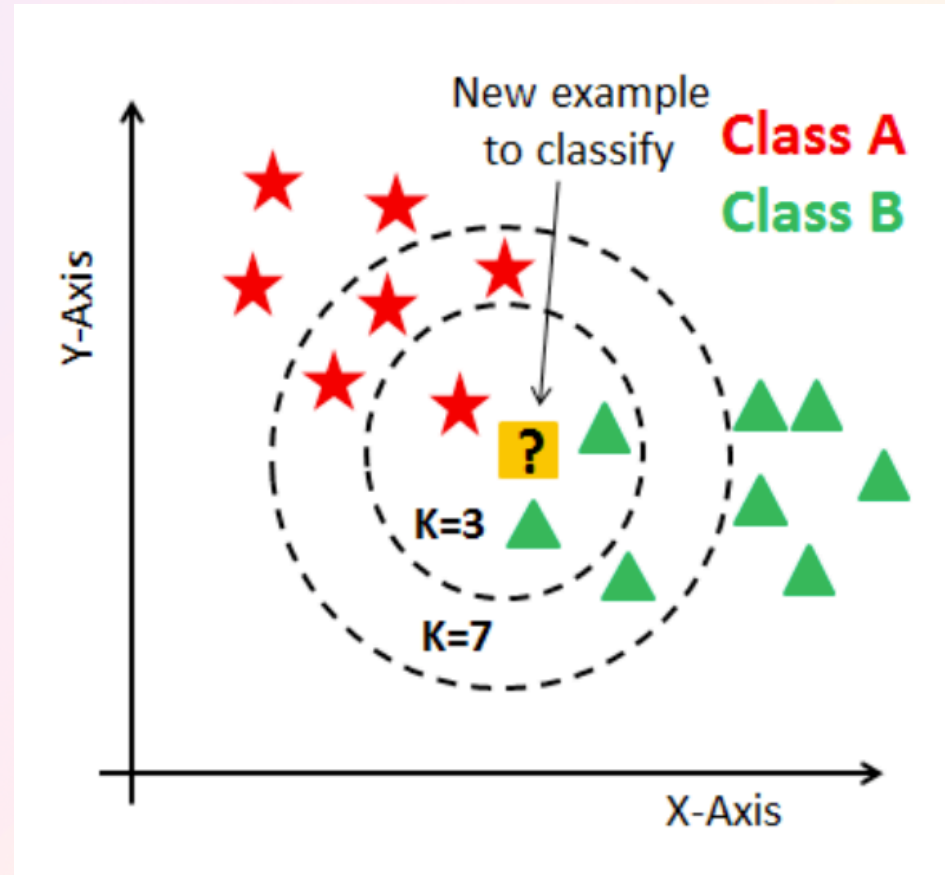
David I. Inouye

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

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



Naïve KNN Requires 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:

$$d_i = d(x_0, x_i), \forall i$$

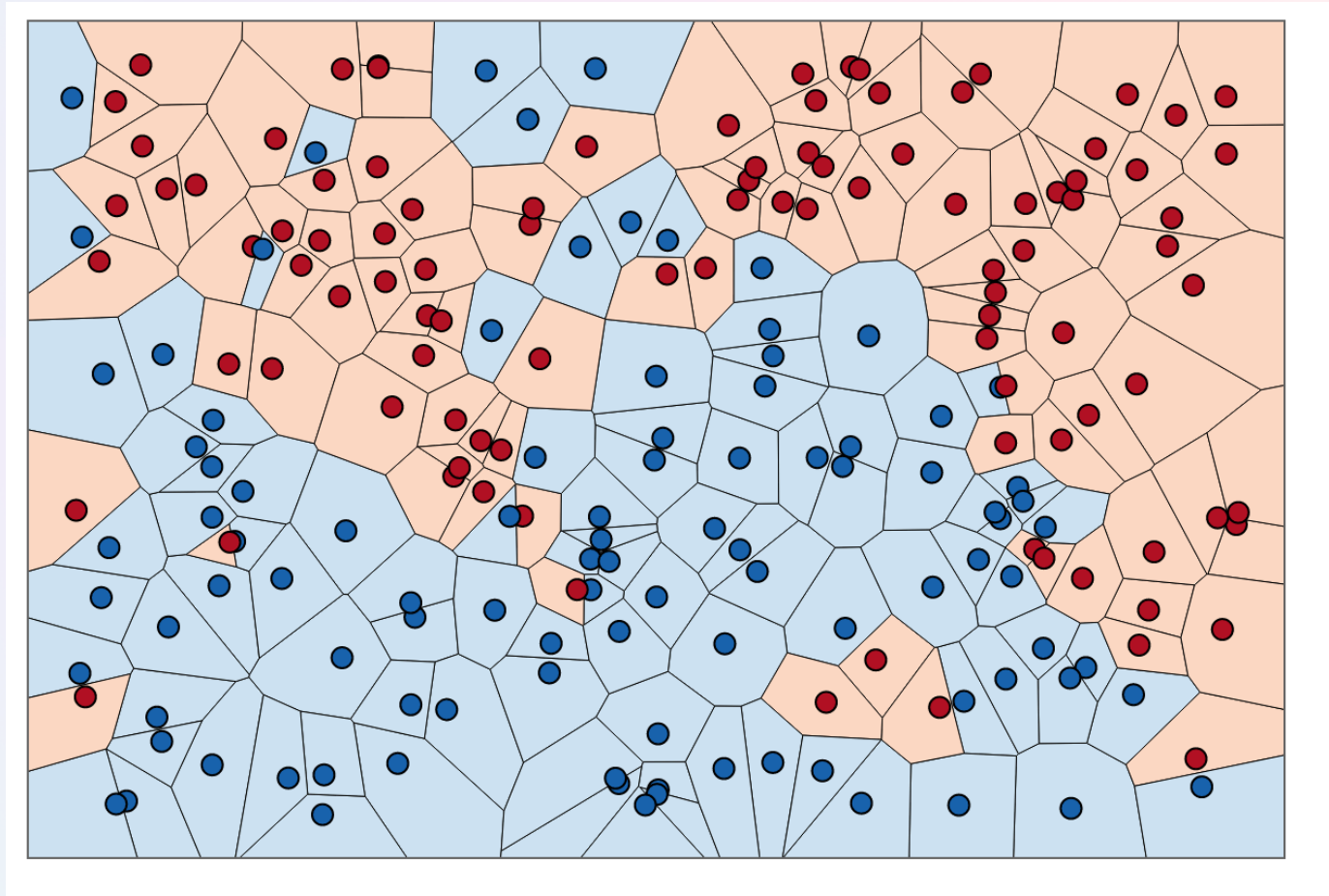
2. Sort distances where π is a permutation (e.g., $\pi(1)$ is the index of the closest point):

$$d_{\pi(1)} \leq d_{\pi(2)} \leq \dots \leq d_{\pi(n)}$$

3. Return the most common class of the top k :

$$y_0 = \text{mode}\{y_{\pi(j)}\}_{j=1}^k$$

1-NN Partitions the Space Into Voronoi Cells Based on the Training Data



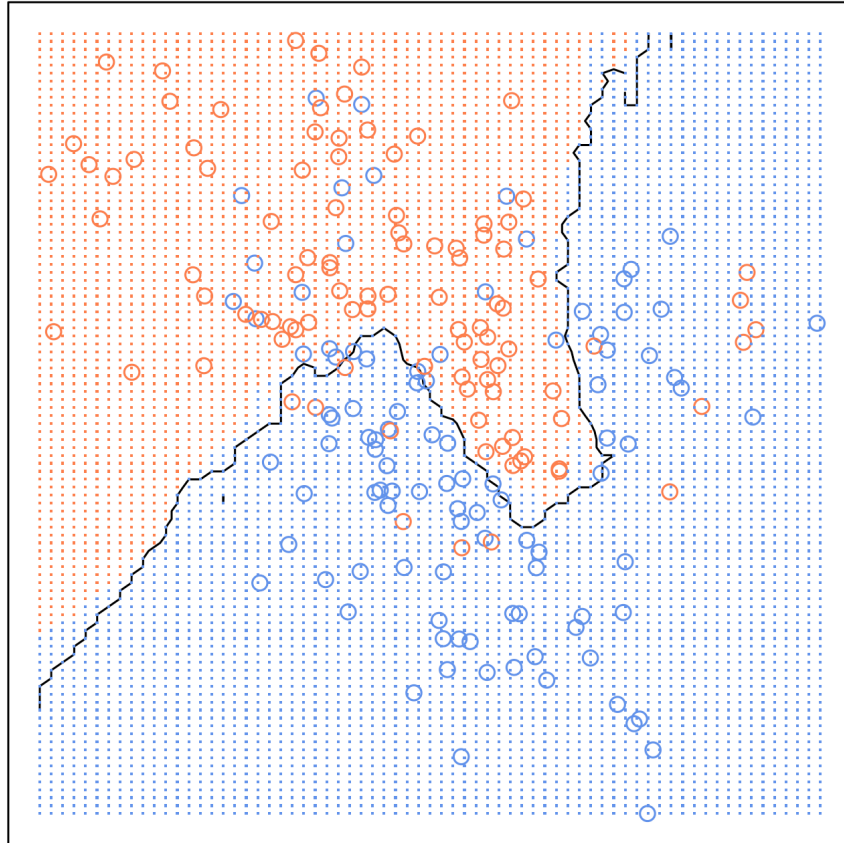
<http://scott.fortmann-roe.com/docs/BiasVariance.html>

The KNN Boundary Gets “Smoother” as k Increases

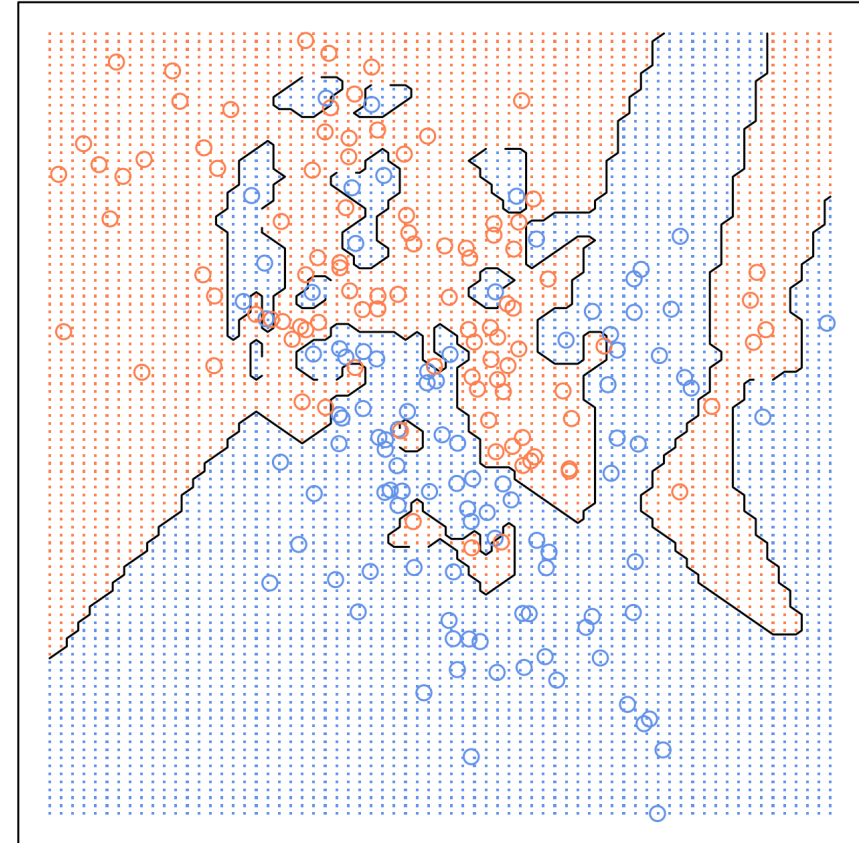
1-nearest neighbours

20-nearest neighbours

20-nearest neighbours



1-nearest neighbours



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

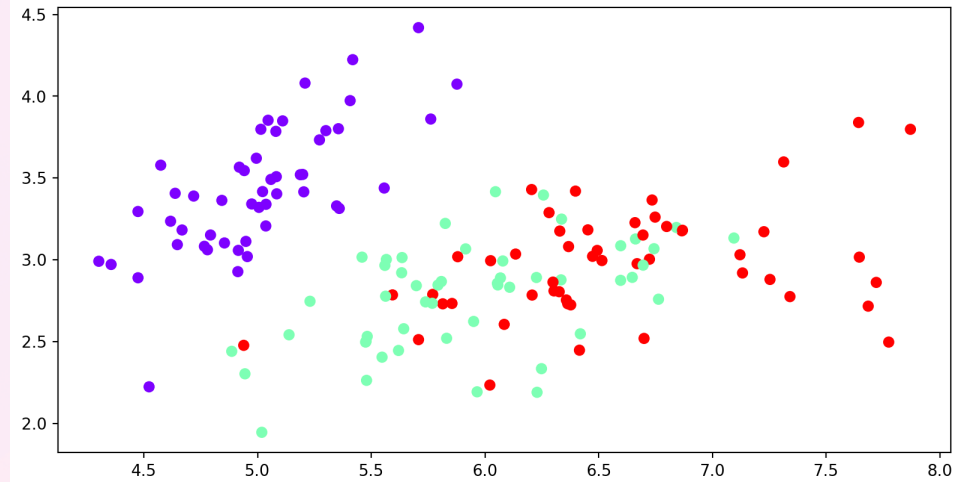
How Should Method Performance Be Estimated?

- We are going to look at a KNN demo to investigate this question.

KNN Demo: Load some toy data

```
1 import numpy as np
2 import scipy.stats
3 import matplotlib.pyplot as plt
4 from sklearn import neighbors, datasets
5 from sklearn.metrics import pairwise_distances
6 from sklearn.utils import shuffle
7
8 # import some data to play with
9 iris = datasets.load_iris()
10 # we only take the first two features
11 X = iris.data[:, :2]
12 # Add random noise since iris has exact values
13 X = X + 0.05 * np.random.RandomState(0).randn(*X.shape)
14 y = iris.target
15 X, y = shuffle(X, y, random_state=0)
16 plt.scatter(X[:, 0], X[:, 1], c=y, cmap='rainbow')
17 print(X.shape)
```

(150, 2)



KNN Demo: Define the KNN Classifier

```
1 class SimpleKNNClassifier():
2     def __init__(self, X_train, y_train, k=1):
3         self.X_train = X_train
4         self.y_train = y_train
5         self.k = k
6
7     def predict(self, X):
8         # 1. Compute distances
9         D = np.nan * np.ones((X.shape[0], self.X_train.shape[0]))
10        for i, x in enumerate(X):
11            for j, xt in enumerate(self.X_train):
12                D[i, j] = np.linalg.norm(x-xt)
13
14        # 2. Get the indices of the top k smallest distances
15        sorted_idx = np.argsort(D, axis=1)
16
17        # For each data point get mode
18        y = np.array([
19            scipy.stats.mode(self.y_train[sidx[:self.k]], keepdims=False)[0]
20            for sidx in sorted_idx
21        ])
22        return y
```

KNN Demo: Define the Vectorized KNN Classifier

```
1 class SimpleKNNClassifier():
2     def __init__(self, X_train, y_train, k=1):
3         self.X_train, self.y_train, self.k = X_train, y_train, k
4
5     def predict(self, X):
6         # 1. Compute distances
7         D = pairwise_distances(X, self.X_train, metric='euclidean')
8
9         # 2. Get the indices of the top k smallest distances
10        sorted_idx = np.argsort(D, axis=1)
11
12        # Faster vectorized version
13        y_ind = self.y_train[sorted_idx[:, :self.k]]
14        y = scipy.stats.mode(y_ind, axis=1, keepdims=False)[0]
15        return y
16
17 for k in [1, 3, 10]:
18     knn = SimpleKNNClassifier(X, y, k=k)
19     y_pred = knn.predict(X)
20     accuracy = np.mean(y == y_pred)
21     print(f'The accuracy on the training data for k={k} is: {accuracy*100:.1f}%')
```

The accuracy on the training data for k=1 is: 100.0%

The accuracy on the training data for k=3 is: 85.3%

The accuracy on the training data for k=10 is: 85.3%

- Given this, what do you think is the best k value? — Discuss with your partner.

KNN Demo: Suppose we only had 100 points for training and then received 50 new flower measurements

```
1 # Use first 100 points
2 X_train = X[:100,:]
3 y_train = y[:100]
4
5 # Setup model
6 for k in [1, 3, 5, 7, 9, 12]:
7     knn = SimpleKNNClassifier(X_train, y_train, k=k)
8
9     # Predict on training data
10    y_pred = knn.predict(X_train)
11    accuracy = np.mean(y_train == y_pred)
12    print(f'The accuracy on the training data for k={k} is: {accuracy}')
13
14    # Now let's test our method on the new flowers
15    X_new = X[100:,:]
16    y_new = y[100:]
17    y_pred = knn.predict(X_new)
18    accuracy = np.mean(y_new == y_pred)
19    print(f'The accuracy on the new data for k={k} is: {accuracy}')
```

The accuracy on the training data for k=1 is: 100.0%
The accuracy on the new data for k=1 is: 72.0%

The accuracy on the training data for k=3 is: 83.0%
The accuracy on the new data for k=3 is: 74.0%

The accuracy on the training data for k=5 is: 83.0%
The accuracy on the new data for k=5 is: 78.0%

The accuracy on the training data for k=7 is: 82.0%
The accuracy on the new data for k=7 is: 80.0%

The accuracy on the training data for k=9 is: 82.0%
The accuracy on the new data for k=9 is: 82.0%

The accuracy on the training data for k=12 is: 83.0%
The accuracy on the new data for k=12 is: 82.0%

- What do you think is the best value of k now?
- The above generalization accuracy estimation algorithm is known as using a train/test split

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 Want the Performance on **New Unseen Data**

Medical Domain

Data we have



What we want



Photos Domain

Data we have



What we want



Business Domain

Data we have



What we want





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?

Generalization Error 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?

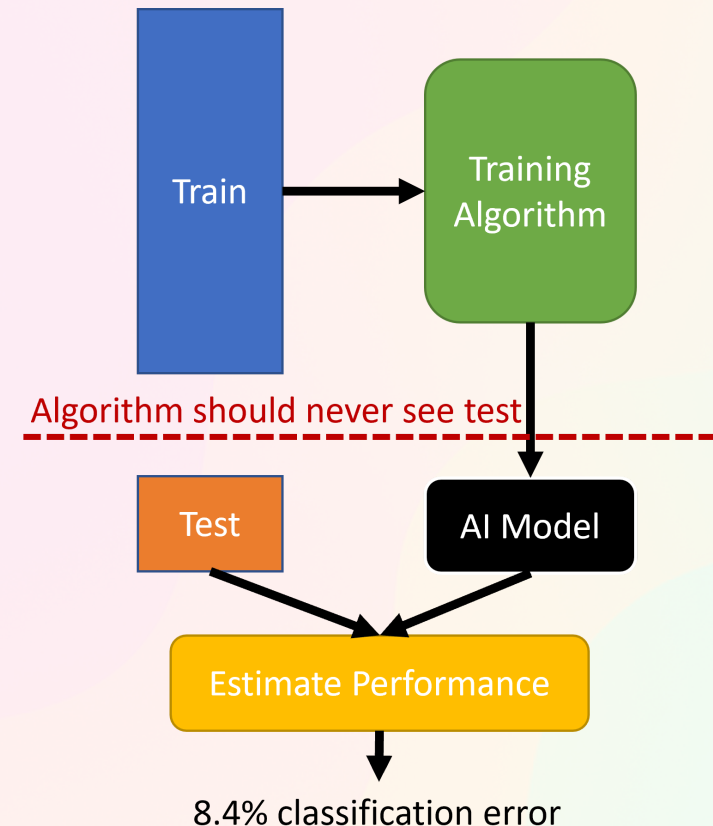
Act like we do!

When you know you're not made for university but you keep trying anyway



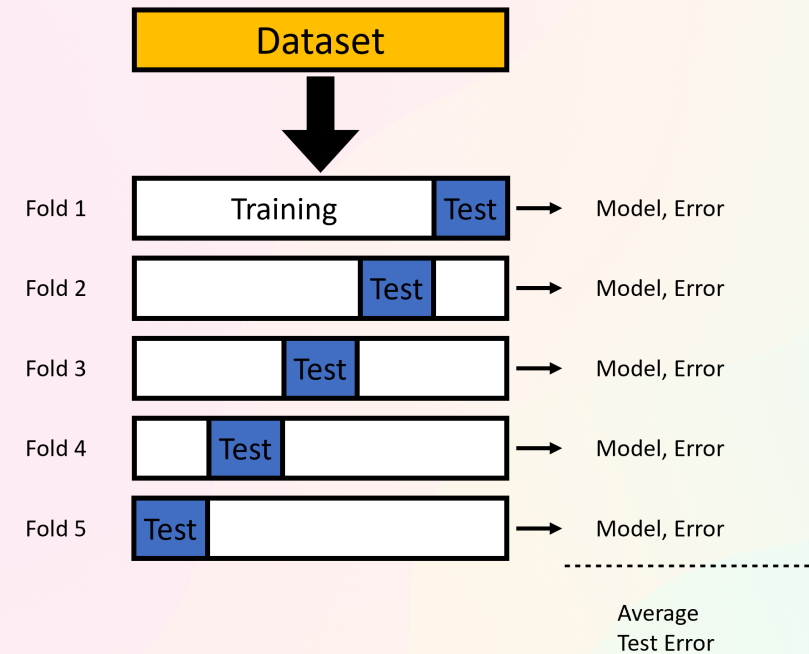
Generalization Error Can Be Estimated by Splitting the Known Dataset

- Split the dataset
 - The *training dataset* is used to estimate the model.
 - The *test dataset* (or *held-out dataset*) is used to estimate generalization error.



Cross-Validation (CV) Generalizes the Simple Train/Test Split to M Disjoint Splits

- 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

KNN Demo: Cross validation is a better estimate of generalization accuracy

```
1 def cv_estimate(X, k, n_splits=3):
2     # Setup split indices and loop over splits
3     split_ind = np.floor(np.linspace(0, X.shape[0], num=n_splits+1))
4     accuracy_list = []
5     for split_start, split_end in zip(split_ind[:-1], split_ind[1:]):
6         # Setup boolean array to select test set
7         test = np.zeros(X.shape[0], dtype=bool) # Initialize false boolean array
8         test[int(split_start):int(split_end)] = True # Set test elements to true
9
10        # Create train and test sets (~ is used to flip boolean values)
11        X_train, y_train = X[~test, :], y[~test]
12        X_test, y_test = X[test, :], y[test]
13
14        # Train model for this split using X_train and y_train
15        knn = SimpleKNNClassifier(X_train, y_train, k=k)
16
17        # Compute accuracy on test split
18        y_pred = knn.predict(X_test)
19        accuracy = np.mean(y_test == y_pred)
20        accuracy_list.append(accuracy)
21    return np.mean(accuracy_list) # Take mean of accuracy
```

KNN Demo: Cross validation is a better estimate of generalization accuracy

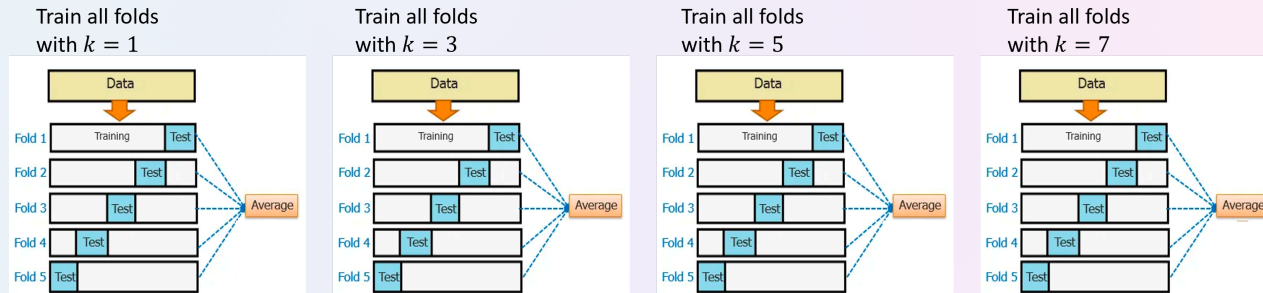
```
1 for k in [1, 3, 5, 7, 9, 12]:  
2     cv_acc = cv_estimate(X, k, n_splits=3)  
3     print(f'CV accuracy estimate for k={k} is {100*cv_acc:.1f}')
```

```
CV accuracy estimate for k=1 is 74.7%  
CV accuracy estimate for k=3 is 76.0%  
CV accuracy estimate for k=5 is 76.7%  
CV accuracy estimate for k=7 is 78.0%  
CV accuracy estimate for k=9 is 77.3%  
CV accuracy estimate for k=12 is 76.0%
```

- What do you think is the best value of k using CV and why? Is it the same as a single train/test split?

CV Generalization Error Can Aid in Model Selection

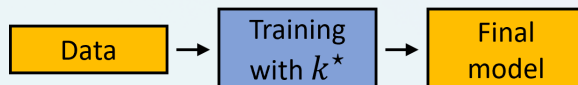
1. Run CV (to estimate generalization) for multiple k .



2. Choose k^* whose CV performance is the best.

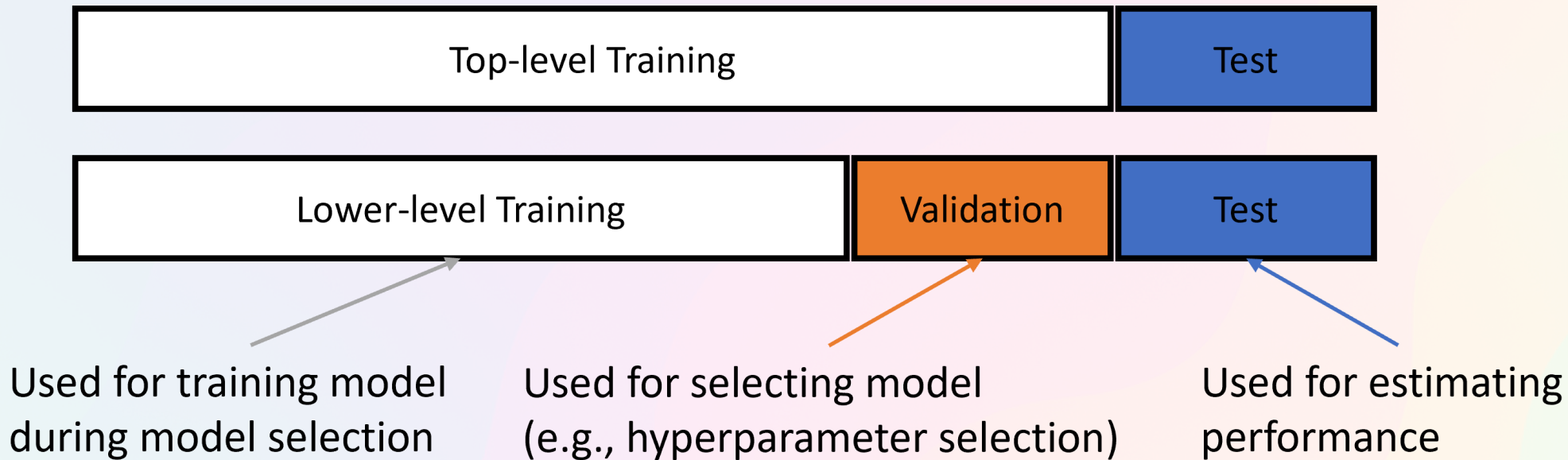
$$k^* = \arg \min_k \text{CVGenError}(k; X)$$

3. For final model, train model with all data using k^*



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)**

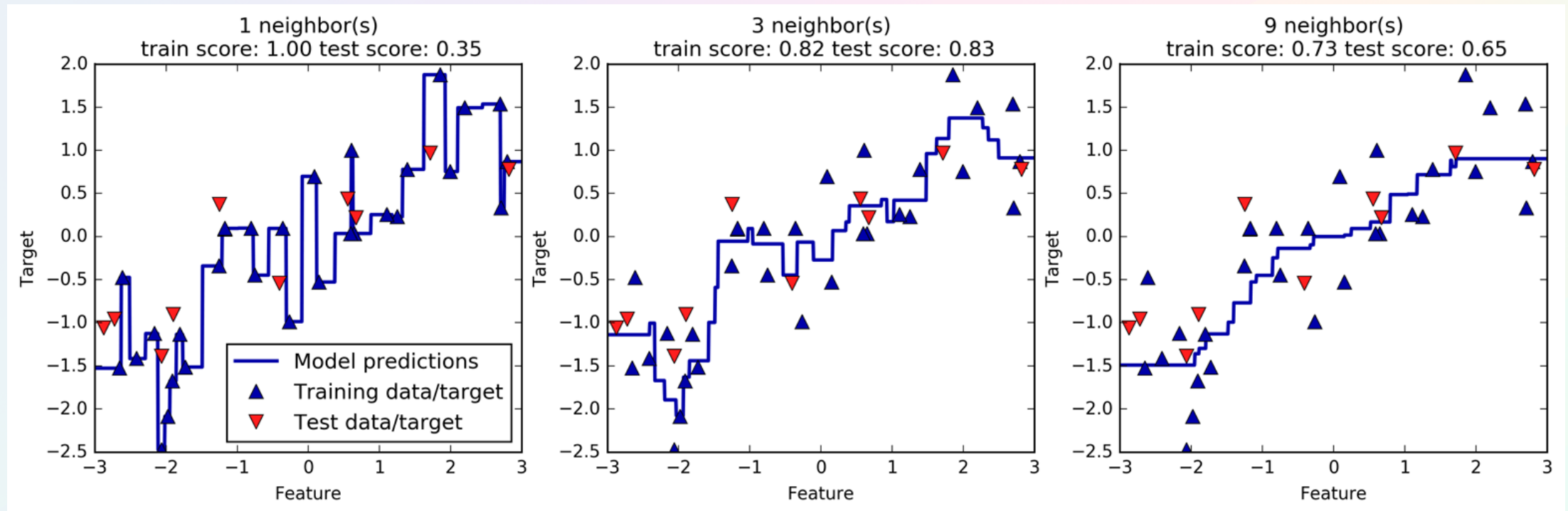
Real-World Caveat: Even CV Performance Estimates Are Only Good if **Real-World Distribution** is Like the Training Data

- 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)

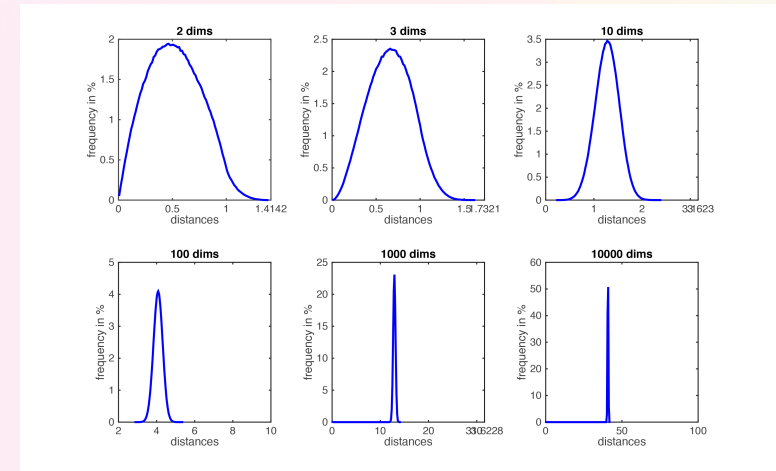


<https://medium.com/analytics-vidhya/k-neighbors-regression-analysis-in-python-61532d56d8e4>

The Performance and Intuitions of KNN Degrade Significantly in High Dimensions

(one consequence of the **curse of dimensionality**)

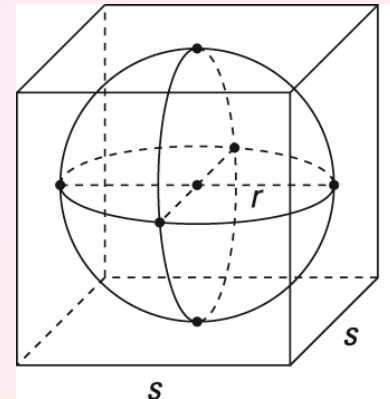
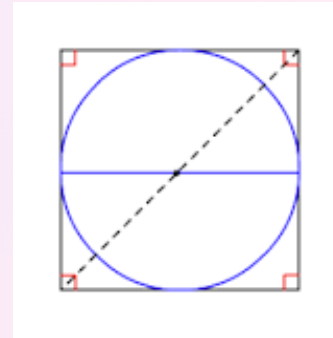
- The distances between *any two points* in high dimensions is nearly the same.



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

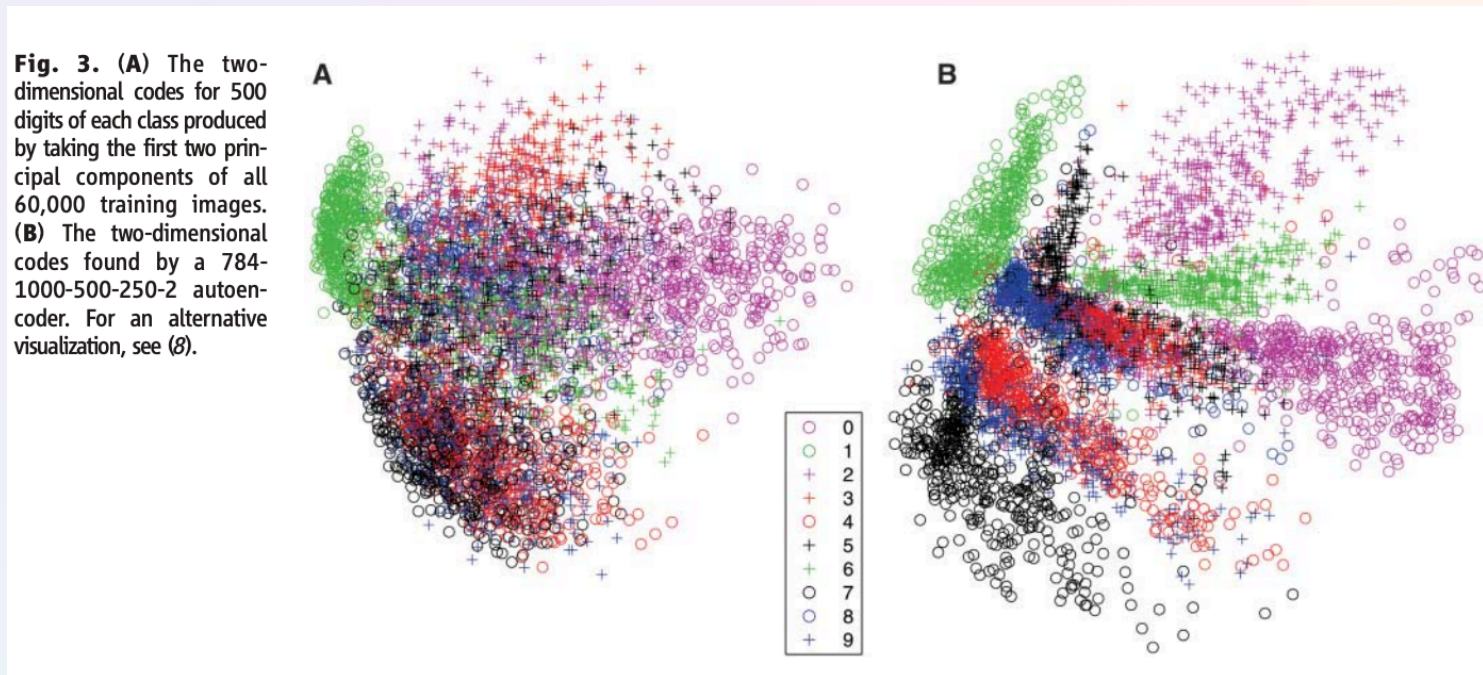
The **Curse of Dimensionality** is Unintuitive: Most Space is in the “Corners”

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



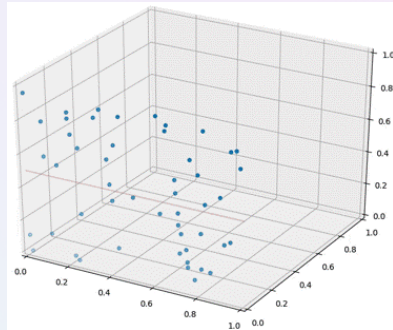
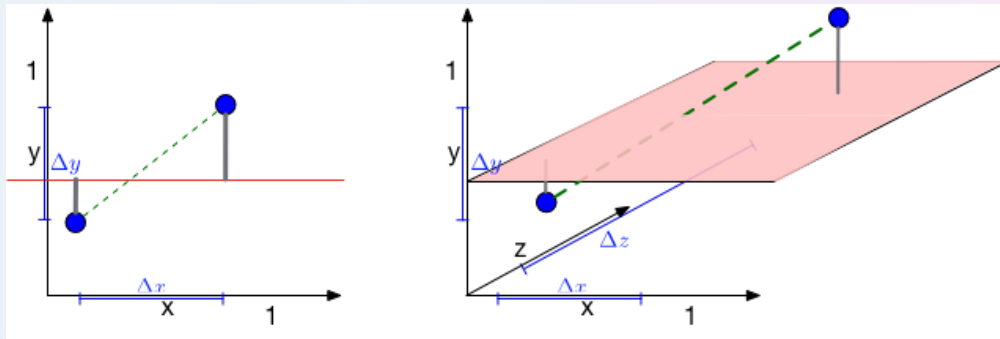
Solution 1: Reduce the Dimensionality and Then Use KNN

MNIST Digits



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 increases.

How do we compute distance to hyperplane?

- Dot product with unit normal vector plus constant!

$$x^T n + c$$

- *One view of linear classifiers*: 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