# MY474: Applied Machine Learning for Social Science

## Lecture 4: Gradient Descent, Bootstrap, Cross-Validation, Hyperparameters

Blake Miller

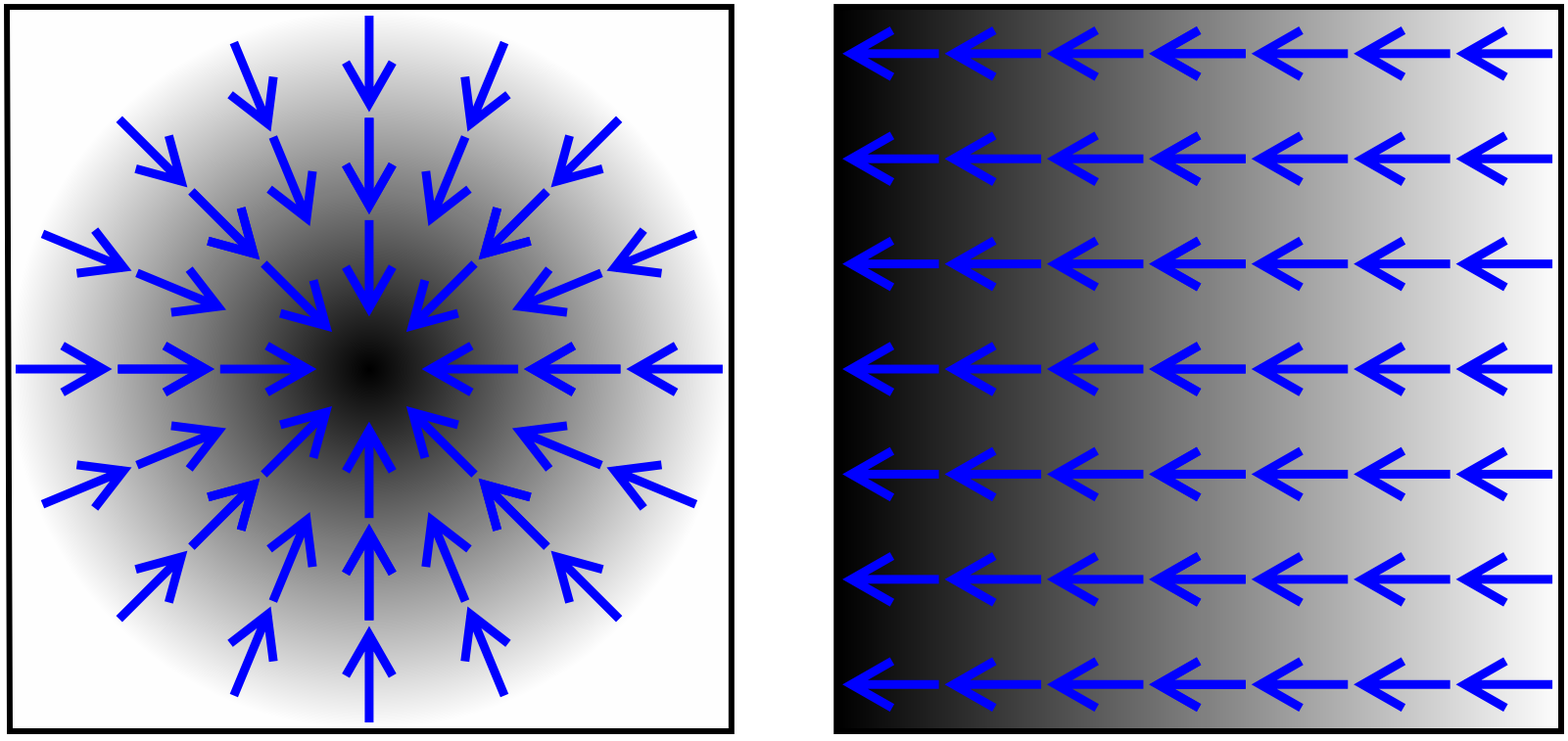
29 January 2021

# Agenda

1. Gradient Descent
2. The Bootstrap
3. Cross-Validation
4. Hyperparameter optimization
   - Grid search
   - Random search
   - Bayesian search

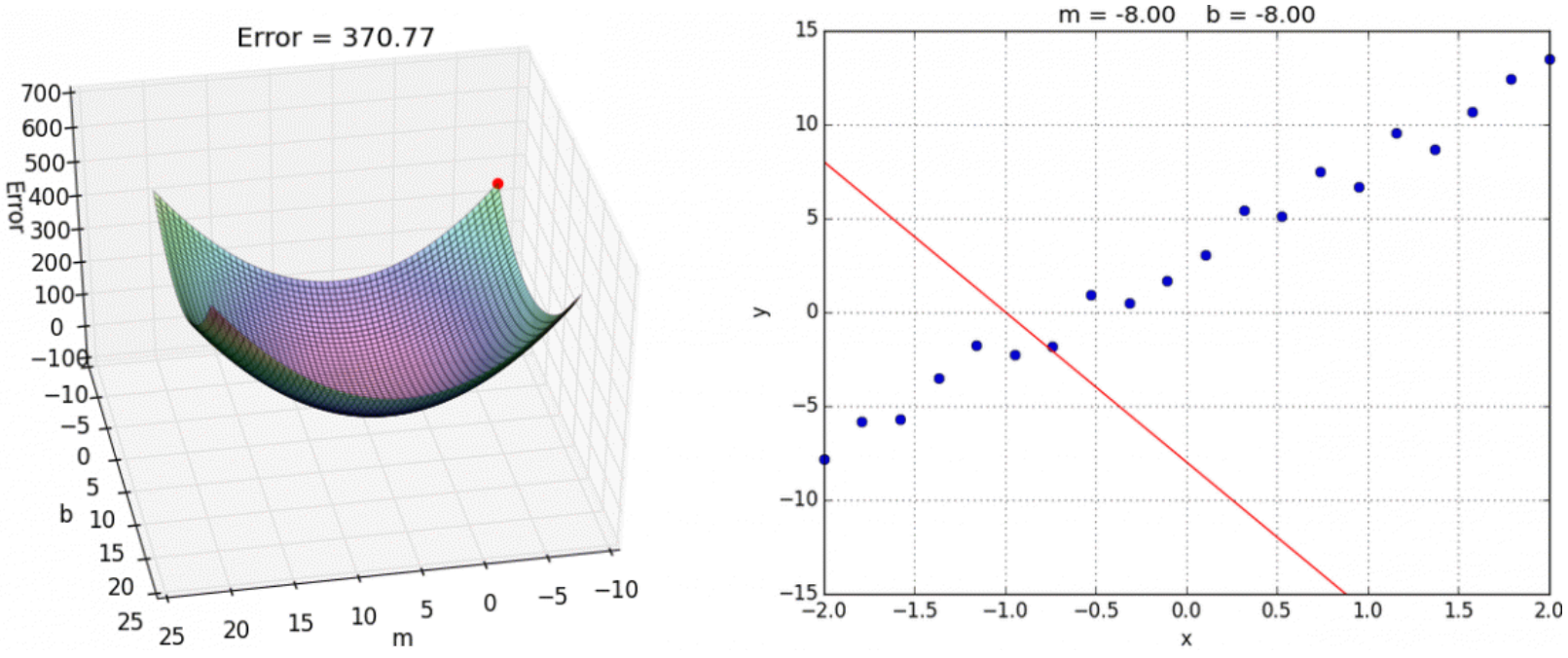
# Gradient Descent

# Gradient Descent



The gradient, represented by the blue arrows, denote the direction of greatest change of a scalar function. The values of the function are represented in greyscale and increase in value from white (low) to dark (high). Source: Wikimedia Commons

# Gradient Descent visualized with Simple Linear Regression




Source: Alykhan Tejani

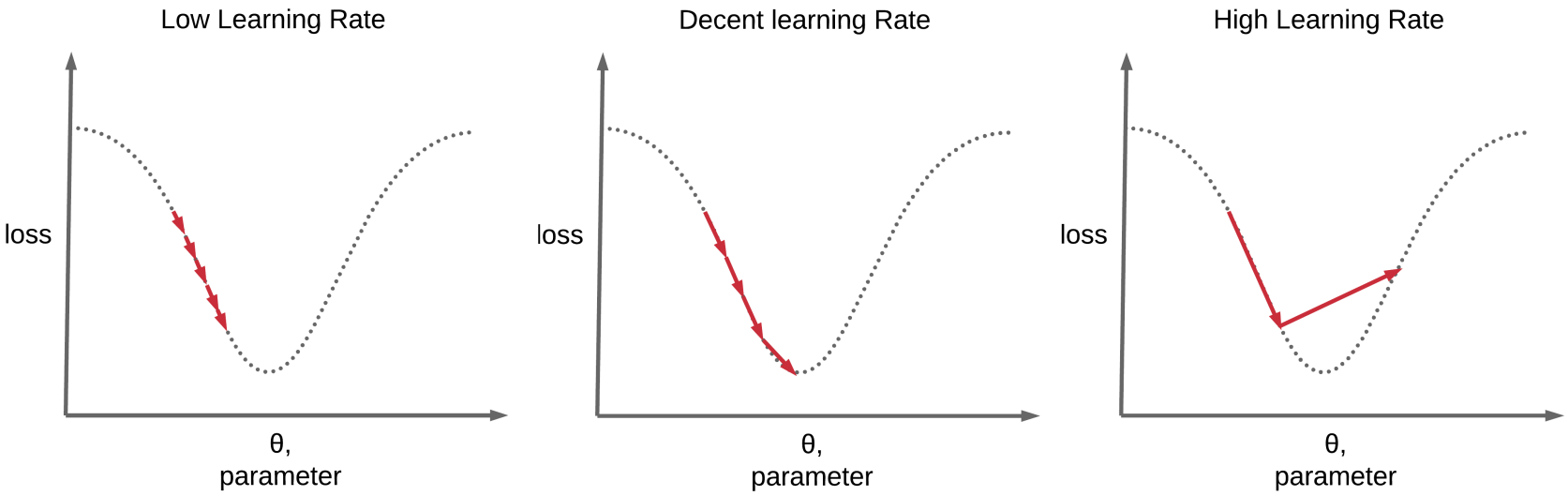
# Gradient Descent

Our goal is to estimate $\beta$ by minimizing the loss function $-ln(\ell(\beta; y_i, \mathbf{x_i}))$:

$$\hat{\beta} = \arg\min_{\beta} - ln(\ell(\beta))$$

- A **loss function** measures how far away predictions $\hat{y}$ are from the true values $y$.
- The **gradient** $\nabla f(\cdot)$ is the vector of parital derivates where the direction represents the greatest rate of increase of the function.
- For a convex function like a loss function, moving in the negative direction of the gradient will take you toward the minimum of that function.
- In other words, the gradient of the loss gives the direction of the prediction error and we can move in the opposite direction to minimize that error.
- How much we move is a parameter $\lambda$ called the learning rate

# Learning Rate




- $\lambda$ too small: will approach the minimum very slowly
- $\lambda$ too big: will jump around the minimum and converge very slowly (if at all)
- We want to choose a $\lambda$ that minimizes the time to convergence and gives us the right answer.

# (Batch) Gradient Descent

**Algorithm 1:** (Batch) Gradient Descent

Set tolerance parameter $\epsilon$ to positive real value;
Set learning rate parameter $\lambda$ to positive real value;
Initialize $\hat{\boldsymbol{\beta}}$ to a random non-zero vector;
**while** $\|\nabla \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})\| > \epsilon$ **do**
    $\hat{\boldsymbol{\beta}} \leftarrow \hat{\boldsymbol{\beta}} - \lambda \nabla \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$;
**end**

# Stochastic Gradient Descent

- With large training data, gradient descent can become very computationally expensive
- Must sum over all examples of the training data at each step in the iteration

$$ln(\ell(\beta; y_i, \mathbf{x_i})) = \sum_{i=1}^{n} \left[ y_i \mathbf{x_i}' \beta - log(1 + e^{\mathbf{x_i}' \beta}) \right]$$

- Instead of considering the full batch gradient (with all training data), we compute a stochastic version of the gradient with a single randomly-selected training observation.
- Very efficient in practice, cuts down training time significantly while performing similarly (and sometimes better) than batch gradient descent.

# Stochastic Gradient Descent

**Algorithm 2:** Stochastic Gradient Descent

Set learning rate parameter $\lambda$ to positive real value;
Initialize $\hat{\beta}$ to a random non-zero vector;
**for** $e \leftarrow 0$ **to** $E$ **do**
    Shuffle $\mathcal{D}$ to prevent cycles;
    **foreach** $(x_i, y_i) \in \mathcal{D}$ **do**
        Compute $\hat{y} \leftarrow h(x_i)$;
        Compute the loss $\mathcal{L}_i(\hat{y}_i, y_i)$;
        Compute the gradient of the loss $\nabla \mathcal{L}_i(\hat{y}_i, y_i)$;
        Update parameters $\hat{\beta} \leftarrow \hat{\beta} - \lambda \nabla \mathcal{L}_i(\hat{y}_i, y_i)$;
    **end**
**end**

# Cross-Validation and the Bootstrap

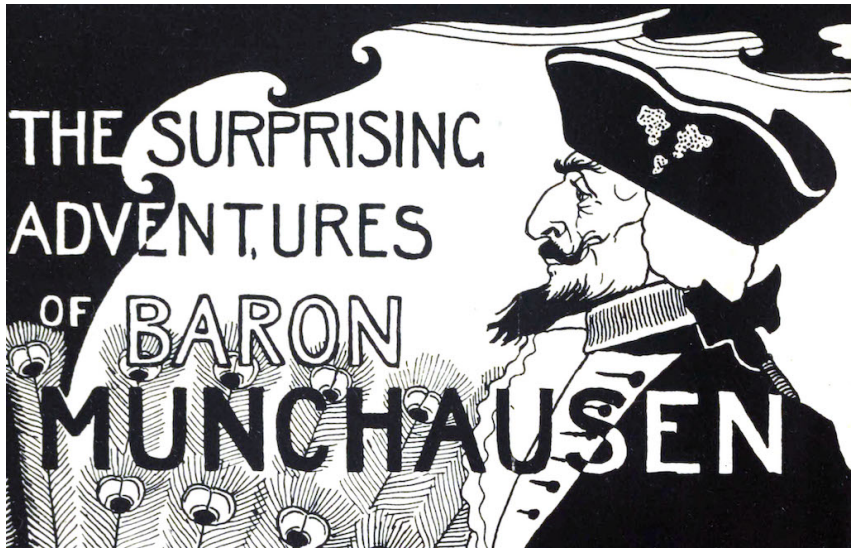
- In the section we discuss two **resampling** methods: **cross-validation** and the **bootstrap**.
- These methods refit a model of interest to samples formed from the training set, in order to obtain additional information about the fitted model.
- For example, they provide estimates of test-set prediction error, and the standard deviation and bias of our parameter estimates

# The Bootstrap

# Fundamentals of Bootstrap

- Applied when there is no theory to compute standard errors, confidence intervals, etc, or the theory cannot be trusted
- May not always work either, but works quite generally
- Fundamental idea: pretend the observed data is the population
- Resample observed data, create multiple samples
- From each sample, estimate parameters and assess variability

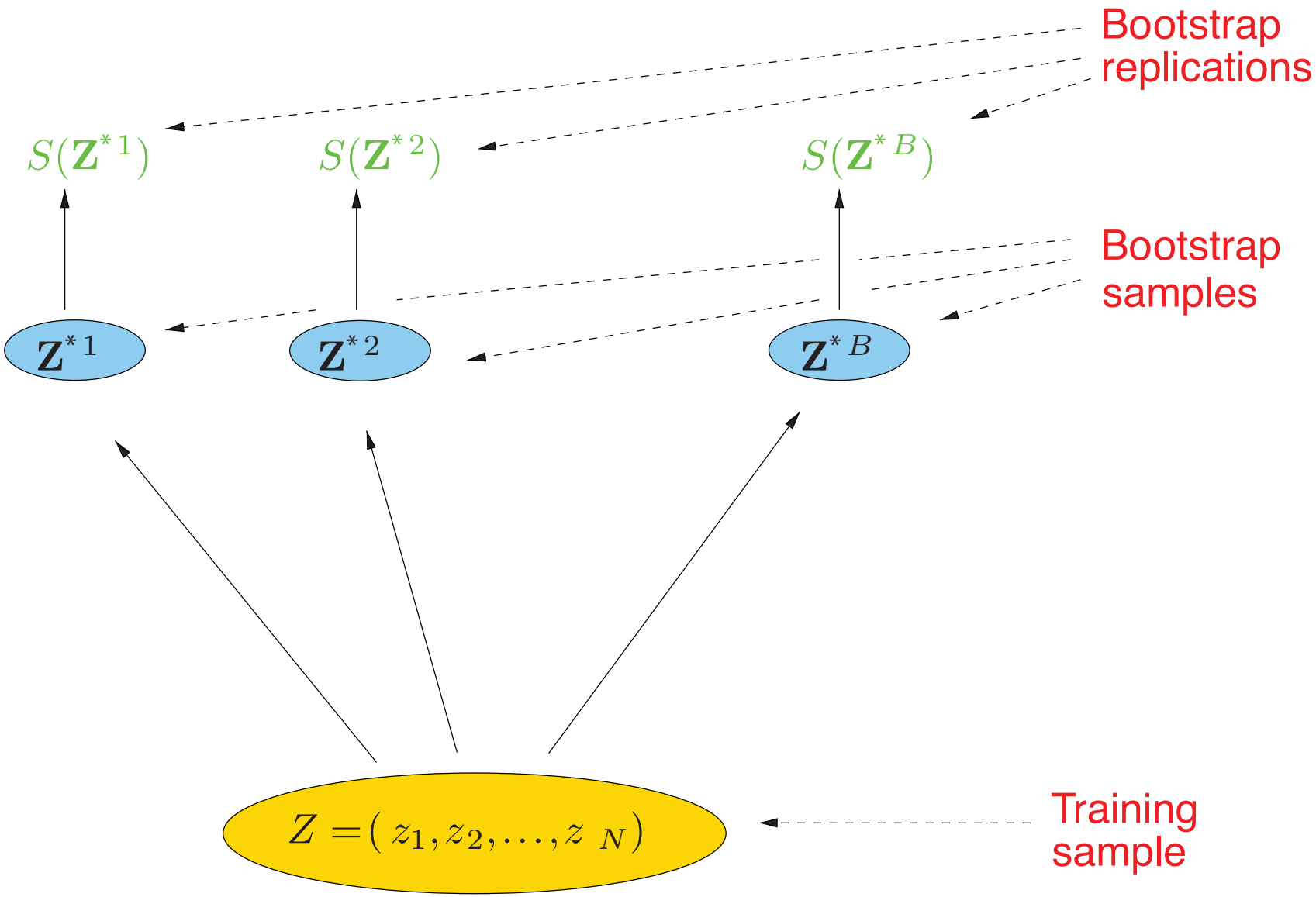
# History of the Bootstrap




The phrase "pull oneself up by one's bootstraps" is thought to come from "The Surprising Adventures of Baron Munchausen" by Rudolph Erich Raspe: The Baron had fallen to the bottom of a deep lake. Just when it looked like all was lost, he thought to pick himself up by his own bootstraps.

- Name comes from the phrase "pull oneself up by one's bootstraps."
- The Bootstrap was invented by Bradley Efron in 1979 as an extension to the jackknife, another resampling method.
- Since then, Efron and others have developed several extensions of the bootstrap (e.g. Bayesian, parametric, etc.)

# The Bootstrap

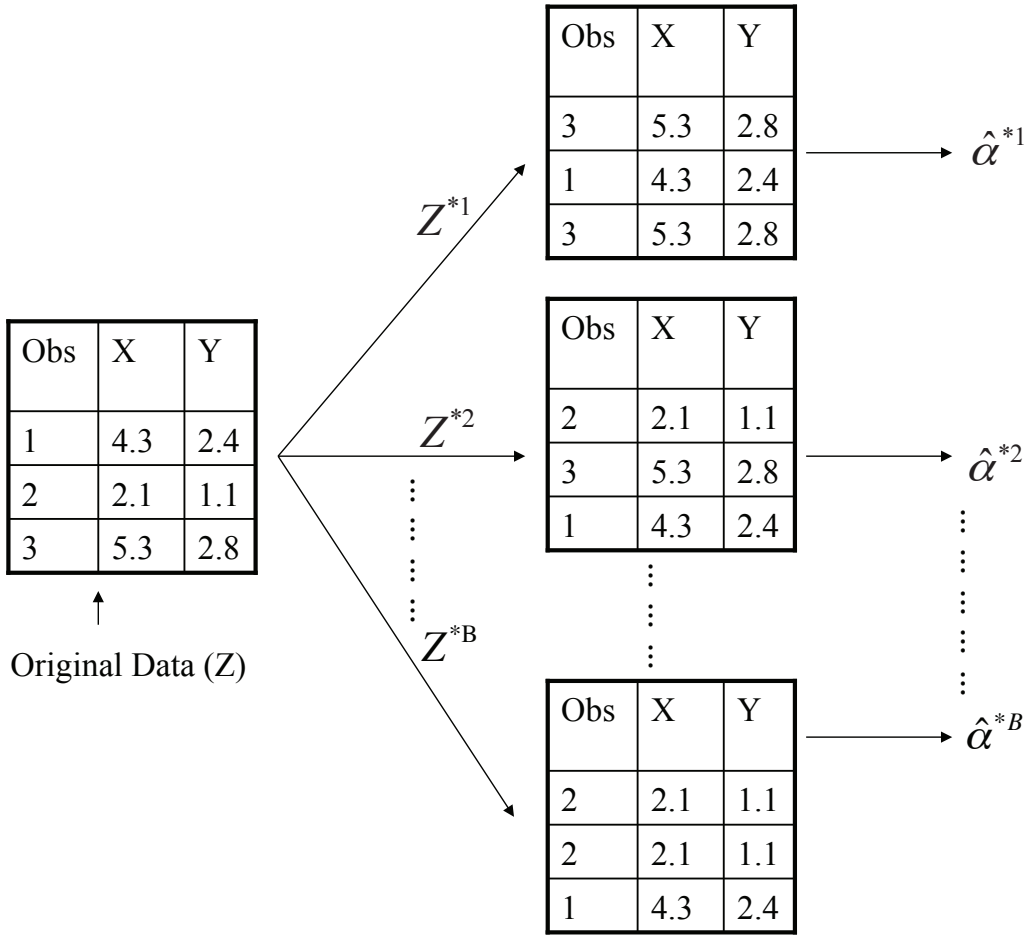


# The Bootstrap

1. Start with training set $Z = (z_1, \ldots, z_N)$; $\quad z_i = (x_i, y_i)$.
2. Randomly draw $B$ bootstrap samples $Z^*$ of size $N$ with replacement from the training data.
3. Fit a model to each $Z^*$, estimate $\hat{a}^*$ for all $B$ bootstrap sample.
4. Characterize behaviors of the fits over the $B$ replications $S(\hat{\alpha}^*)$. $S(\hat{\alpha}^*)$ could be, for example, the standard error:

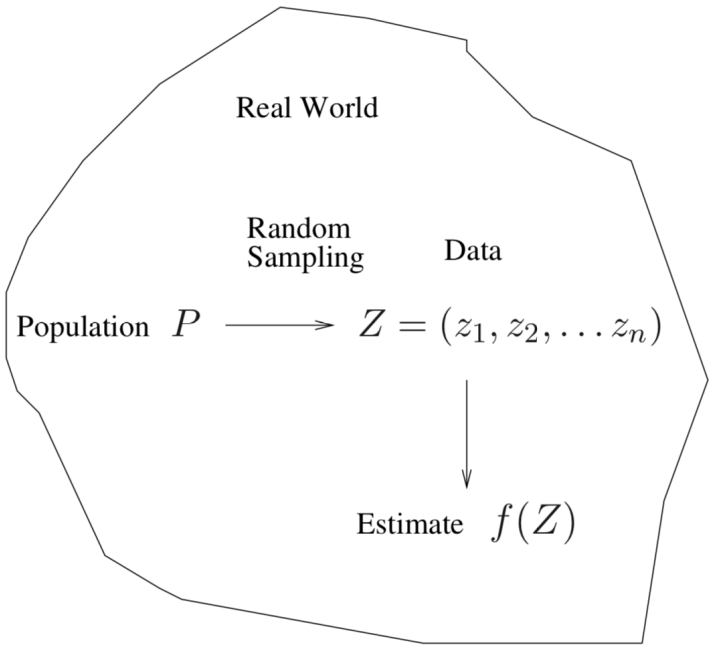
$$SE_B(\hat{\alpha}) = \sqrt{\frac{1}{B-1}\sum_{r=1}^{B}(\hat{\alpha}^{*r} - \bar{\hat{\alpha}}^*)^2}$$

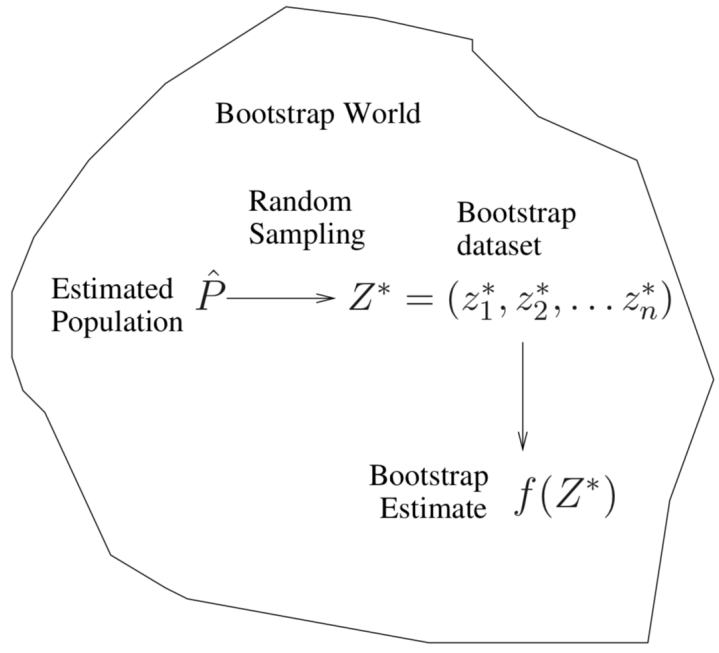
# Example with 3 Observations




A graphical illustration of the bootstrap approach on a small sample containing $n = 3$ observations. Each bootstrap data set contains n observations, sampled with replacement from the original data set. Each bootstrap data set is used to obtain an estimate of $\alpha$

# Bootstrap World vs. Real World



# Cross-Validation

# Finding $g(x) \approx f(x)$

- We aim to find $g(x)$ that approximates an unknown function $f(x)$
- There are, however, many model specifications to choose from.
- Two important goals:
  - **Model selection:** estimating the performance of different models in order to choose the best one.
  - **Model assessment:** having chosen a final model, estimating its prediction error (generalization error) on new data.
- For model assessment, we would ideally have a large designated test set that is never used to train models; this is often not feasible.
- For model selection, we need to train several models; don't want to throw away data.

# Partitioning Data to Learn $g(x)$


A typical split of data: 50% for training, and 25% each for validation and testing

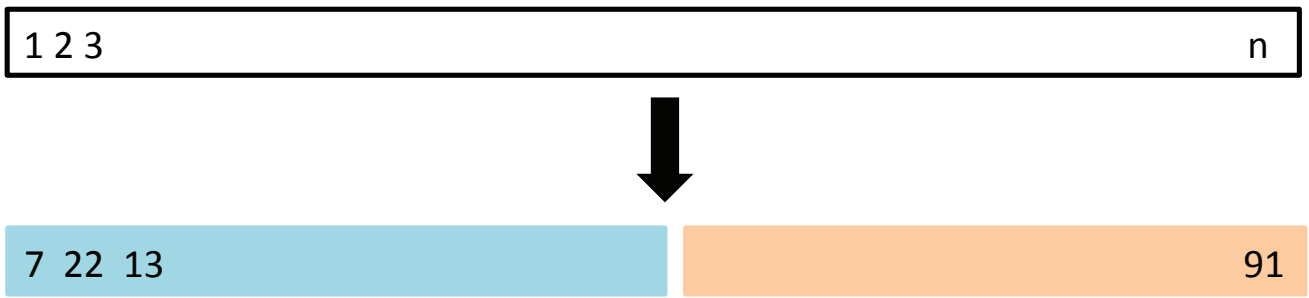
- **Training set**: A subsample used to fit a model (e.g. training set can be used to estimate model parameters).
- **Validation set**: A subsample used for model selection.
- **Test set**: A subsample used only model assessment; i.e. to measure the generalization error of a fully specified model.

# Validation-Set Approach

- Here we randomly divide the available set of samples into two parts: a training set and a **validation** or **hold-out set**.
- The model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set.
- The resulting validation-set error provides an estimate of the test error. This is typically assessed using MSE in the case of a quantitative response and misclassification rate in the case of a qualitative (discrete) response.

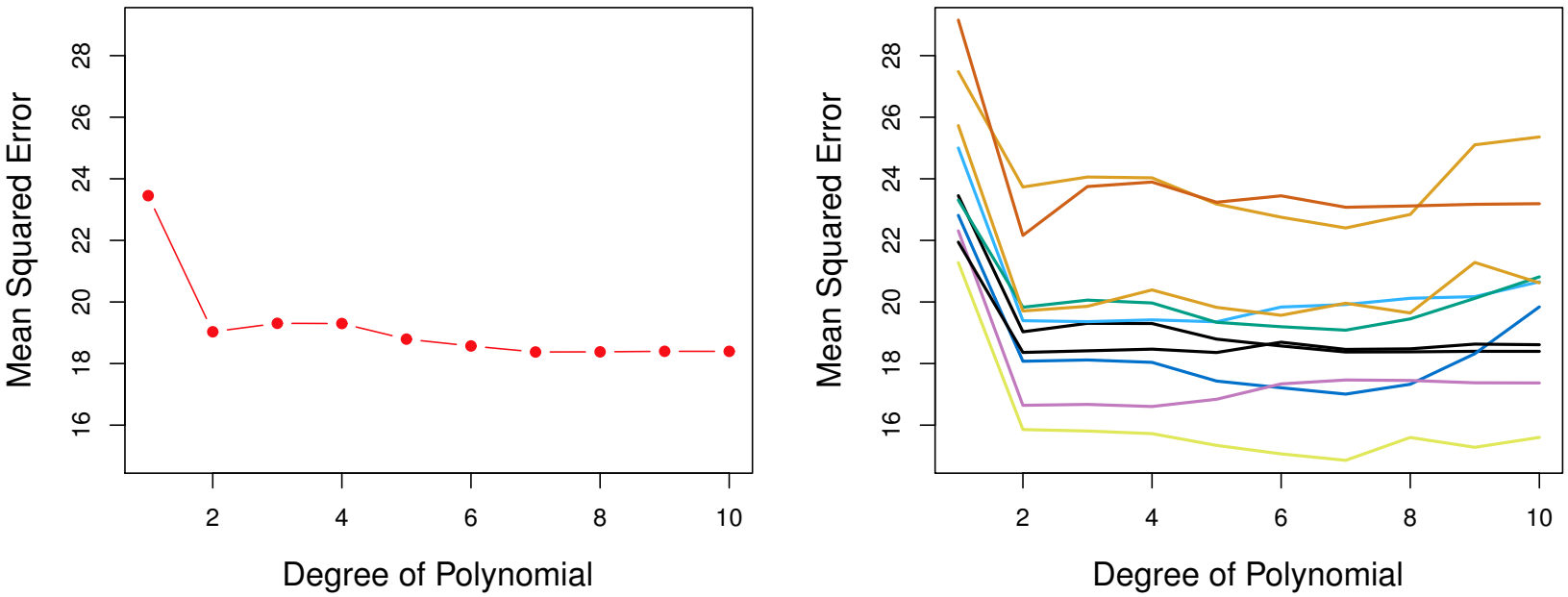
# The Validation Process

A random splitting into two halves: left part is training set, right part is validation set



# Example: Polynomial Regression

- Goal: Compare performance of linear vs higher-order polynomial terms in a linear regression




Left panel shows single split; right panel shows multiple splits

# Drawbacks of the Validation Set Approach

- The validation estimate of the test error can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the validation set.
- In the validation approach, only a subset of the observations—those that are included in the training set rather than in the validation set — are used to fit the model.
- This suggests that the validation set error may tend to overestimate the test error for the model fit on the entire data set. Why?

# K-fold Cross-Validation

- Estimates of generalization error from one train / validation split can be noisy, so shuffle data and average over K distinct validation partitions instead
- K-fold cross-validation is a widely used approach for estimating test error.
- Estimates can be used to select best model, and to give an idea of the test error of the final chosen model.

# K-fold Cross-Validation

How it works:

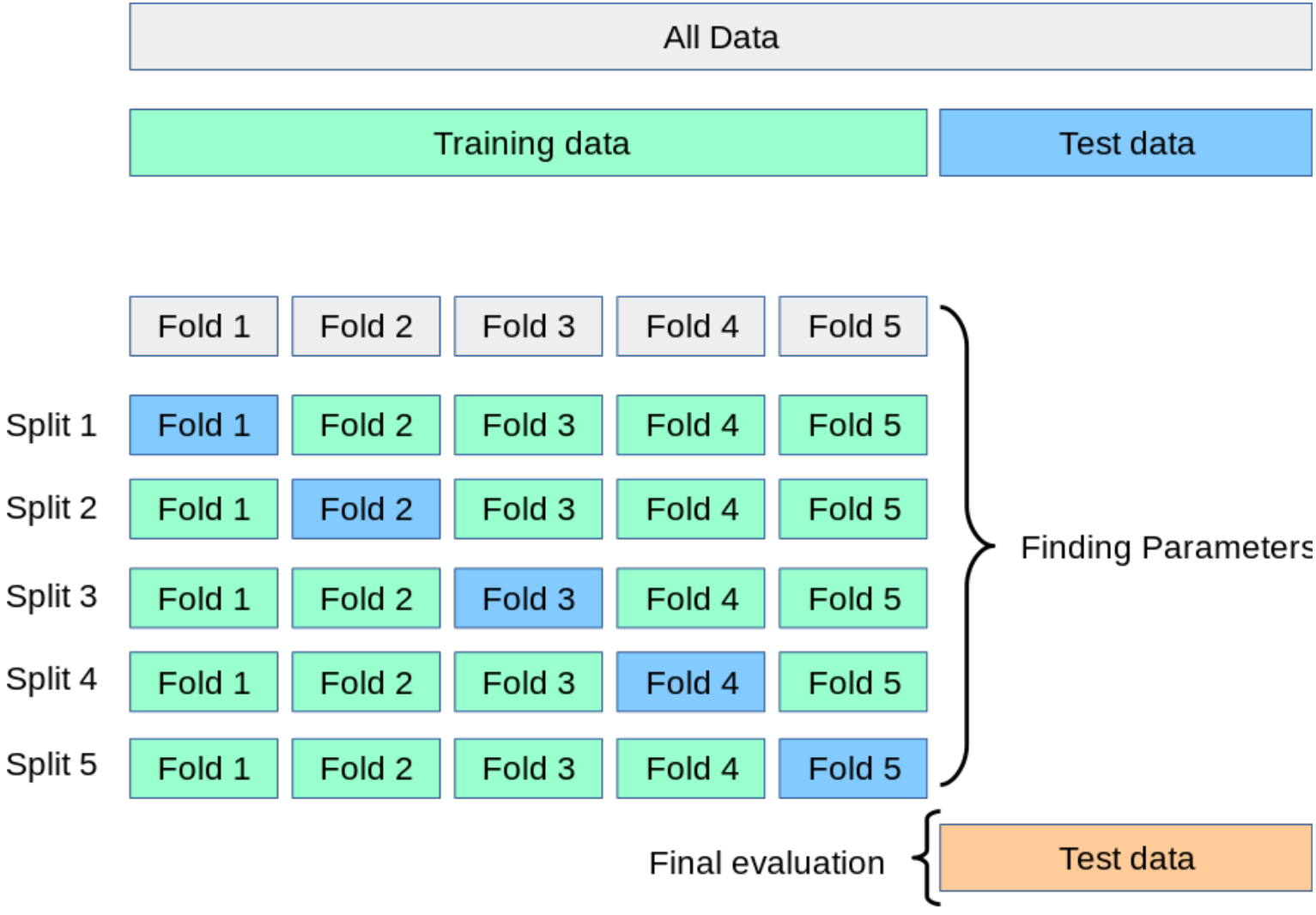
1. Randomly divide the data into $K$ parts of (roughly) equal length
2. Hold out one of the parts to be the "test" data.
3. Use all remaining data to train the model.
4. Predict on the held-out part and compute the error rate.
5. Repeat steps 2-4 for each of the $K$ parts.
6. Average all "test" errors to obtain the **cross-validation error**.

# K-Fold Cross-Validation Algorithm

**Algorithm 3:** K-Fold Cross-Validation

Randomly partition the data $\mathcal{D}$ into $K$ parts of (roughly) equal length $d_1, \ldots, d_k$;
Initialize empty error vector $\mathrm{E} \leftarrow (e_1, \ldots, e_K)$;
**for** $k \leftarrow 1$ **to** $K$ **do**
    $\mathrm{Tr}_k \leftarrow \mathcal{D} - d_k$;
    $\mathrm{Te}_k \leftarrow d_k$;
    Train model $g(\mathrm{Tr}_k)$;
    Predict outcome of $\mathrm{Te}_k$ using model $g(\mathrm{Tr}_k)$ yeilding $\hat{y}_{\mathrm{Te}_k}$;
    Compute error $e_k$ with $\hat{y}_{\mathrm{Te}_k}$ and $y_{\mathrm{Te}_k}$;
    Insert $e_k$ into error vector E;
**end**
**return** $\sum_{k=1}^{K} \frac{n_k}{n} e_k$;

# K-Fold Cross-Validation Details




Divide data into $K$ roughly equal-sized parts ($K = 5$ here)

# K-Fold Cross-Validation Details

- Let the $K$ parts be $C_1, C_2, \ldots C_K$, where $C_k$ denotes the indices of the observations in part $k$. There are $n_k$ observations in part $k$: if $N$ is a multiple of $K$, then $n_k = n/K$.
- Compute

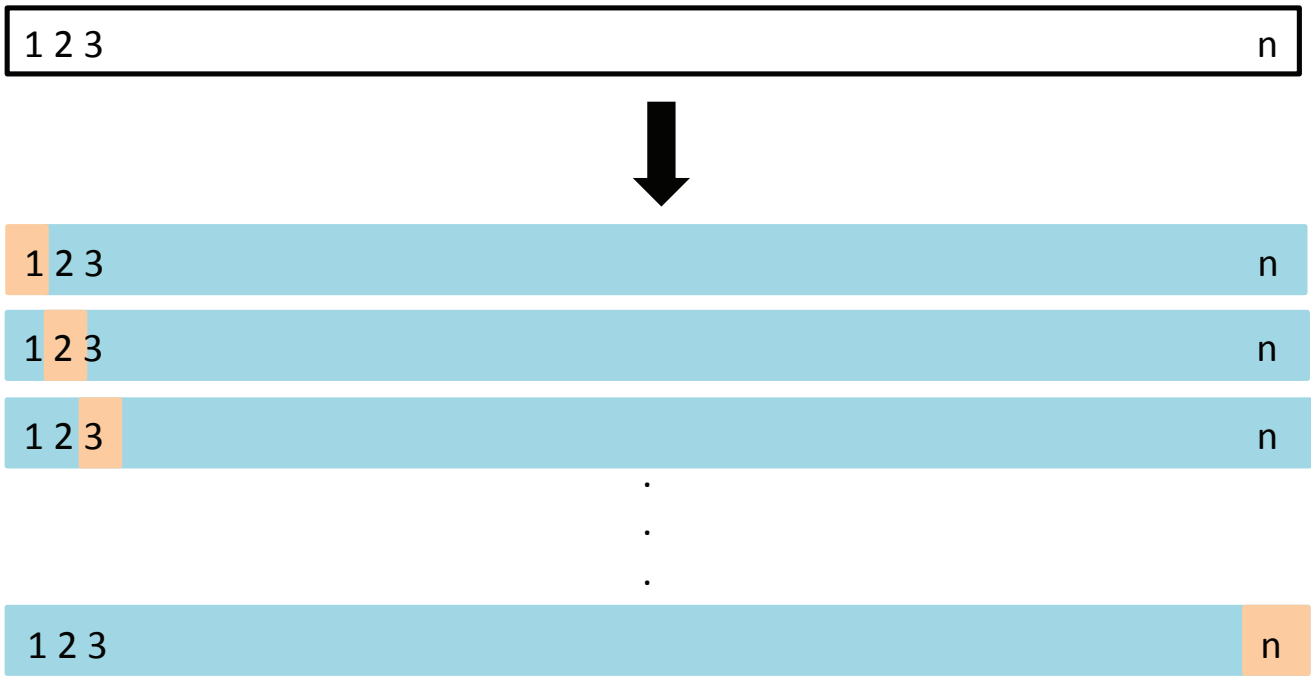
$$CV_{(K)} = \sum_{k=1}^{K} \frac{n_k}{n} MSE_k$$

where $MSE_k = \sum_{i \in C_k} (y_i - \hat{y}_i)^2 / n_k$, and $\hat{y}_i$ is the fit for observation $i$, obtained from the data with part $k$ removed. - Setting $K = n$ yields $n$-fold or leave-one out cross-validation (LOOCV).

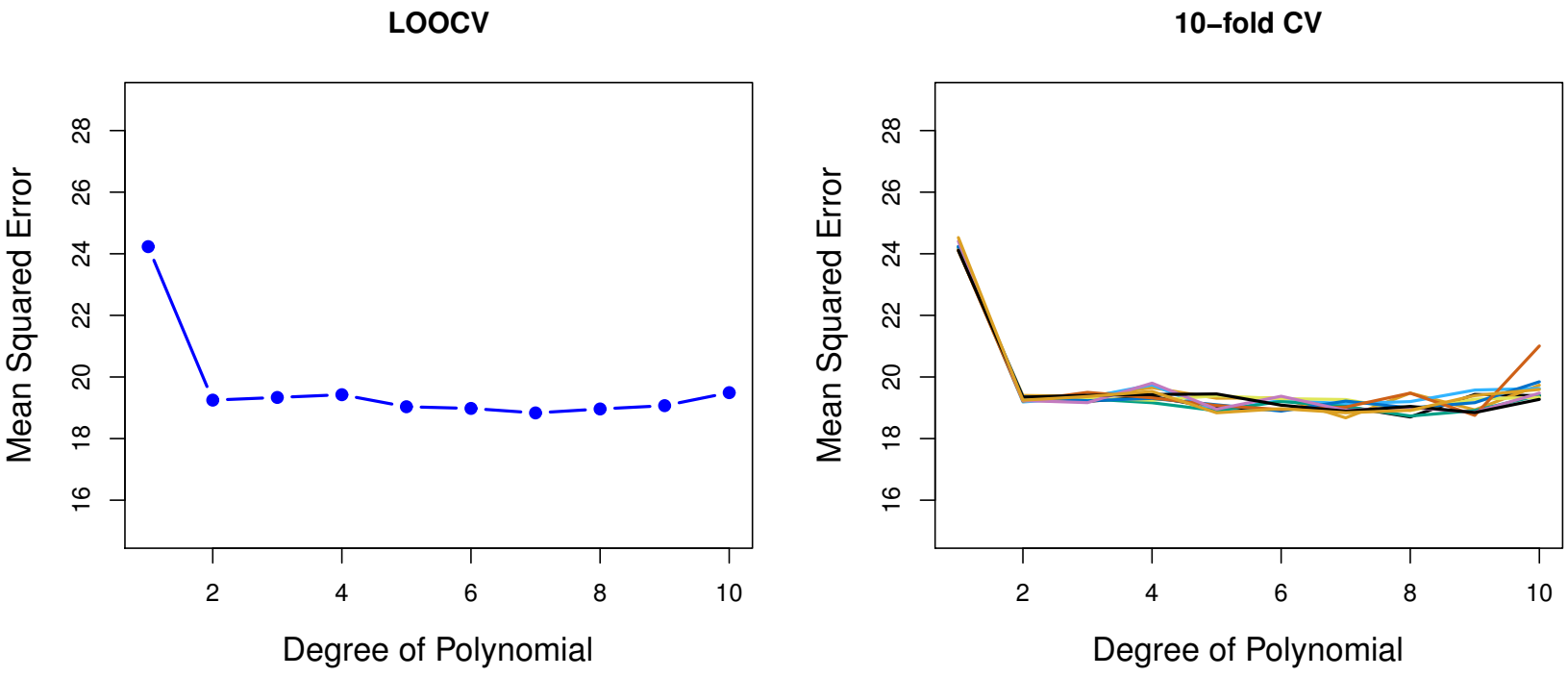
# More About Cross-Validation

- $K$ can be anything; popular values are 5, 10, $n$.
- The cross-validated error rate tends to be closer to the true error rate than to the apparent error rate.
- The computational cost can become a concern, particularly if there are many tuning parameters

# Leave One Out Cross-Validation (LOOCV)



# Example: Polynomial Regression




On the right, each line represents a different partition of the data.

# Other Issues with Cross-Validation

- Since each training set is only $(K-1)/K$ as big as the original training set, the estimates of prediction error will typically be biased upward. Why?
- This bias is minimized when $K = n$ (LOOCV), but this estimate has high variance, as noted earlier.
- $K = 5$ or 10 provides a good compromise for this bias-variance tradeoff.
- Like our choice of models, our CV decision involves a bias-variance tradeoff

# Hyperparameter Selection

# Hyperparameters

- **Hyperparameters**, also called **tuning parameters**, refer to specifications that change the behavior of the learning model (e.g. $k$ in KNN) or the learning algorithm (e.g. learning rate $\lambda$ for gradient descent).
- Cross-validation is used to select the **hyperparameters** that will give us a $g(x)$ that best generalizes to new data.
- The cross-validation procedure is performed in a **hyperparameter search**, which can be conducted in several ways.

# Hyperparameters vs. Parameters

- While **parameters** like $\beta$ in regression are set during training, **hyperparameters** are set before training.
- Hyperparameters come in two main types:
  - **Model hyperparameters** are attributes of models, usually refer to flexibility of fit.
  - **Algorithm hyperparameters** refer to methods of training the model; these hyperparameters don't affect model performance, but rather the quality and speed of learning.

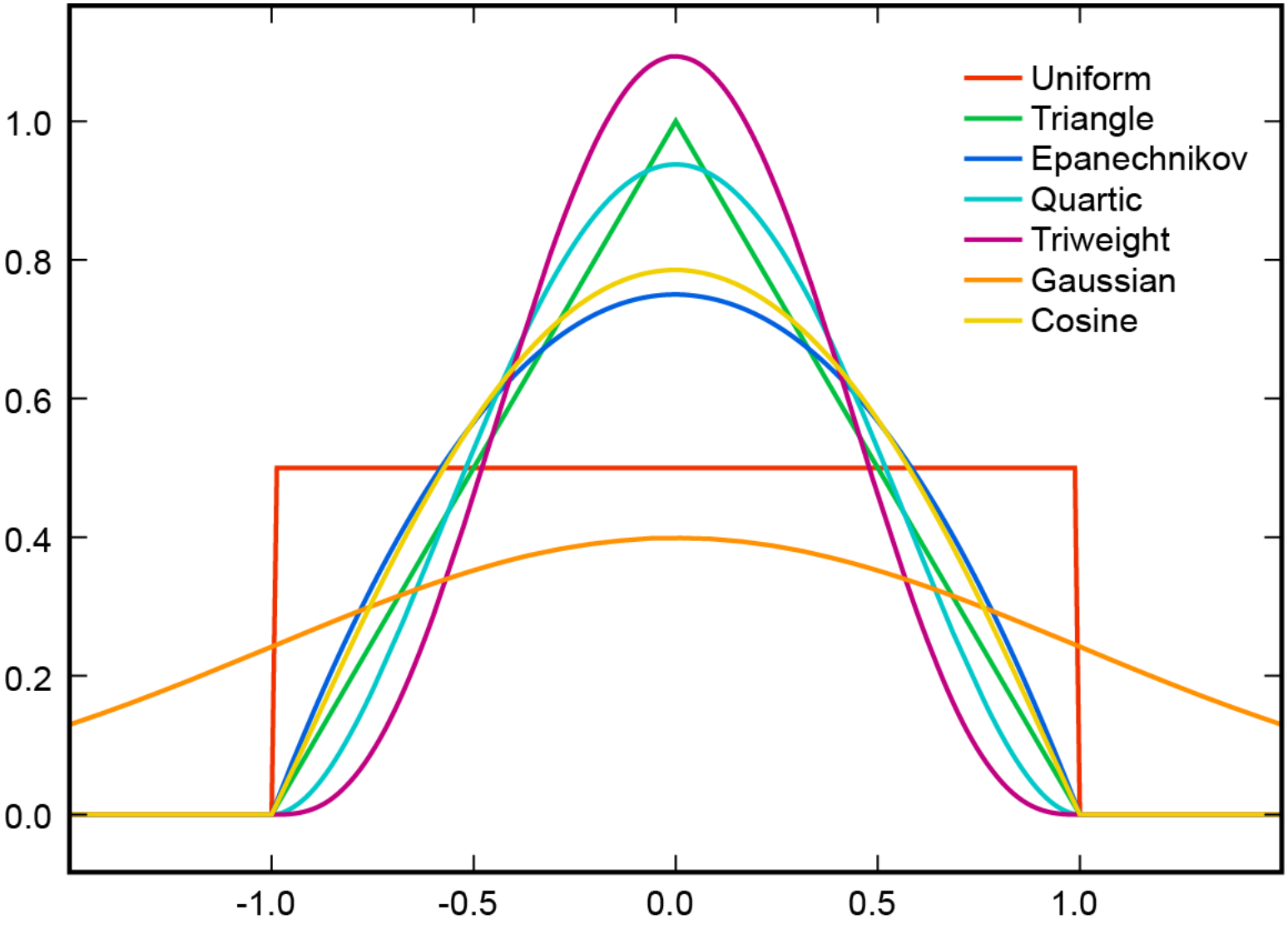
# Model Hyperparameter Example: Kernel KNN

- $k$ in KNN is a **model hyperparameter**.
- KNN can also be extended to use additional **model hyperparameters**.
- Ordinary KNN gives equal weight to all neighbors, the equivalent of a "uniform" kernel.
- We could potentially improve performance by adding a weight to each of these neighbors based on their distance from the reference observation.
- Weighted KNN downweights neighbors farther from our target point:

$$\hat{C}(x) = \sum_{i=1}^{k} w_i * y_i;\ 0 < w_i < 1$$

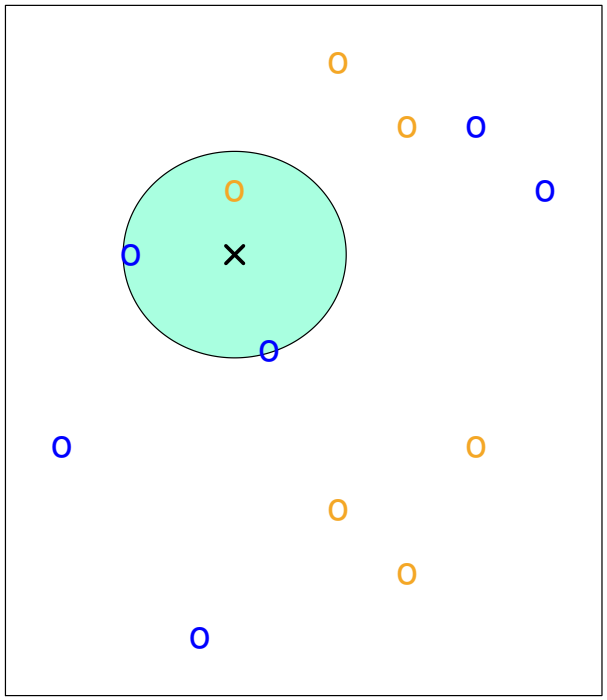
- We want our weight $w_i$ to be
  - small if the distance to our reference point is large
  - large if the distance to our reference point is small

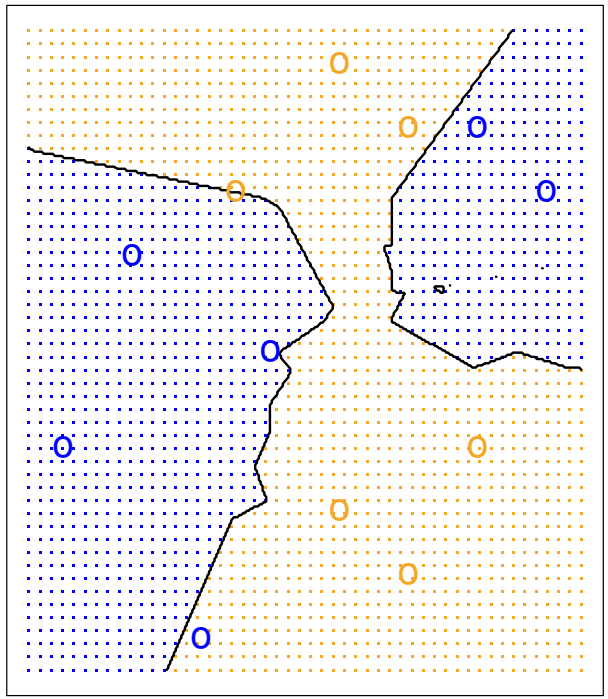
# Model Hyperparameter Example: Kernel KNN




We can use kernels that are functions of the distance between points. This is another hyperparameter in addition to $k$.

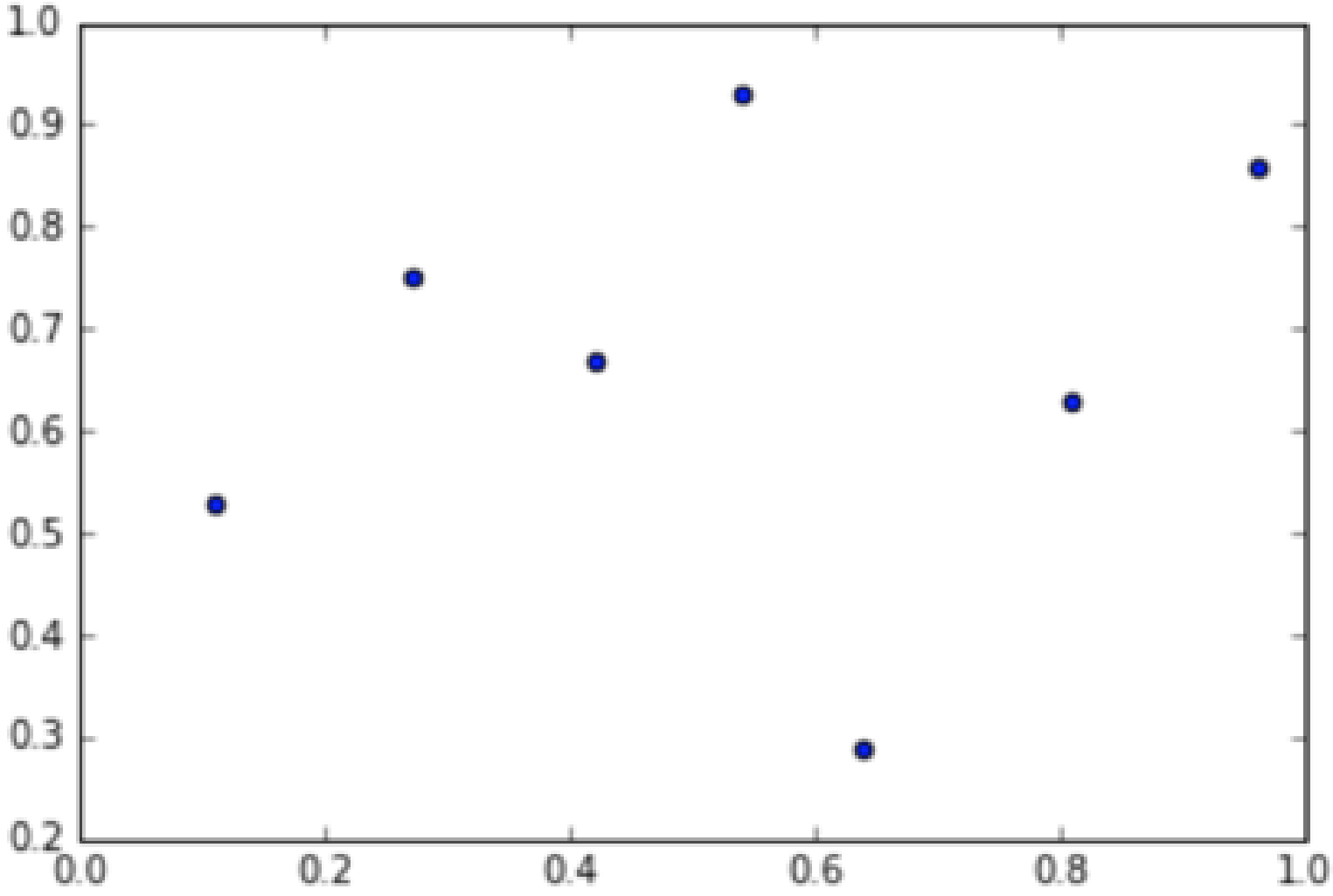
# Algorithm Hyperparameters: K-D Tree vs. Brute Force





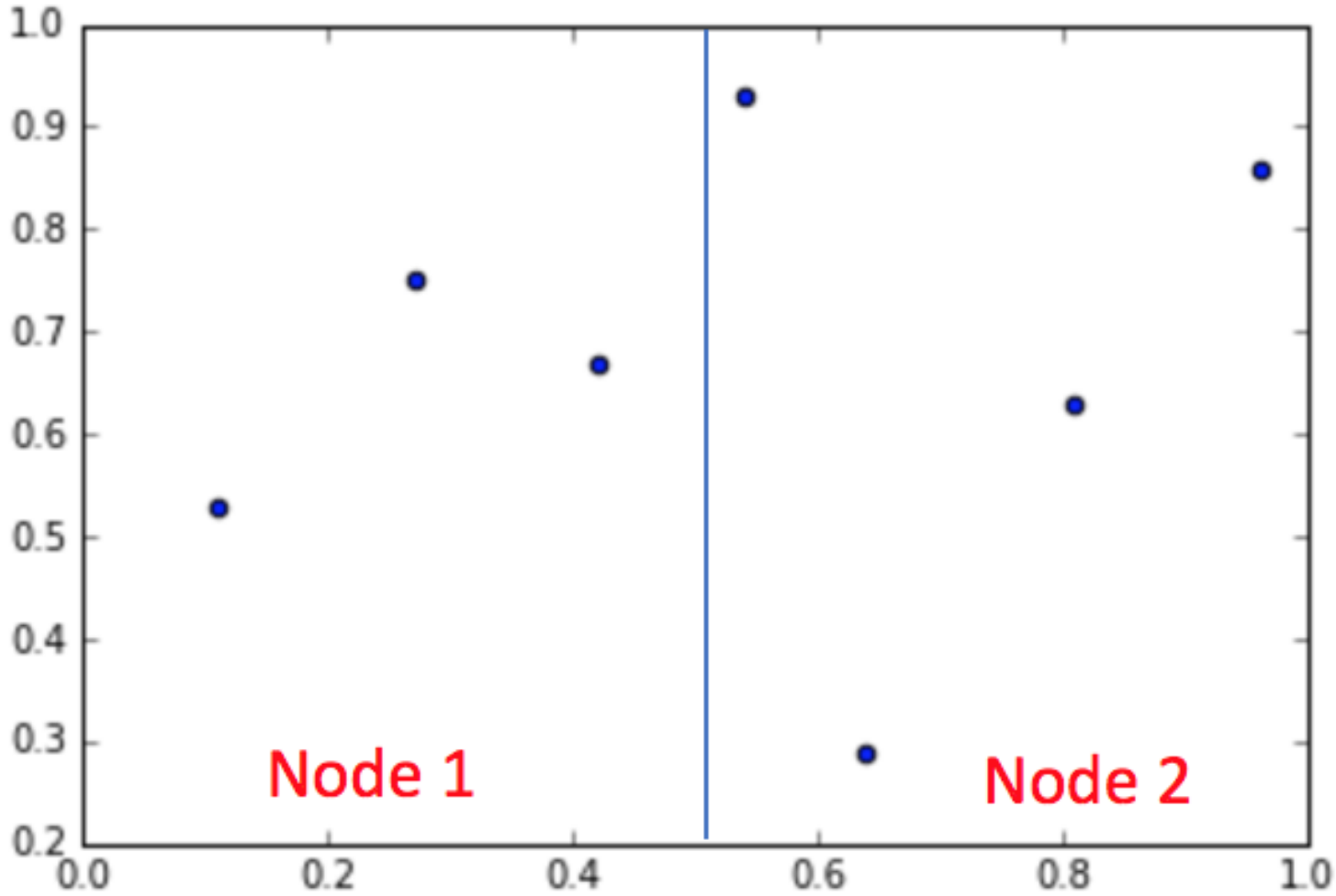
- Recall the KNN algorithm: in order to find the $k$ nearest neighbors, we have to calculate the distance between a reference point and all of our training observations.
- This can be VERY expensive in a large dataset.
- As with many other learning algorithms, often there are computational shortcuts that get you close to the same answer (recall stochastic gradient descent).

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



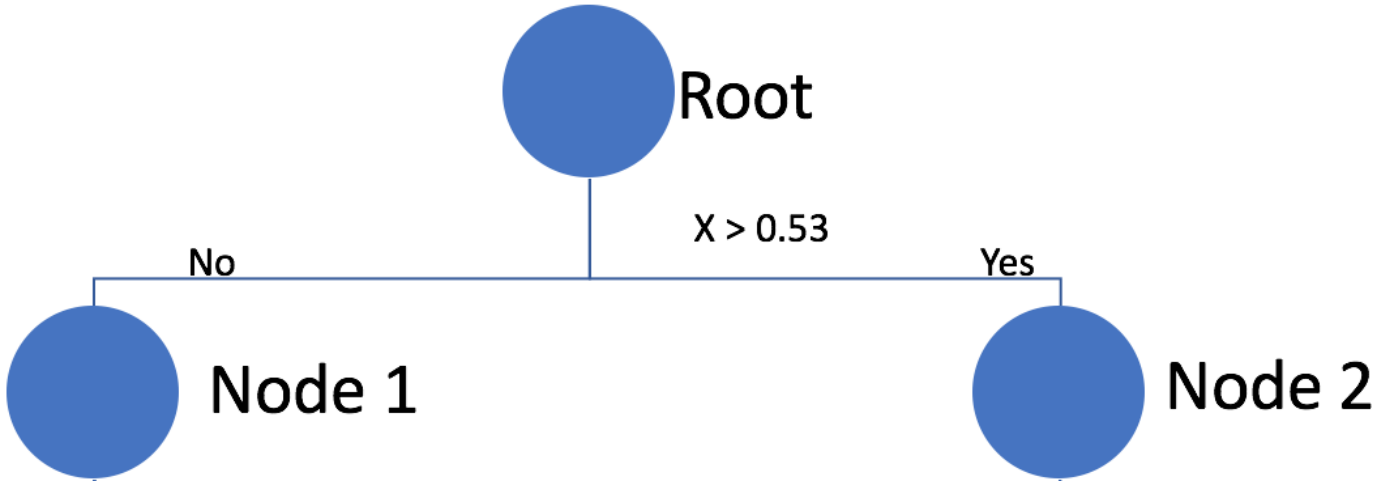

Say we have 2-dimensional training data like the following.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



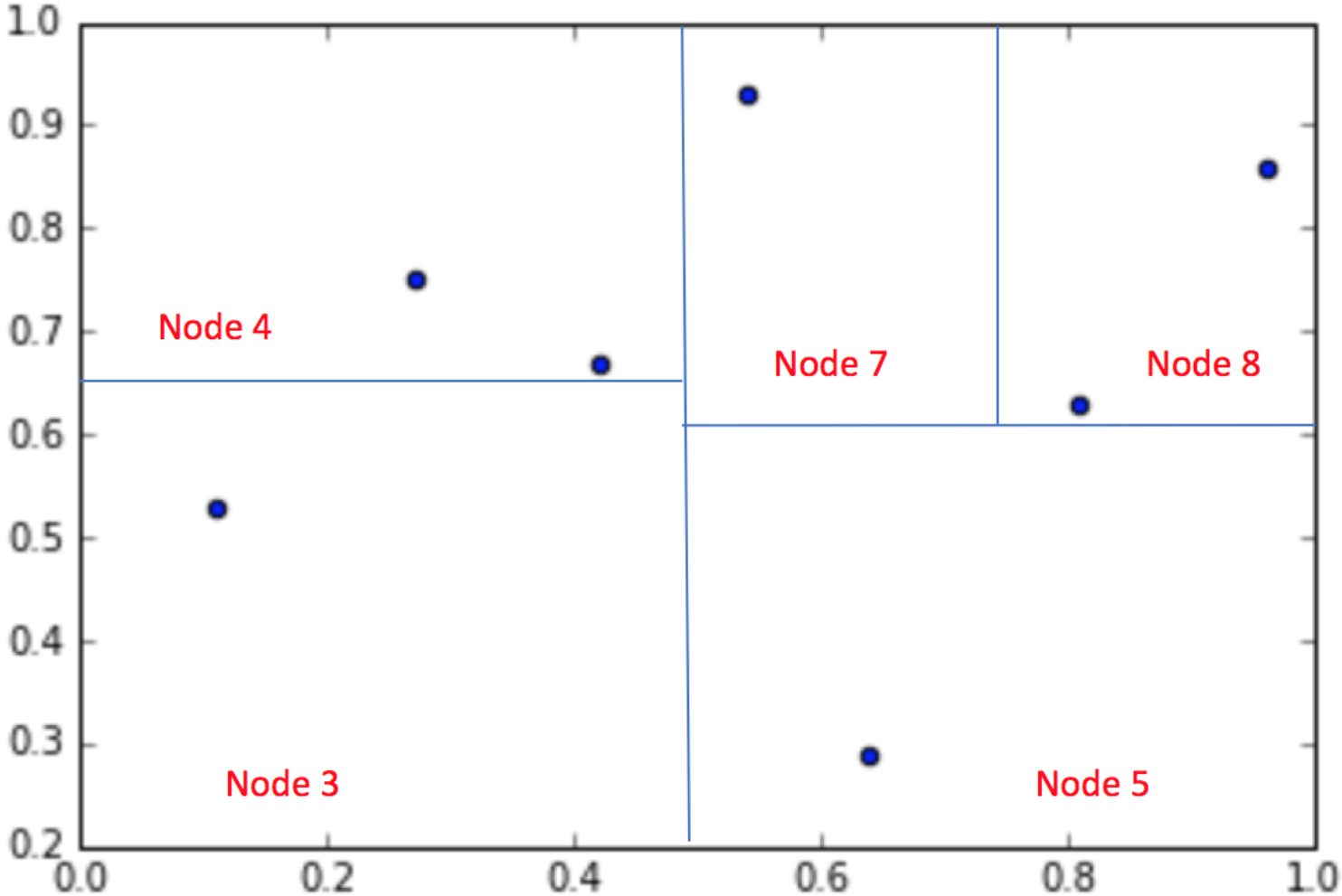

First split a randomly selected feature at the median.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



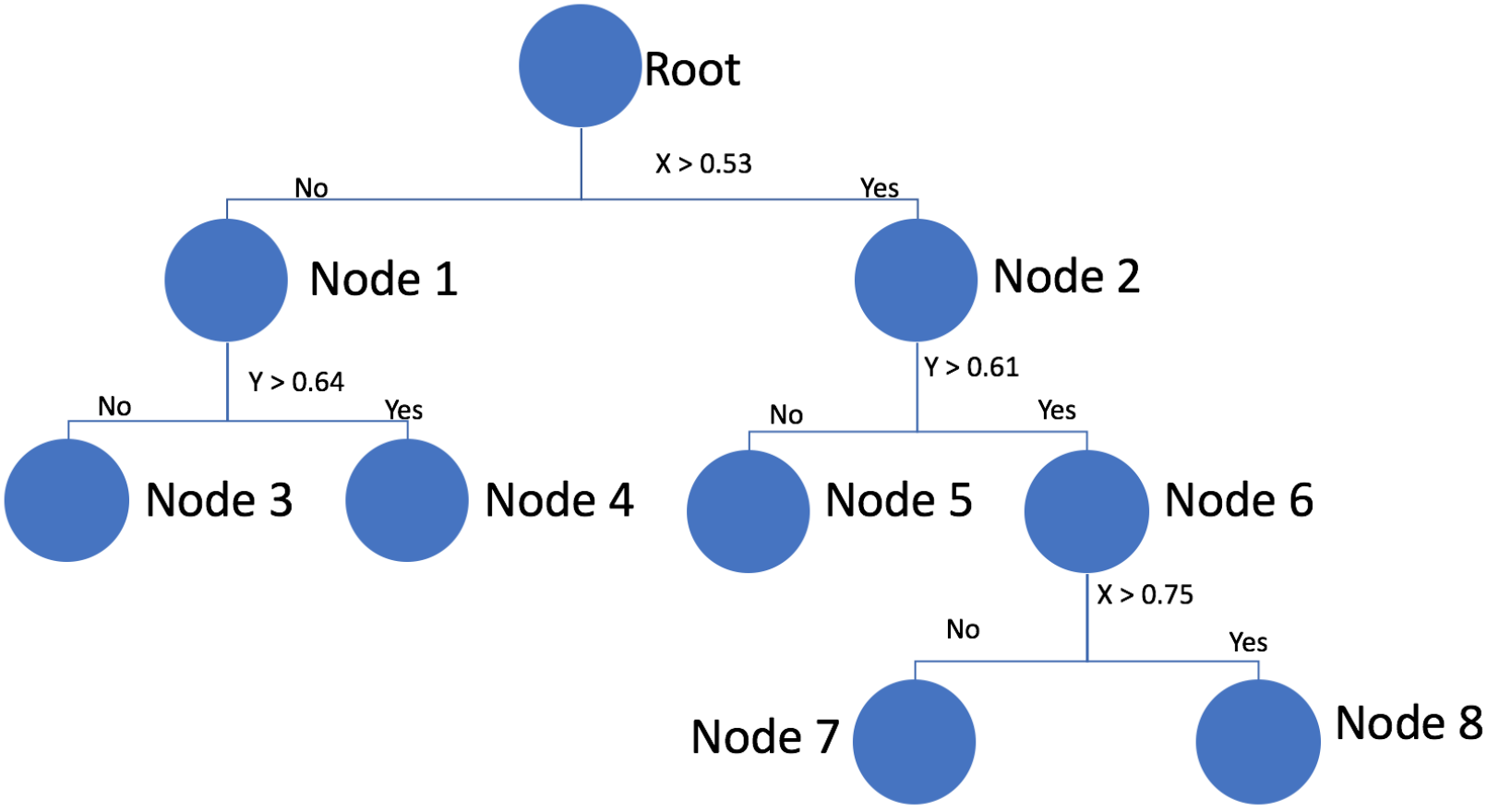

Now you have a tree with two nodes.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



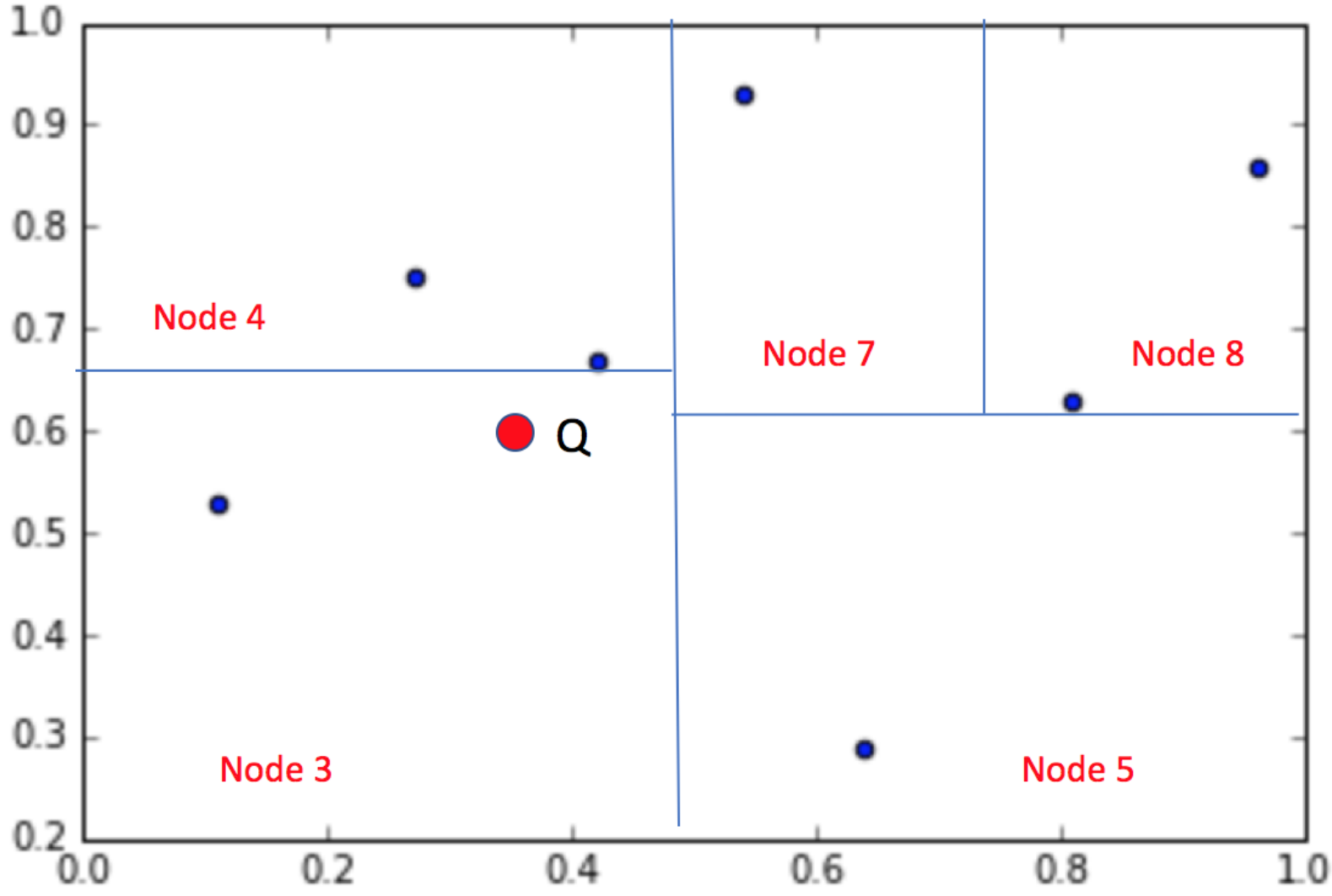

Continue to split the feature space along a random dimension until you have no more than $m$ observations in each bounding box. In this case $m = 2$.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



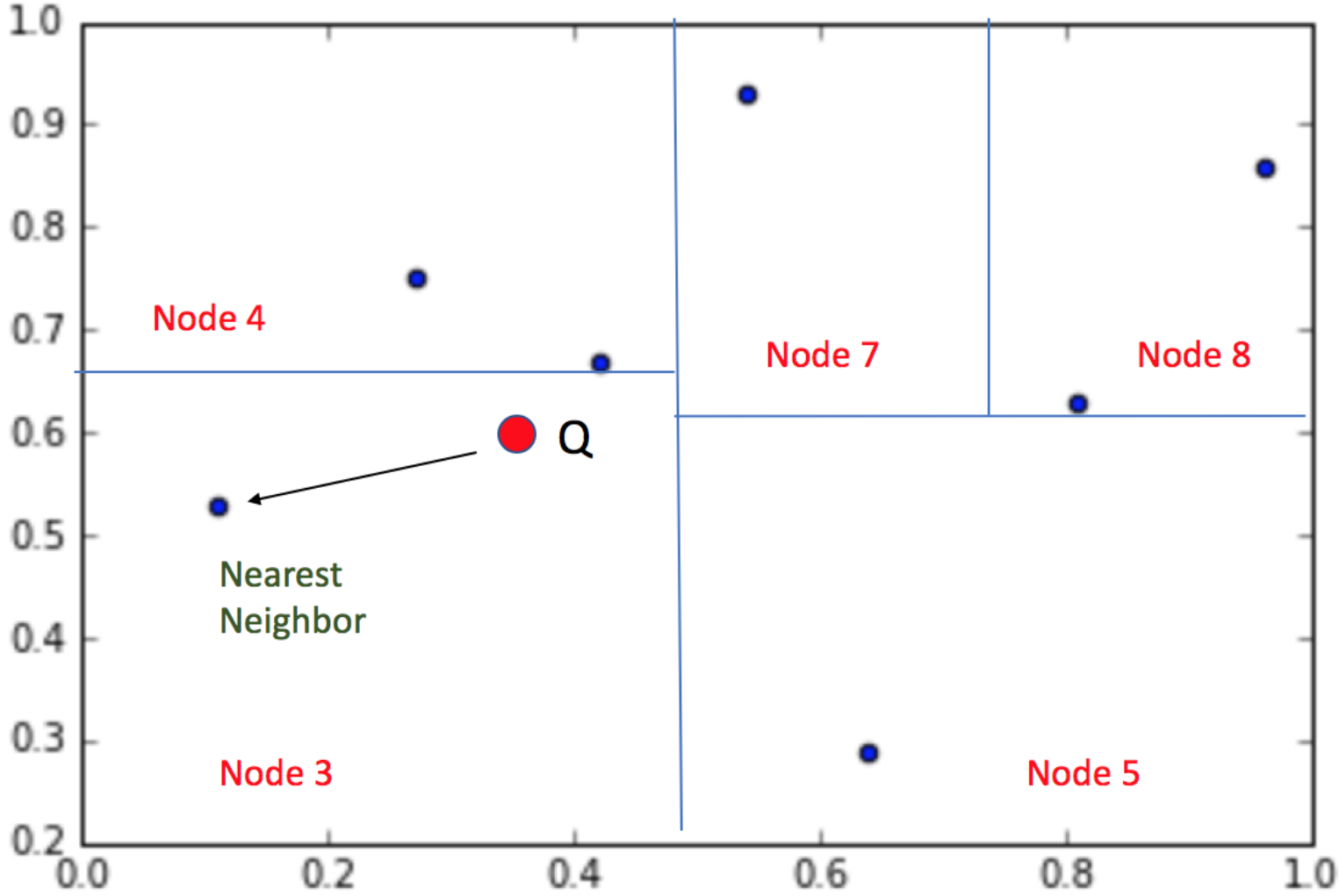

Each of these bounding boxes are represented by terminal nodes on the following tree.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



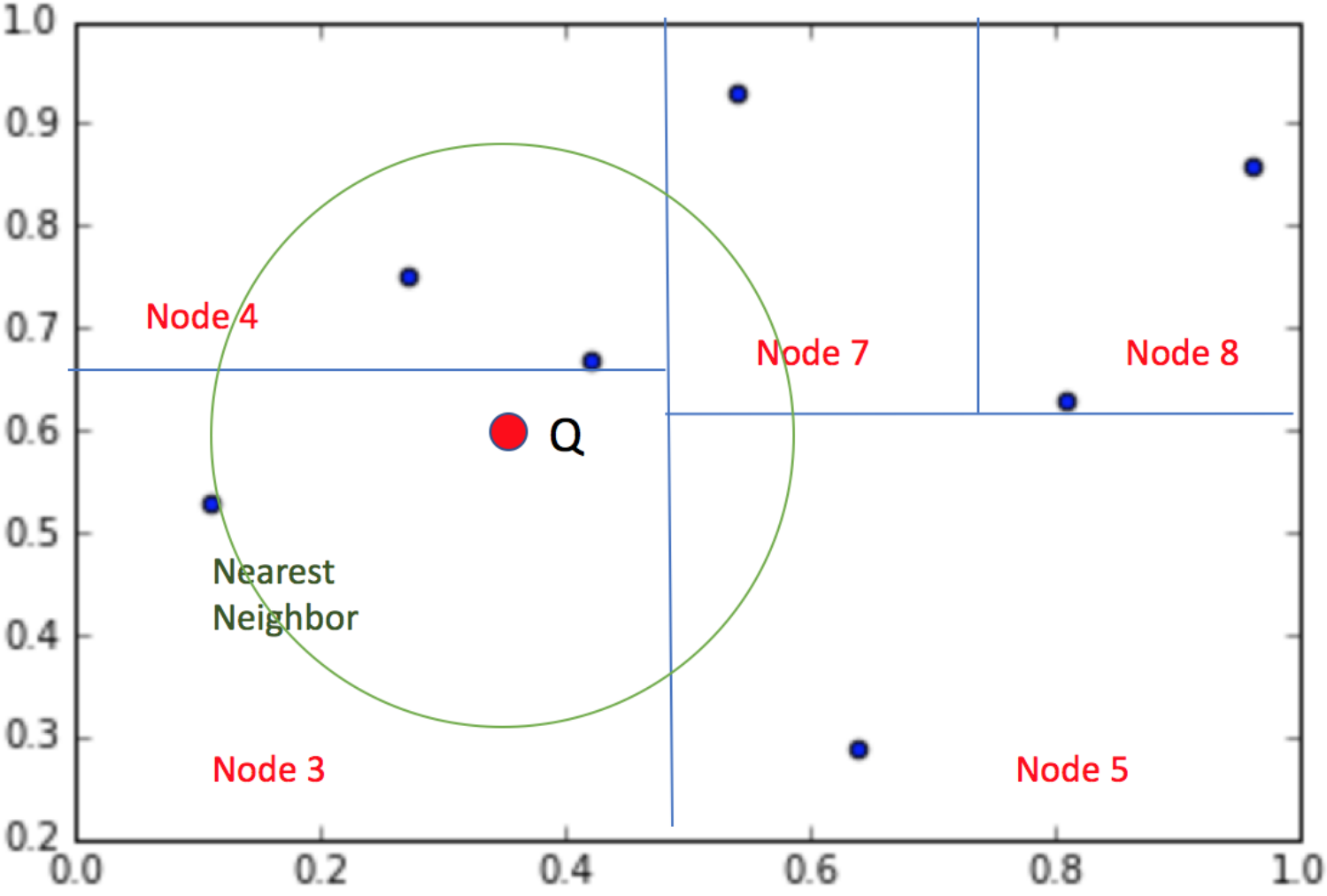

When predicting the class of a new observation, we first determine which bounding box it is in.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force



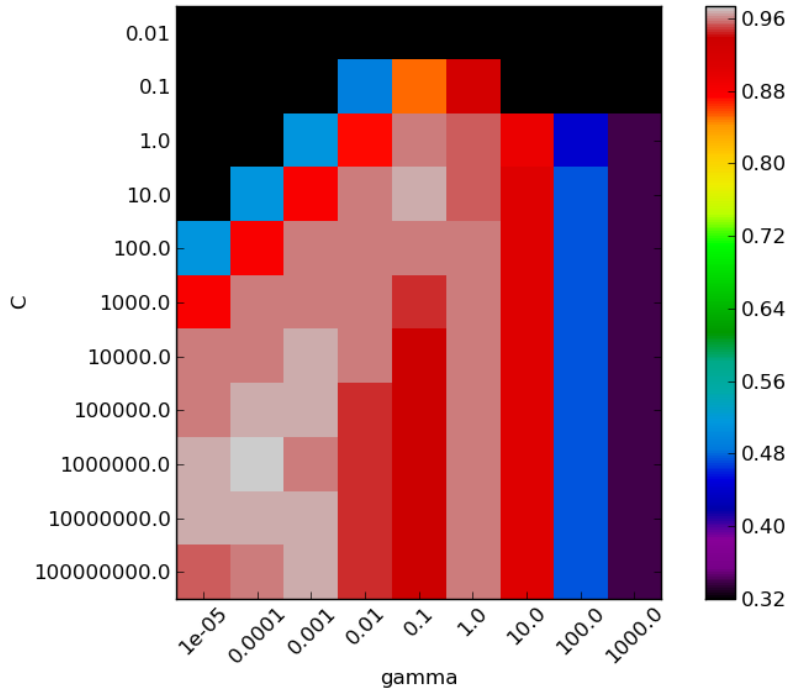

Instead of calculating the distance between all observations in the training data, we restrict our search to the bounding box where the query/reference point resides.

# Algorithm Hyperparameters: K-D Tree vs. Brute Force




Unfortunately, we oftentimes do not find the true nearest neighbor, but we get close!

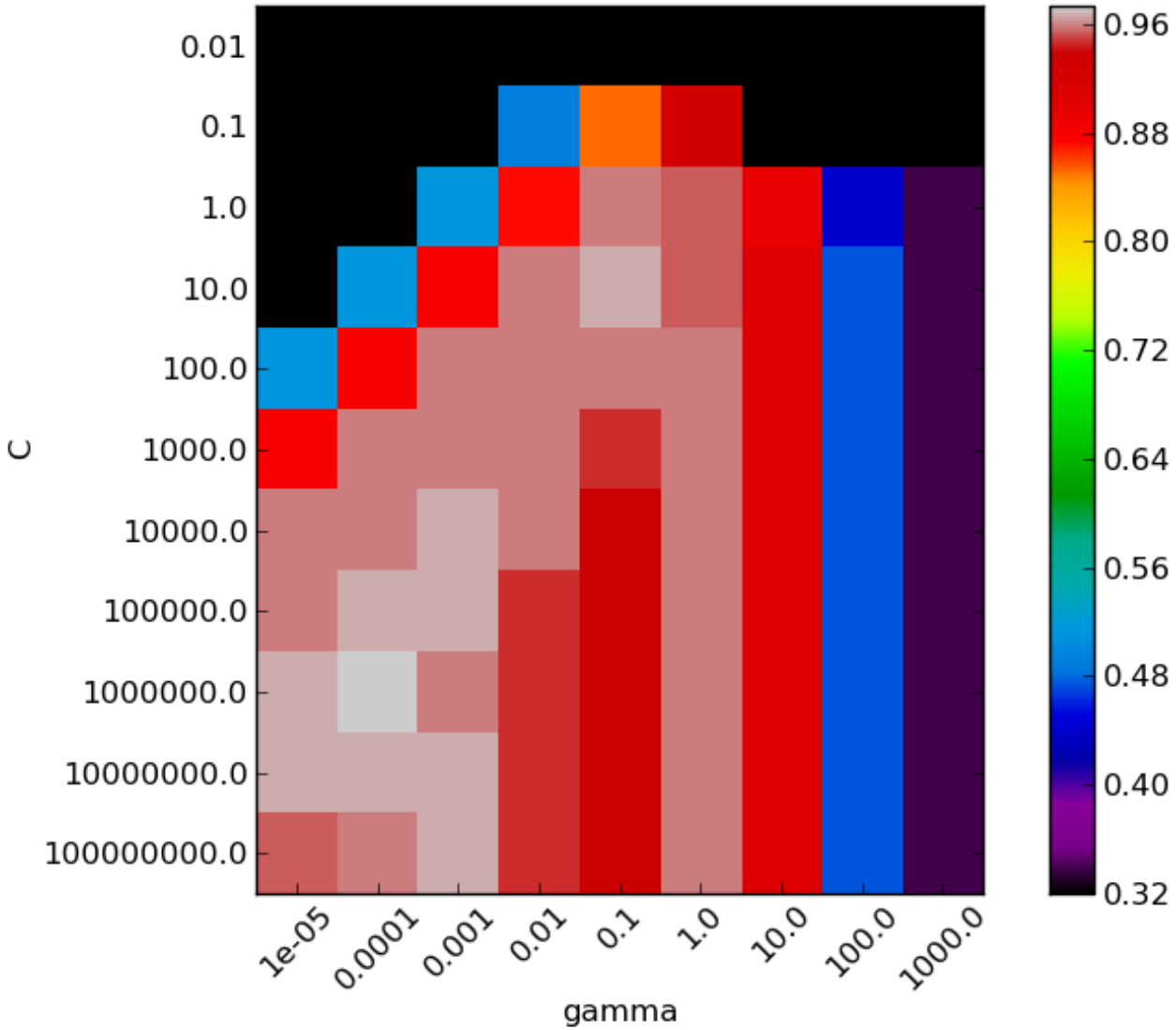
# Grid Search




Here we see an example of a grid search using two hyperparameters for support vector machines.

- Exhaustive search of all potential parameter combinations.
- Parameters, even when continuous, must be **discretized**
- Suffers from the **curse of dimensionality**. *Why?*

# Grid Search




Here we see an example of a grid search using two hyperparameters for support vector machines.

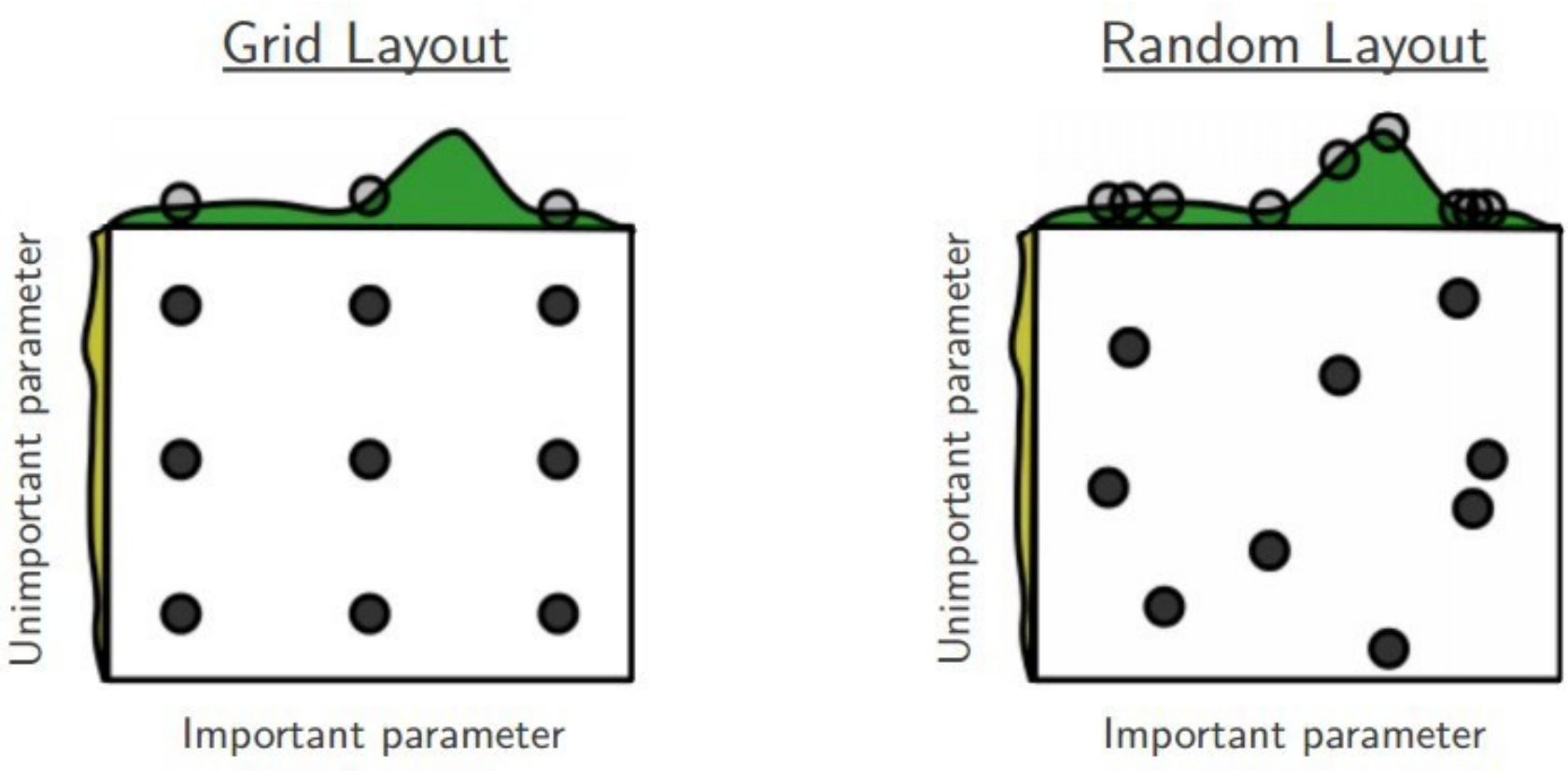
# Problems with Grid Search

- The search space can become extremely large as the number of hyperparameters or feature selection/preprocessing steps increases
- One solution is to randomly sample a subset of possible models using a **randomized hyperparameter search**
- Another solution is to use a **surrogate model** in **sequential model-based optimization (SMBO)**

# Randomized Hyperparameter Search

- In randomized search, we treat each hyperparameter as a **distribution** and sample values from it at random.
- Randomized search does not exhaustively search the space of possible hyperparameter combinations, instead taking repeated random samples from distributions of hyperparameters.
- Because of this, a **computational budget**—the number of random samples to be drawn—can be set ahead of time.
- Randomized search can **discretize** parameters, but does not necessarily have to.

# Randomized Hyperparameter Search




On the left we see the difference in sampling of parameters in grid search vs. random search. The green distribution represents a hyperparameter that is important if tuned. The high peak represents the area of the optimal hyperparameter value.

# Problems with Randomized Hyperparameter Search

- It can become quite expensive to check many different model configurations.
- Each new sample of hyperparameters must be used to train a new model on training data, make predictions with the validation data, and calculate our performance metric.
- With randomized search, the algorithm has no memory of past successes and failures, wanders randomly through the hyperparameter space
- An alternative, **bayesian hyperparameter search**, uses past records of hyperparameters and performance metrics to decide where to look next.

# Bayesian Hyperparameter Search

- Uses a probabilistic model of hyperparameter values. Maps these hyperparameters probabilistically to a score on a chosen performance metric.
- The approach works as follows:
  1. Build a **surrogate probability model** that predicts how hyperparameter values will impact the model performance based on **prior knowledge**.
  2. Find the hyperparameters that perform best on the surrogate
  3. Train the model with these hyperparameters on the **training data**; evaluate on the **validation set.**
  4. Update the surrogate model incorporating the new information about how hyperparameter values impact the model's performance.
  5. Repeat steps 2–4 until max iterations or time is reached

# AutoML

- Why not just take humans out of the loop altogether?
- ML algorithms (KNN, logistic regression, neural networks) can be treated like hyperparameters (choose the algorithm that minimizes validation error)
- Features and preprocessing can also be treated like hyperparameters (choose feature representations that minimize validation error)