CAP4770 – Intro to Data Science

Machine Learning Landscape

Jan. 9, 2024

Prof. Ye Xia

Machine Learning (ML) Example - Email Spam Filter

- You are given examples of spam emails flagged by users, and examples of nonspam emails.
- The system will (automatically) learn from the examples so that it can flag future emails as spams or nonspams.
- The set of examples that the system uses to learn is called the **training set** or **training data**. Each example is called a **training instance**.
- Compare this with a non-ML system: You examine the training data and identify the words and phrases that are associated with typical spams, e.g., credit card, free, 4U, or other patterns. You then write code to detect such words or patterns. An email is flagged if a number of them are detected.

- A ML-based algorithm can automatically discover/learn the patterns associated with spams, including patterns that are hard to identify or describe by a human operator.
  - Consider another simple example of speech recognition between the words ‘one’ and ’two’. The each training instance contains digitized speech audio, which is difficult for a human to describe.
- What if spammers change 4U to ‘for U’ to get around the filter? You will have to update your code, whereas a ML-based filter can automatically learn from new emails that contain ‘for U’ and are flagged as spams.
- Side note: At this point, the above ML algorithm may seem like magic. It is not. It is just a more abstract, general classification algorithm that is capable of dealing with abstract data.

To summarize, ML is great for

- Problems for which existing solutions require a lot of fine-tuning or long lists of rules: A Machine Learning algorithm can often simplify code and perform better than the hard-coded approach.
- Complex problems for which using a traditional approach yields no good solution: The best Machine Learning techniques can perhaps find a solution.
- Fluctuating environments: A Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data, by discovering hidden rules or patterns.

Types of Machine Learning Systems

- **Supervised Learning**: You provide the solutions in the training set. Each training instance is input-output pair, where the output is what is desired for the input.
- An input is often called a set of **features** or **attributes**. An output is called a **label**.
- **Classification** Example – the email spam filter. The input of each training instance is an email (the sender, recipient, the email body); the label denotes the class: either spam or nonspam.

  For classification, the label is discrete.

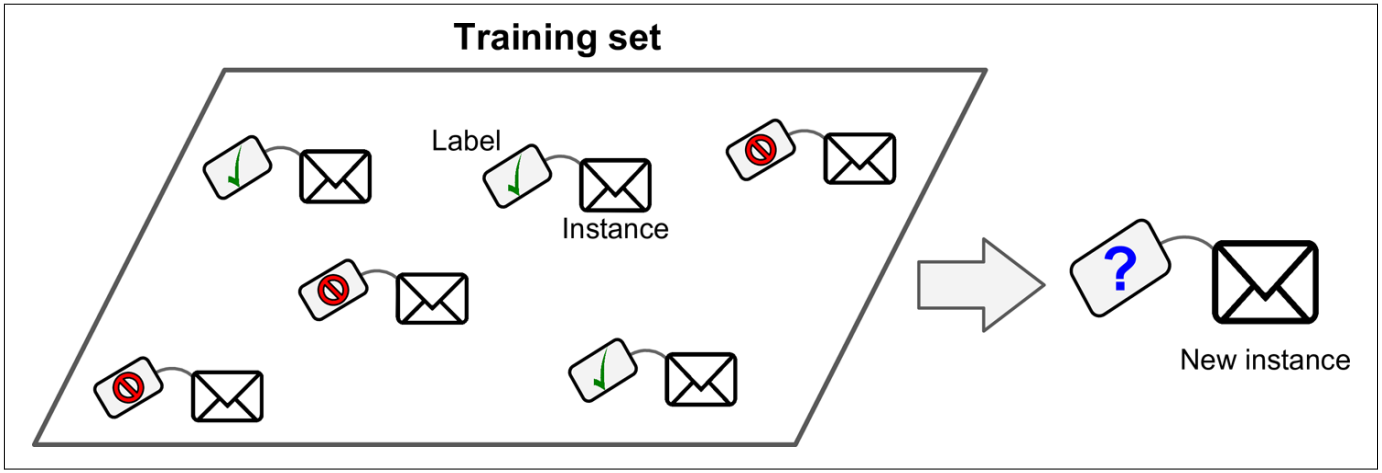


Figure 1: Supervised Learning Example: Spam Filter

- **Prediction** Example – predict a numeric value of the price of a car. For each training instance, the input features of a car are mileage, age, brand, etc; the label is the price of the car.

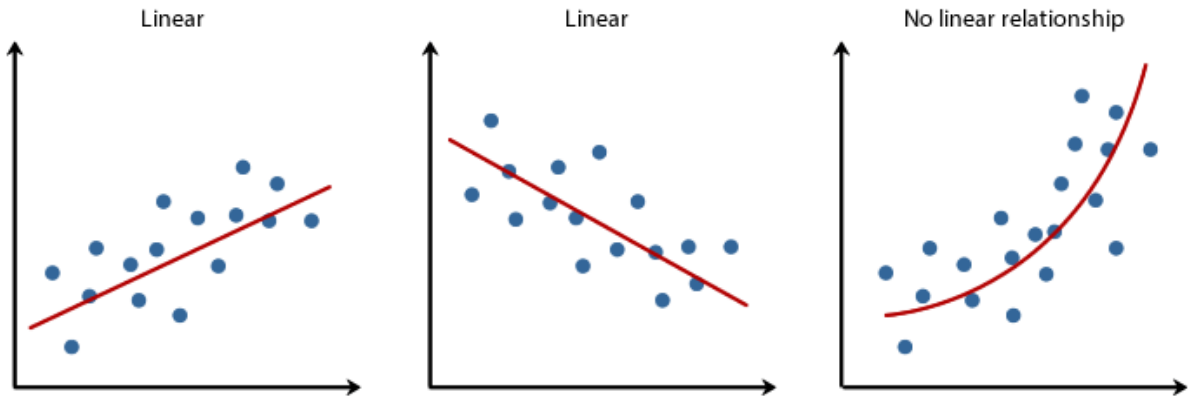
  In this context, the features are also called **predictors**.

  For prediction, the output is continuous (or nearly so).

- In prediction, the ML algorithm learns a (continuous) functional

dependency of the output vs. the input features.

In traditional statistics, this sort of task is called **regression**. The input is called **independent variables**, and the output is called **dependent** variable.



Most Important Supervised Learning Algorithms

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural Networks – This is a very large category. There are many different kinds of networks.

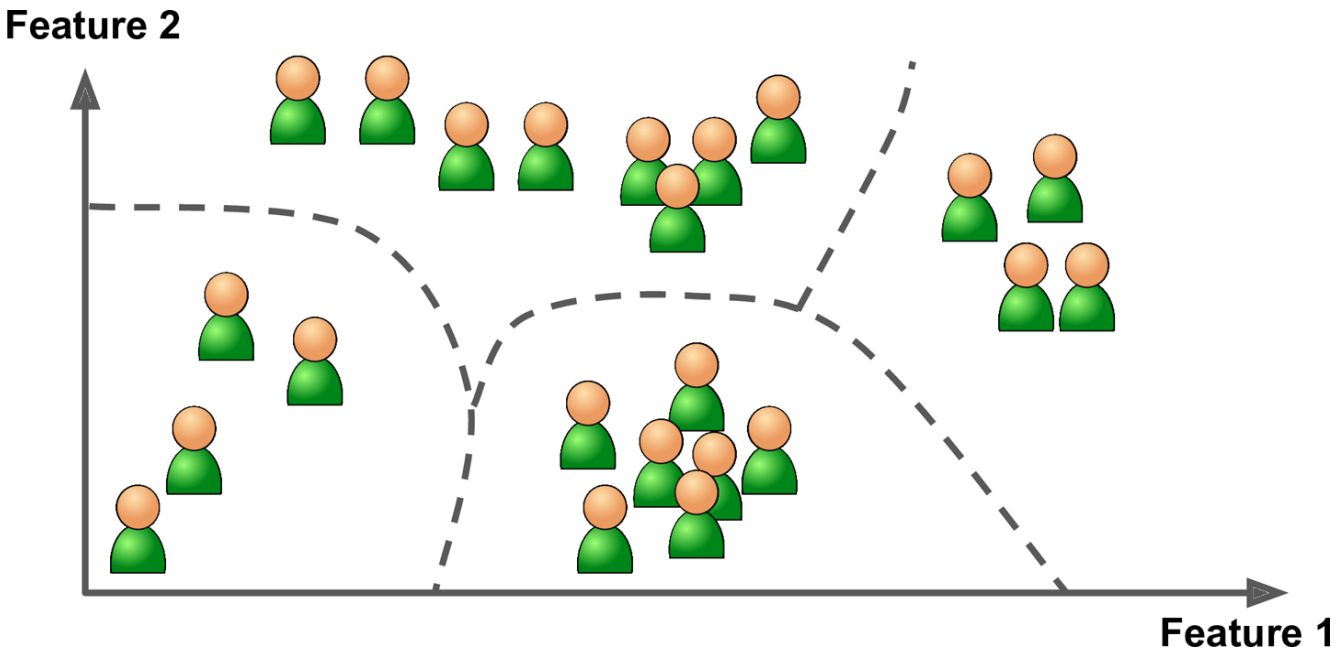
We will cover most of the above except neural networks.

Unsupervised Learning

- The training data is *unlabeled*. The system tries to learn without a teacher.
- Most important unsupervised learning algorithms:
- Clustering
  - K-Means*
  - DBSCAN
  - Hierarchical Cluster Analysis (HCA)
- Anomaly detection and novelty detection
  - One-class SVM
  - Isolation Forest
- Visualization and dimensionality reduction

  - Principal Component Analysis (PCA)*
  - Kernel PCA
  - Locally Linear Embedding (LLE)
  - t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
  - Apriori
  - Eclat
- We will cover them selectively.

Unsupervised Learning: Clustering



Unsupervised Learning: Dimensionality Reduction

- Reduce high-dimensional (usually raw) data into lower-dimensional features.

  For instance, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear.

  This is called **feature extraction**.

- Dimensionality reduction is often used as a pre-processing step.

  It is often a good idea to try to reduce the dimension of your training data using a dimensionality reduction algorithm before you feed it to another machine learning algorithm, including supervised learning algorithms.

Unsupervised Learning: Anomaly Detection

- Examples: detect unusual credit card transactions to prevent fraud, catch manufacturing defects, or automatically remove outliers from a dataset before feeding it to another learning algorithm.
- The training set consists of all (or nearly all) normal instances.
- When it sees a new instance, the system can tell whether it looks like a normal one or whether it is likely an anomaly.

Unsupervised Learning: Association Rule Learning

- Example: supermarket – Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another.
- Dig into large amounts of data and discover interesting relations between attributes.

Reinforcement Learning (RL)

- Example: AlphaGo; robots learning to walk
- The learning system is called an **agent**. It observes the state/environment, performs actions and gets rewards (negative reward is penalty).
- **Policy** or **Strategy**: Rules that govern what action to take for each observed state.
- The agent tries to learn the best policy based on performing actions and getting rewards many times.
- In AlphaGo, the agent analyze millions of games including self-playing games to learn a winning strategy.
- In RL, no labels as in supervised learning. But, also not completely unsupervised: there is reward.

- Agent needs to explore and get feedback (rewards).
- Reinforcement part: Reward or penalty obtained by an agent should reinforce its behavior in a positive or negative way.

Generalization: Instance-Based vs. Model-Based Learning

- Need to **generalize**: Most machine learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before.
- Having good performance on the training data is good, but insufficient; the true goal is to **perform well on new instances**.
- **Instance-based learning**: The system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them).

  Spam Filter Example: The system remembers the spams flagged by users. An email that is identical or similar to a known spam is flagged. A simple measure of similarity between two emails could be the count of words they have in common.

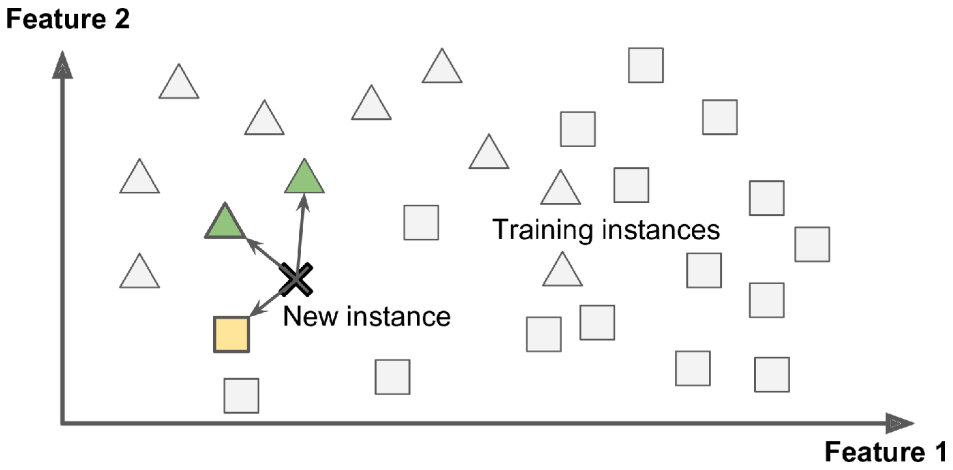


Figure 2: Instance-based learning: New instance is more similar to triangles

Model-Based Learning

- The system builds/learns a model from the examples.
- It uses the model to make prediction for new instances.
- What is a model? Usually, it is a mathematical relationship between the input and the output.

Formalization in Statistics

- $X \in \mathbb{R}^p$: *random* input vector
- $Y \in \mathbb{R}$: *random* output scalar
- $P(X, Y)$: joint distribution, which we don't really know.
  - $P(X, Y)$ is a lazy way to write the joint distribution of X and Y. More formally, we should write $P_{X,Y}(x, y)$, which means

$$P_{X,Y}(x, y) = \text{Prob}(X \leq x, Y \leq y).$$

- We observes realizations of (X, Y) or the training data: $(x_1, y_1), \ldots, (x_N, y_N)$. We wish to learn $P(X, Y)$.
- In the above, the form of the distribution function is not specified; i.e., every distribution function is allowed.
- More often, we restrict the distribution function to be from a given

family, e.g., Gaussian distribution functions; but, the parameters (e.g., the mean) of the function is unknown. The goal of learning is to determine the parameters based on the training data. We call this sort of models **parametric models**.

The joint distribution function is written as $P(X, Y; \theta)$. The expression of $P(X, Y; \theta)$ is known; however, it contains the unknown parameters θ (a vector in general).

- Much of probabilistic machine learning is about learning θ in $P(X, Y; \theta)$.
- To make prediction for a new input: Suppose the learning algorithm determined that the true parameter is $\hat{\theta}$. For prediction or classification, if we have a new instance with input $X = x$, we can compute the conditional probability $P(Y = y|X = x; \hat{\theta})$ for each value of y. Then, we can choose the value y that maximizes the conditional probability as the predicted output.

- In Bayesian learning, θ is viewed as also random with a known distribution, which is called the **prior**. One can then compute $P(\theta|\mathcal{D})$, the **posterior** distribution, where $\mathcal{D}$ denotes the training data.

  For prediction, one computes $P(Y = y|X = x, \mathcal{D})$ for different values of y and use the maximizing value of y as the predicted output. The knowledge of $P(\theta|\mathcal{D})$ usually shows up in the computation of $P(Y = y|X = x, \mathcal{D})$.

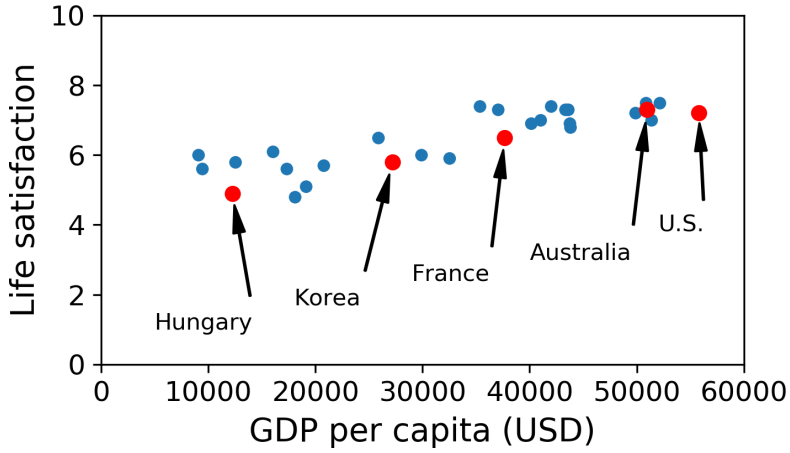
Formalization by Functional Approximation

- Very often, we like to think X and Y are functionally related by $Y = f(X)$, which isn't necessarily true, but can be useful for prediction or classification. In this case, we wish to learn f based on the training data.
- Since the training data contains noise, the learned function is $\hat{f}$, which in general is different from f.
- If we have $\hat{f}$, given another input vector x_0, we can predict the output $\hat{y}_0 = \hat{f}(x_0)$.
- $\hat{y}_0$ is usually not the same as the true output y_0. We need to worry about how good the prediction is.
- Usually, the function f is in parametric form, with unknown parameters θ (a vector).

- Example: Linear Regression

Table 1-1. Does money make people happier?

Country	GDP per capita (USD)	Life satisfaction
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2



- We wish to fit a linear model:

$$y = \theta_0 + \theta_1 x$$

y: output; life satisfaction

x: input; GDP per capita

Linear regression: Find the parameters θ_0 and θ_1 that minimize the square error loss function.

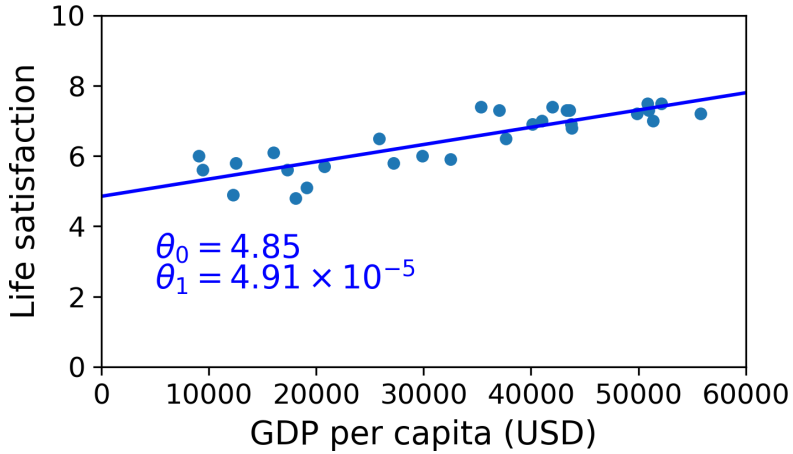


Figure 3: Linear regression results

Neural Networks

- Much of **neural networks** takes this functional approximation viewpoint. A neural network can be viewed as a function that maps the input to the output.

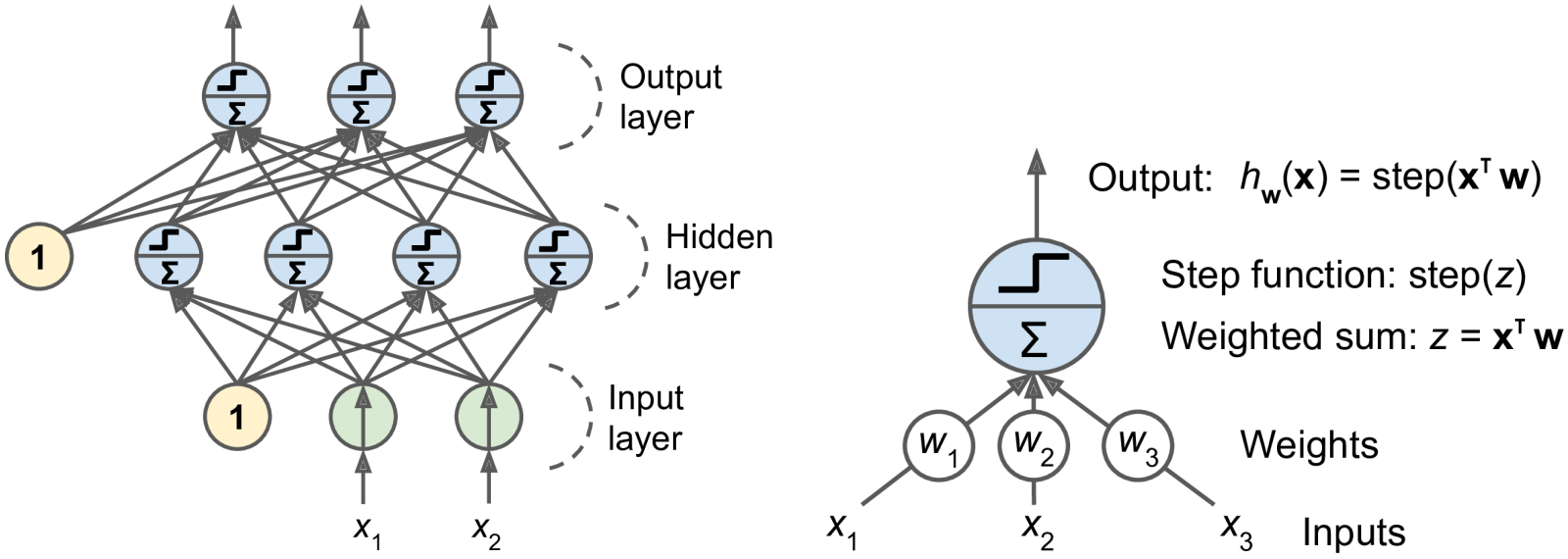


Figure 4: Left: Multilayer Perceptron with two inputs and three outputs. Right: Zooming in one neuron in the hidden layer.

In the above, the edges have unknown weights $\mathbf{w} = (w_1, w_2, w_3)^T$, which are not shown in the left figure, but shown in the right figure. The unknown weights $\mathbf{w}$ need to be learned by training.

- Each neuron first performs a weighted sum:

$$z = \mathbf{x}^T \mathbf{w} = w_1 x_1 + w_2 x_2 + w_3 x_3.$$

Then, the weighted sum is passed to the step function step(z), which is defined as

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0. \end{cases}$$

- In the case of deep neural networks, the number of inputs (and possibly also the number of outputs) can be very large – thousands or more. The number of layers is more than 100. As a result, the number of parameters is very large.

The goal of learning is again to learn the (many) parameters based on

training data.

- One can conjecture that in many real-world situations, the input and output are related through some **very complex** function f, i.e., $Y = f(X)$.
- One can approximate any function f closely by increasing the size of the neural network.

Relate back to Probabilistic Models

- If X and Y are truly related through $P(X, Y)$, then given $X = x_0$, Y is a random variable fully described by the conditional probability $P(Y|X = x_0)$. The prediction using $\hat{y}_0 = f(x_0)$ is a drastic simplification
  - because, given $X = x_0$, Y is random rather than fixed.
- Sometimes, X and Y may be related by $Y = f(X) + \varepsilon$, where ε is random. Even in that situation, it is incorrect to say $Y = f(X)$.

More Details about Model-Based Learning

Square Error Loss

- Suppose the underlying joint distribution function is $P(X, Y)$.
- Suppose we insist on predicting Y by a function of the input f.
  - To be more clear, we should have writen f as $\hat{f}$. But, we omit the hat for simplicity in notation.
- The **predicted output** is written as $\hat{Y}$, where $\hat{Y} = f(X)$.
- Y and $\hat{Y}$ are not the same in general, and they are both random.
- Question: What should f be? We need a **loss function** $L(Y, \hat{Y})$.
- It is common to use the **square error loss**

$$L(Y, \hat{Y}) = L(Y, f(X)) = (Y - f(X))^2.$$

- The expected loss (or expected prediction error EPE) is

$$EPE(f) = E[L(Y, f(X))] = E[(Y - f(X))^2].$$

- The expectation is taken with respect to $P(X, Y)$, which is unknown. To be clear, we may write $E_{X,Y}$.

What f minimizes $EPE(f)$ above?

Answer: the conditional expectation $E[Y|X]$, which is a function of X.

$$EPE(f) = E_{X,Y}[(Y - f(X))^2] = E_X E_{Y|X}[(Y - f(X))^2|X]$$

- Here, when computing the expectation $(Y - f(X))^2$ with respect to the joint distribution of (X, Y), we do the usual trick of '**conditioning on something first**'. Here, we first condition on X and compute the expectation $(Y - f(X))^2$ with respect to the conditional distribution of Y given X. We are left with a function of X, $h(X) \triangleq E[(Y - f(X))^2|X]$; $h(X)$ is random.
- We then compute the expectation of $h(X)$ with respect to the distribution of X.

What f minimizes $EPE(f)$ above?

At each $X = x$,

$$E_{Y|X}[(Y - f(X))^2|X = x] = E_{Y|X}[(Y - f(x))^2|X = x],$$

f is unknown at this point. Note that for any function f, when x is given, $f(x)$ is just a number.

We can solve the following problem

$$\min_y E_{Y|X}[(Y - y)^2|X = x].$$

Let the minimum be denoted by y_x^*. There is a subscript x in y_x^* because the minimum depends on x.

We then have a function, denoted by f^*, $f^* : x \mapsto y_x^*$.

The function f^* must minimize $E_{X,Y}[(Y - f(X))^2]$ (over all possible

functions f) because

$$\begin{aligned} E_{X,Y}[(Y - f^*(X))^2] &= E_X E_{Y|X}[(Y - f^*(X))^2|X] \\ &\leq E_X E_{Y|X}[(Y - f(X))^2|X] \\ &= E_{X,Y}[(Y - f(X))^2]. \end{aligned}$$

Why the inequality: For two functions h and g with $h \leq g$, we have

$$E[h(X)] \leq E[g(X)],$$

because expectation is a weighted sum, or in general, an integral.

Now,

$$\begin{aligned} E_{Y|X}[(Y - y)^2|X = x] &= E_{Y|X}[Y^2 - 2yY + y^2|X = x] \\ &= E_{Y|X}[Y^2|X = x] - 2yE_{Y|X}[Y|X = x] + y^2. \end{aligned}$$

By taking the derivative with respect to y, we see that the minimum y_x^* is equal to $E_{Y|X}[Y|X = x]$, the conditional expectation of Y given $X = x$.

We then have $f^*(x) = E_{Y|X}[Y|X = x]$.

Conclusion: We see that the function $f(X)$ that minimizes $EPE(f)$ is equal to $E[Y|X]$, the conditional expectation.

This is not surprising: At each $X = x$, the best predictor/estimator is the mean; but it has to be the conditional mean, conditional on $X = x$, since that information is given.

$E[Y|X]$ is the 'weighted' average of Y for each given X, where the weight is the conditional distribution $P(Y|X)$.

Recall that here we assume $P(X, Y)$ is known, but in reality, it is not known.

Side Note

- Formula: $E_{X,Y}[g(X,Y)] = E_X E[g(X,Y)|X]$.
- Proof for discrete random variables: Suppose $p_{X,Y}$ is the joint probability mass function and p_X is the marginal probability mass function for X.

$$\begin{aligned} E_{X,Y}[g(X,Y)] &= \sum_{x,y} g(x,y) p_{X,Y}(x,y) \\ &= \sum_x \sum_y g(x,y) p_{Y|X}(y|x) p_X(x) \\ &= \sum_x E_{Y|X}[g(x,Y)|X=x] p_X(x) \\ &= E_X E_{Y|X}[g(X,Y)|X]. \end{aligned}$$

Random Sample

- In reality, we do not know $P(X, Y)$. We observe the training data $(x_1, y_1), \ldots, (x_N, y_N)$.
- **Usual Assumption:** Each (x_i, y_i) is drawn from the distribution $P(X, Y)$. Different (x_i, y_i)'s are drawn independently from each other.

  That is, $(x_1, y_1), \ldots, (x_N, y_N)$ is realization of the underlying IID random variables $(X_1, Y_1), \ldots, (X_N, Y_N)$, each distributed as $P(X, Y)$
- We call $(X_1, Y_1), \ldots, (X_N, Y_N)$ a **random sample**, or simply a **sample**.
- The training data $(x_1, y_1), \ldots, (x_N, y_N)$, once given, are not random. However, we might be given different training data at different time.

- When we write down a machine learning algorithm, the algorithm is expressed as a sequence of manipulations of the random sample $(X_1, Y_1), \ldots, (X_N, Y_N)$.
- The algorithm is fixed, but the training data may be different for different runs of the algorithm.
- **Key:** Whatever we will learn or compute from the sample $(X_1, Y_1), \ldots, (X_N, Y_N)$ is random. We can talk about the mean and variance of the learned result.

- The literature and textbooks often abuse the notation and use $(x_1, y_1), \ldots, (x_N, y_N)$ to represent the random sample. We will often do the same.

Supervised Learning

Suppose we insist on predicting Y based on X, i.e., $\hat{Y} = f(X)$, and we know that in principle the best prediction function is $f(X) = E[Y|X]$ (assuming the goal is to minimize the square error loss).

However, we cannot compute $E[Y|X]$ because we don't have the full knowledge of the distribution $P(X, Y)$. We only know the training data, which is a sequence of independent realizations of (X, Y), which give us some information about $P(X, Y)$.

Question: How do we figure out such f based on the training data (in general, a sample)?

Answer: We will do 'hackish' things (like in many learning algorithms), and maybe later, try to improve or say something more definitive.

Much of statistical learning is about improvement and/or saying something more definitive.

One Route: Parametric Models

We consider a class of functions indexed by a set of parameters, θ (in general a vector), $\{g_\theta\}_{\theta \in \Theta}$, where Θ is the set of allowed parameters.

We assume $f = g_\theta$ for some θ and will find the best θ, denoted by $\hat{\theta}$, based on the training data.

Such $\hat{\theta}$ must be computed based on the training data $(x_1, y_1), \ldots, (x_N, y_N)$. Therefore, $\hat{\theta}$ can be viewed as being random.

We will develop this further.

p-Dimensional Input Vector, Scalar Output

Suppose the original X is a p-dimensional vector and we write $X = (X_1, \ldots, X_p)^T$. Each X_j is a component of X instead of a sample.

Warning: We are switching notations here.

Suppose Y is a scalar random variable.

In the training data, each x_i is a p-dimensional vector, and each y_i is scalar.

Example - Linear (Affine) Functions

Suppose we have scalar variables $z_1, \ldots, z_p$ and scalar parameters $\theta_0, \theta_1, \ldots, \theta_p$.

For convenience, we define $z = (z_1, \ldots, z_p)^T$, $\bar{z} = (1, z_1, \ldots, z_p)^T$ and $\theta = (\theta_0, \theta_1, \ldots, \theta_p)^T$.

A function of the form $\bar{z}^T\theta = \theta_0 + \sum_{j=1}^{p} z_j\theta_j$ is affine in z and linear in θ. We will see the linearity in θ is more important.

The family of functions are $g_\theta(z) = \bar{z}^T\theta$, where $\theta \in \Theta = \mathbb{R}^{p+1}$.

Consider the special case where $E[Y|X = z]$ actually has the form $\bar{z}^T\theta$ (θ unknown). Then, we find θ to minimize $E_{X,Y}[(Y - \bar{X}^T\theta)^2]$, where $\bar{X} = (1, X_1, \ldots, X_p)^T$. Suppose the minimum is denoted by θ^*.

Earlier, we saw that the function f^* that minimizes $EPE(f) = E_{X,Y}[(Y - f(X))^2]$ is $E[Y|X]$. There, the minimization is over the function f.

In our special case, $E[Y|X]$ has the form $\bar{X}^T\theta$. Therefore, minimizing over the function f is the same as minimizing over θ.

We conclude that $E[Y|X]$ is exactly $\bar{X}^T\theta^*$ in this special case

(because, when viewed as a function of X, $\bar{X}^T\theta^*$ is the function that minimizes $EPE(f)$ out of all functions of the form $\bar{X}^T\theta$).

In linear regression, one assumes that $E[Y|X = z]$ is linear or approximately linear in z, i.e., has the form $\bar{z}^T\theta$ (actually, affine in z). This assumption may be problematic (see later).

The other big problem is that we can't compute $E[(Y - \bar{X}^T\theta)^2]$ because the probability distribution $P(X, Y)$ is not completely known.

Solution to the second problem: We can approximate $E[(Y - \bar{X}^T\theta)^2]$ using the training data and then minimize over θ. That is,

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - (\theta_0 + \sum_{j=1}^{p} x_{ij}\theta_j))^2. \tag{1}$$

In other words, we are using the empirical distribution in place of the true distribution.

This is the method of **least squares**.

Drawbacks: Consider the special case where X and Y are related by $Y = h(X) + \varepsilon$ for some function h, and the random error ε is independent of X and has $E[\varepsilon] = 0$. Then, $E[Y|X] = h(X)$.

If h is not a linear (or affine) function, then $\bar{z}^T\theta$ may not be close to $h(z)$ for any θ. Then, there seems to be little reason that the method of least square will give us good prediction.

What do we even mean by that? See later.

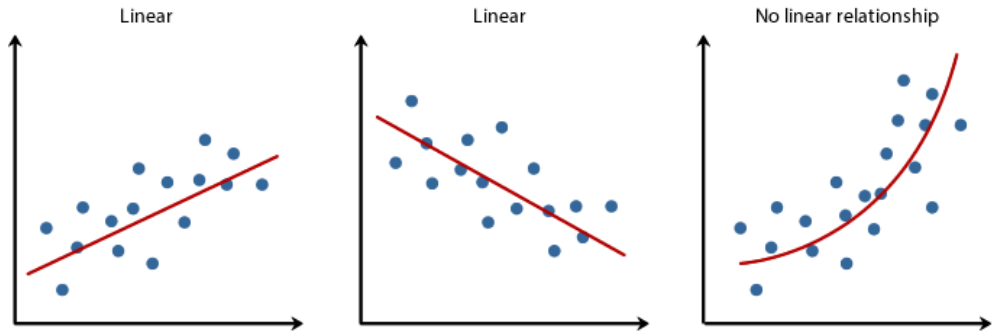


Figure 5: h is not linear in the third case. Fitting a linear function to the data may not be great.

Linear Regression

We will solve (1). Let $\mathbf{y} = (y_1, \ldots, y_N)^T$.

Let $\mathbf{X}$ be the $N \times (p+1)$ matrix based on the input vectors:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{Np} \end{bmatrix}$$

The function to be minimize in (1) can be written as

$$RSS(\theta) \triangleq (\mathbf{y} - \mathbf{X}\theta)^T(\mathbf{y} - \mathbf{X}\theta),$$

where RSS stands for residual sum of squares.

(The factor $1/N$ doesn't matter and it is discarded.)

Then, the gradient and Hessian of RSS are:

$$\nabla RSS(\theta) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\theta)$$
$$\nabla^2 RSS(\theta) = 2\mathbf{X}^T\mathbf{X}.$$

Let us assume $\mathbf{X}$ has full column rank and therefore $\mathbf{X}^T\mathbf{X}$ is positive definite. Then, $RSS(\theta)$ is convex. Its minimum is obtained by solving $\nabla RSS(\theta) = 0$. We get the unique solution:

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{2}$$

We now can define the prediction function $\hat{f}$. At any input vector x_0,

$$\hat{f}(x_0) = (1, x_0^T)\hat{\theta}.$$

where $(1, x_0^T)$ is the row vector by concatenating 1 with the x_0^T.

The predicted output is denoted by $\hat{y}_0$, where $\hat{y}_0 = \hat{f}(x_0)$.

Interpretations and Discussions

- $\hat{\theta}$ in (2) depends on the **entire** training data $(x_1, y_1), \ldots, (x_N, y_N)$.
- When the training data are viewed as a random sample, we can say $\hat{\theta}$ is a random vector. The distribution of the random sample is the joint distribution $P((X_1, Y_1), \ldots, (X_N, Y_N))$, where (X_i, Y_i)'s are IID, each distributed as (X, Y).
- For simplicity, let $\mathcal{T}$ denote the distribution of the random sample.
- At each input vector x_0, the predicted output $\hat{y}_0 = \hat{f}(x_0)$ depends on the entire training data and $\hat{y}_0$ is random.
- We can ask the mean (vector) and covariance (matrix) of $\hat{\theta}$, and the mean and variance of $\hat{y}_0$.

- The true output Y_0 is also random, with the distribution $P(Y|X = x_0)$, which can be computed from the joint distribution $P(X, Y)$.
- We can ask about the bias in the prediction (aka estimate): $E[Y_0 - \hat{f}(x_0)|X = x_0]$, where the expectation is taken over $\mathcal{T}$ and $P(Y|X = x_0)$.

  This expectation is the difference between the mean of the estimate and the true mean of the output, given $X = x_0$.

  The estimator/prediction is said **unbiased** if $E[Y_0 - \hat{f}(x_0)|X = x_0] = 0$.

  A large-magnitude bias is not good. Zero bias is not necessarily good enough. Why?
- A better performance measure is the prediction error at x_0

$$Err(x_0) = E[(Y_0 - \hat{f}(x_0))^2|X = x_0].$$

Bias and Variance of Estimator

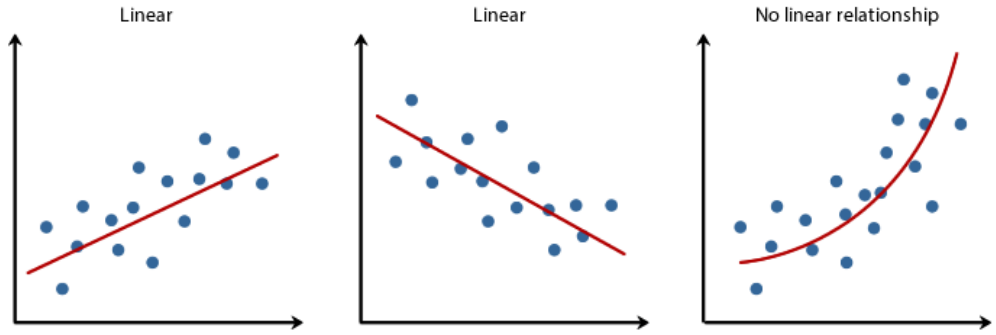
- Special Case: $Y = f(X) + \varepsilon$, where ε is an independent noise with 0 mean and variance σ^2.
- f is not necessarily linear. The estimator $\hat{f}$ is not necessarily from the method of least squares.
- The bias is: $E[Y_0 - \hat{f}(x_0)|X = x_0] = f(x_0) - E_{\mathcal{T}}[\hat{f}(x_0)]$.
- Will show: The prediction error is the sum of an irreducible component σ^2, the squared bias and the variance of the estimate.
- There is often a trade-off between the bias and the variance in different estimates/models/prediction methods.

$$
\begin{aligned}
Err(x_0) =& E[(Y_0 - \hat{f}(x_0))^2 | X = x_0] \\
=& E[(Y_0 - f(x_0) + f(x_0) - \hat{f}(x_0))^2 | X = x_0] \\
=& E[(\varepsilon + f(x_0) - \hat{f}(x_0))^2 | X = x_0] \\
=& E[\varepsilon^2 + 2\varepsilon(f(x_0) - \hat{f}(x_0)) + (f(x_0) - \hat{f}(x_0))^2 | X = x_0] \\
=& \sigma^2 + E[(f(x_0) - \hat{f}(x_0))^2] \qquad (3) \\
=& \sigma^2 + E[(f(x_0) - E\hat{f}(x_0) + E\hat{f}(x_0) - \hat{f}(x_0))^2] \\
=& \sigma^2 + (f(x_0) - E\hat{f}(x_0))^2 + E(E\hat{f}(x_0) - \hat{f}(x_0))^2 \\
& - 2(f(x_0) - E\hat{f}(x_0))E[E\hat{f}(x_0) - \hat{f}(x_0)] \\
=& \sigma^2 + \text{Bias}^2(\hat{f}(x_0)) + \text{Var}(\hat{f}(x_0)).
\end{aligned}
$$

Starting from (3), the expectation is taken with respect to $\mathcal{T}$.

- A small squared bias means that, when averaged over different possible training datasets, the "average" (i.e., mean) prediction $E[\hat{f}(x_0)]$ is close to the true output $f(x_0)$.
- Small variance means that the fluctuation of the predicted output $\hat{f}(x_0)$ around its mean $E[\hat{f}(x_0)]$ is small, where the fluctuation occurs when different training datasets are used.
- A good learning algorithm should have a small prediction error, which means it should have a small squared bias and a small variance.
- In practice, most algorithms have a fair amount of prediction error. When examined further, some algorithms have a small squared bias but a large variance; some are the opposite.
- Example: Suppose our algorithm has zero variance. That implies we use a constant c as the predicted output. The mean $E[\hat{f}(x_0)]$ is also equal to c. The squared bias is equal to $(f(x_0) - c)^2$. This is fine if we are lucky that c is such that $f(x_0) = c$. But, this may not happen.

- The discussion here assumes $X = x_0$ for a given x_0. A small prediction error at a particular input x_0 is not enough.



- For the third case in the figure, fitting a linear function will work fine for some input value, but not others.
- We need to worry about if a learning algorithm should be adaptive to each x_0.

Example - k-Nearest Neighbor

Up to now – a summary:

Recall that, in general, the optimal predictor $E[Y|X]$ minimizes the expected prediction error $EPE(f) = E_{X,Y}(Y - f(X))^2$.

However, since we don't have full knowledge of $P(X, Y)$, we can't compute $E[Y|X]$; nor can we compute $EPE(f)$ for any given f.

But, we can think about estimating/approximating $E[Y|X]$ using the training data.

Now, consider the special case $Y = f(X) + \varepsilon$, where f is fixed but unknown. In this case, $E[Y|X] = f(X)$.

Recall the least-squares method. Instead of the optimal predictor $f(X) = E[Y|X]$, we use a function of the form $\hat{f}(X) = \bar{X}^T\theta$ as a

predictor, with an appropriately chosen θ. In other words, we are using a linear function to approximate the unknown f.

Naturally, we want to choose the θ that minimizes the expected prediction error $EPE(\hat{f}) = E_{X,Y}(Y - \hat{f}(X))^2$, as that is the performance metric. But, we can't do that without knowing $P(X, Y)$.

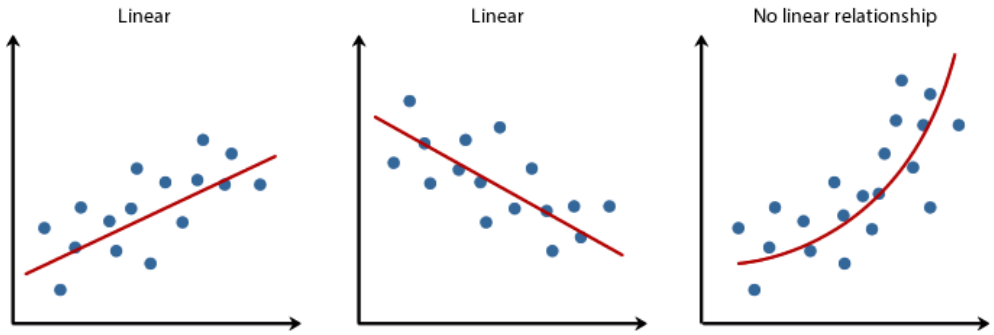
The next best thing to do is to choose θ by minimizing an **estimate** of the expected prediction error $EPE(\hat{f})$ using the training data.

We use the sum of squares from the training data, then divided by N, as an estimate of $EPE(\hat{f})$ and minimize over θ.

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - (1, x_i^T)\theta)^2.$$

The optimal $\hat{\theta}$ uses information from all the training data.

The bias at $X = x_0$ is $E[Y_0 - (1, x_0^T)\hat{\theta}|X = x_0] = f(x_0) - E[(1, x_0^T)\hat{\theta}]$.



If f is significantly different from any linear function (see the third figure; the solid line is f), the least-squares predictor will have significant bias at some input vectors, and hence, significant prediction error.

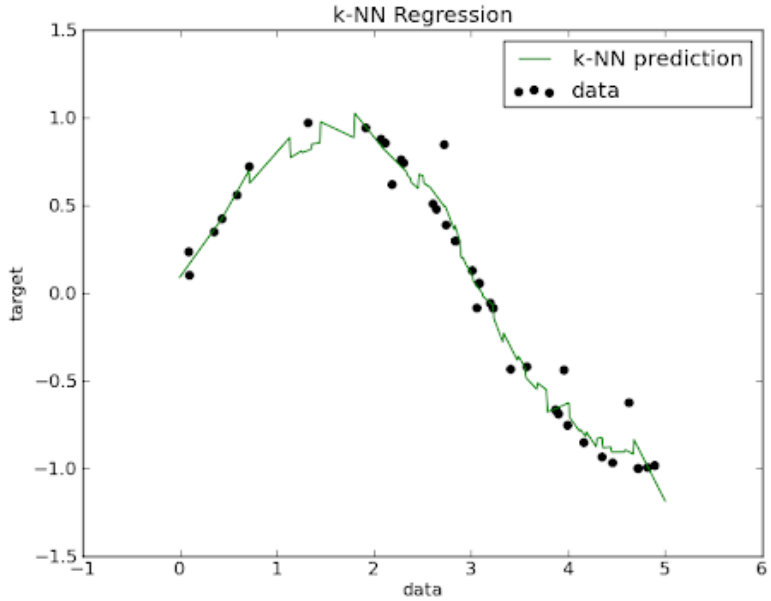
End of summary

We next consider the k**-nearest-neighbor** predictor/estimator.

$$\hat{f}(x_0) = \text{Ave}(y_i | x_i \in N_k(x_0)).$$

$N_k(x_0)$: the set containing k input vectors (in the sample) closest to x_0;

Ave: the average.



$\hat{f}(x_0)$ is used to directly approximate optimal predictor $E[Y|X = x_0]$:

- expectation is approximated by averaging over the training data
- conditioning on a point x_0 is relaxed to conditioning on some region 'close' to x_0.

$\hat{f}(x_0)$ fits this data well.

It lacks smoothness, which may suggest large variances (assuming the true $f(x)$ is a smooth function of the input x).

A linear function will not work well for this data.

k-nearest-neighbor reminds us instance-based learning. So, the separation between instance-based learning and model-based learning is not that strong.

Curse of Dimensionality

If all the input vectors in $N_k(x_0)$ are very close to x_0, taking the average of the corresponding outputs should approximate $E[Y|X = x_0]$ well.

If the input has high dimensional, the input vectors in $N_k(x_0)$ are not necessarily close to x_0 at all $\rightarrow$ the 'curse of dimensionality'.

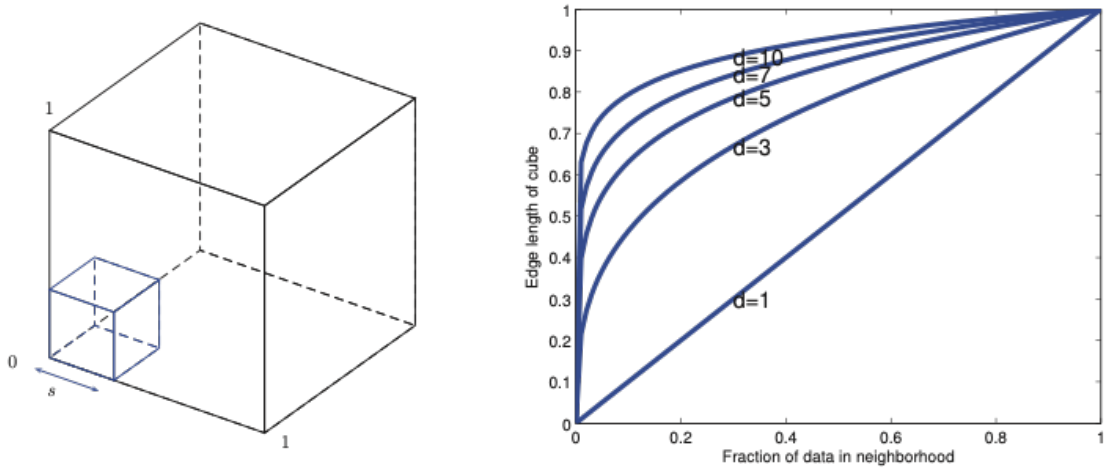


Figure 6: Cure of dimensionality: Suppose points are uniformly distributed in the d-dimensional unit cube. Consider the case where the dimensionality d is large. The smaller cube has a small volume even when its length is relatively large, e.g., $s = 0.8$. In that case, the smaller cube contains only a small fraction of points, even though some points in it is fairly far from the point 0.

To have k points in a very high-dimensional cube, the sides of the cube may not be small, which implies that some points in the cube may be far apart.

Linear Models vs. k-Nearest Neighbor

- For simplicity of discussion, suppose $Y = f(X) + \varepsilon$.
- We wish to approximate f by constructing some $\hat{f}$ based on the training data, which can be viewed as a noisy representation of f.
- In the linear models, $\hat{f}(z)$ is found in the class $(1, z^T)\theta$ by the method least squares.
- If $f(z)$ is far from linear, none of the of linear functions $(1, z^T)\theta$ may approximate f well.
- Least squares and linear model use the training data in 'global' fashion.
- When $f(z)$ is complex looking, the k-nearest-neighbor method may adapt to $f(z)$ better.
- It uses the training data in a local fashion.

- But, it has problems in dealing with high-dimensional input; may require a large amount of training data (so that $N_k(x_o)$ contains enough points); the resulting $\hat{f}$ lacks smoothness.
- Many popular learning methods are variants of the above two methods. They often address the issues in either method.
- Examples: kernel methods, local regression, basis expansion.