CSC 311: Introduction to Machine Learning

Lecture 3 - Ensemble methods I & Linear Regression

Murat A. Erdogdu & Richard Zemel

University of Toronto

Announcements

- Homework 1 is posted! Deadline Oct 2, 23:59.
- TA office hours are announced on the course website.

Today

- Bias-Variance decomposition
- Ensemble methods I: Bagging, Random Forests
- Linear regression

Bias-Variance decomposition: Loss Functions

- A **loss function** $L(y, t)$ defines how bad it is if, for some example x, the algorithm predicts y, but the target is actually t.
- Example: **0-1 loss** for classification

$$L_{0-1}(y, t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{if } y \neq t \end{cases}$$

  - Averaging the 0-1 loss over the training set gives the **training error rate**, and averaging over the test set gives the **test error rate**.

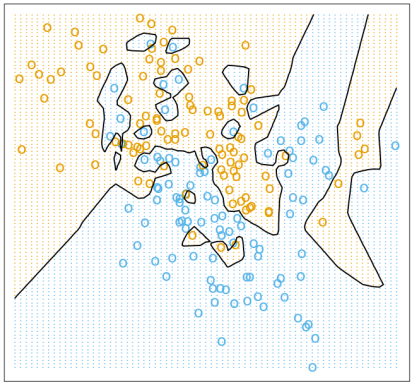
- Example: **squared error loss** for regression

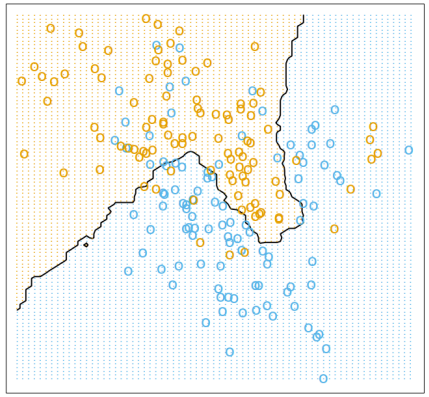
$$L_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2$$

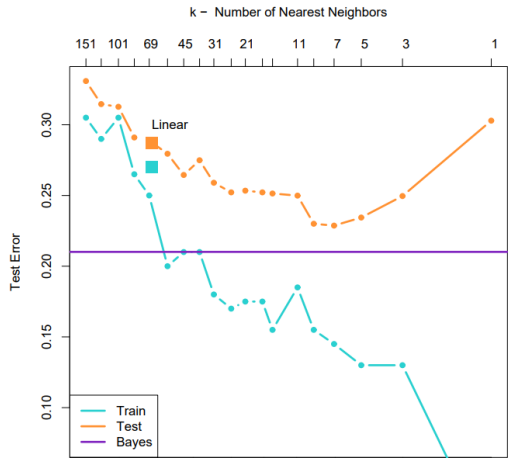
  - The average squared error loss is called **mean squared error (MSE)**.

Bias-Variance Decomposition

- Recall that overly simple models underfit the data, and overly complex models overfit.



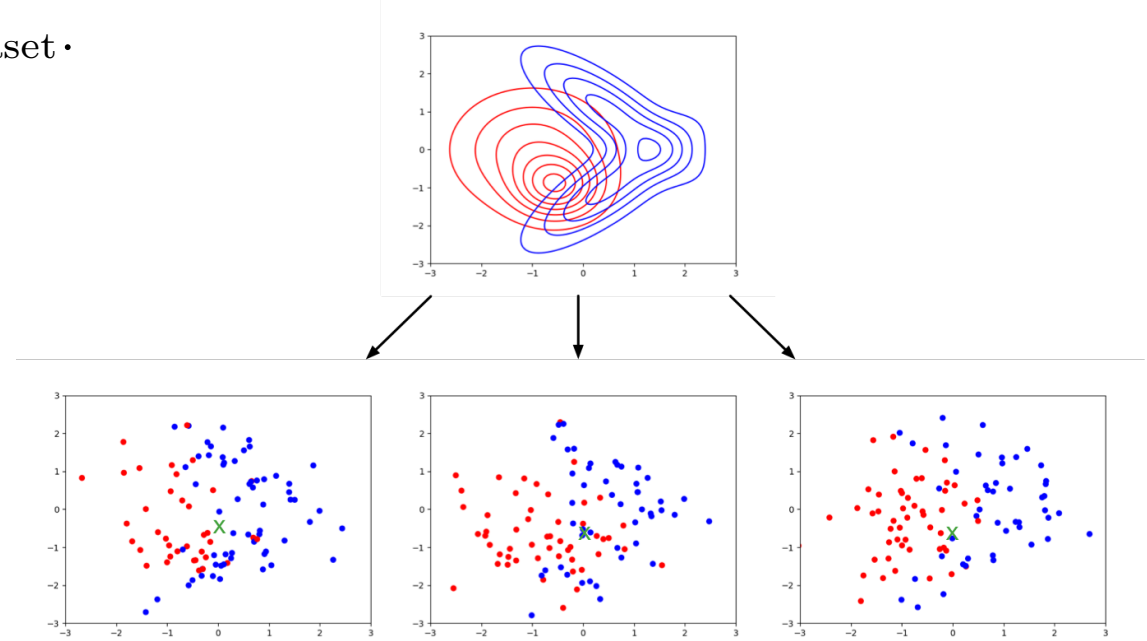




- We can quantify this effect in terms of the **bias/variance decomposition**.
- Bias and variance of what?

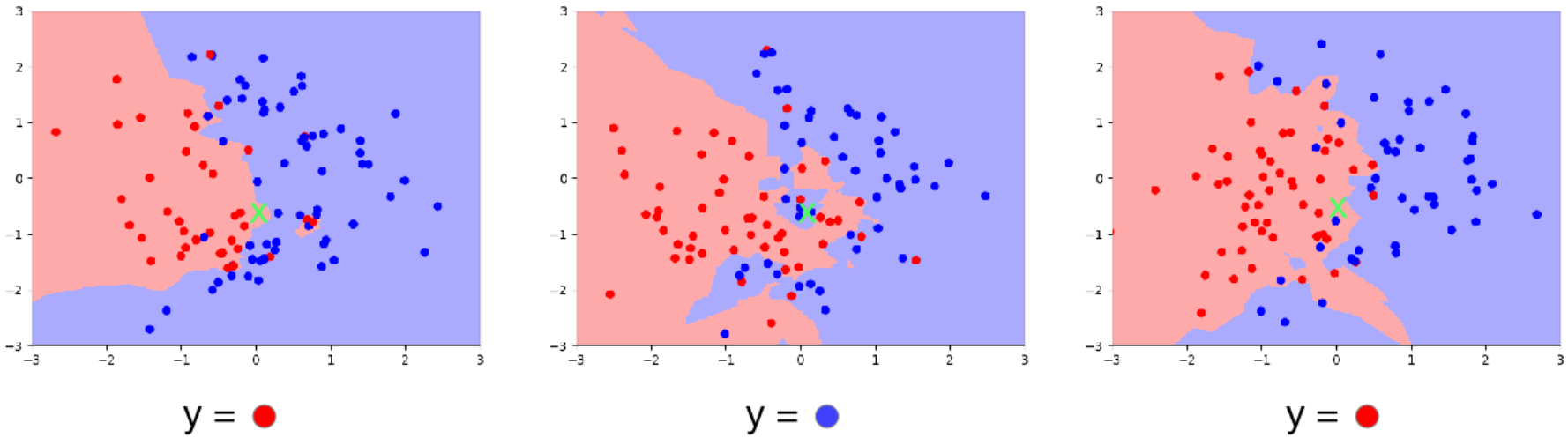
Bias-Variance Decomposition: Basic Setup

- Suppose the training set $\mathcal{D}$ consists of N pairs $(\mathbf{x}^{(i)}, t^{(i)})$ sampled **independent and identically distributed (i.i.d.)** from a **sample generating distribution** p_{sample}, i.e., $(\mathbf{x}^{(i)}, t^{(i)}) \sim p_{\text{sample}}$.
  - Let p_{dataset} denote the induced distribution over training sets, i.e. $\mathcal{D} \sim p_{\text{dataset}}$
- Pick a fixed query point $\mathbf{x}$ (denoted with a green x).
- Consider an experiment where we sample lots of training datasets i.i.d. from p_{dataset}.



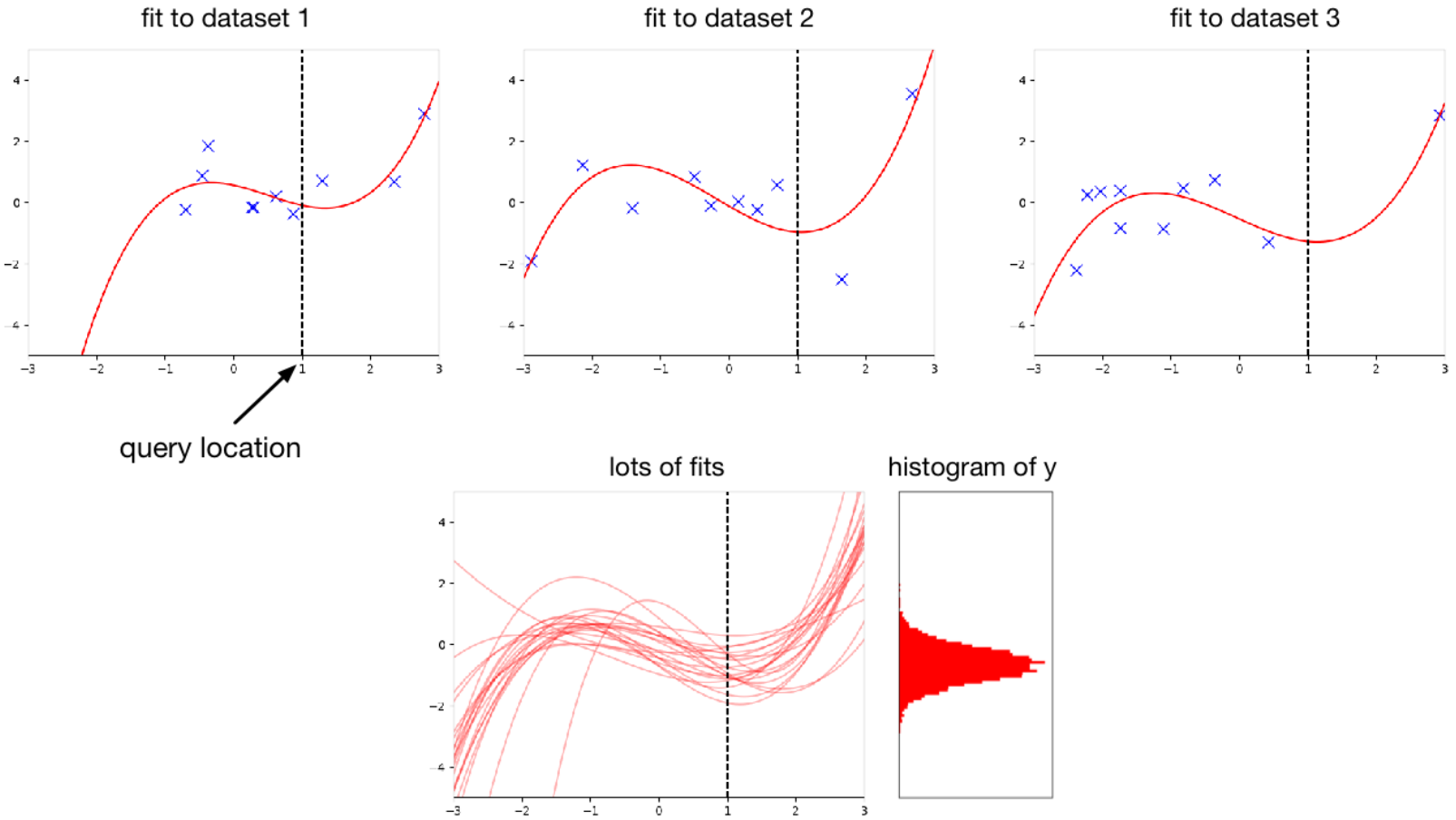
Bias-Variance Decomposition: Basic Setup

- Let's run our learning algorithm on each training set $\mathcal{D}$, producing a classifier $h_{\mathcal{D}}$
- We compute each classifier's prediction $h_{\mathcal{D}}(\mathbf{x}) = y$ at the query point $\mathbf{x}$.
- y is a random variable, where the **randomness comes from the choice of training set**
  - $\mathcal{D}$ is random $\implies$ $h_{\mathcal{D}}$ is random $\implies$ $h_{\mathcal{D}}(\mathbf{x})$ is random



Bias-Variance Decomposition: Basic Setup

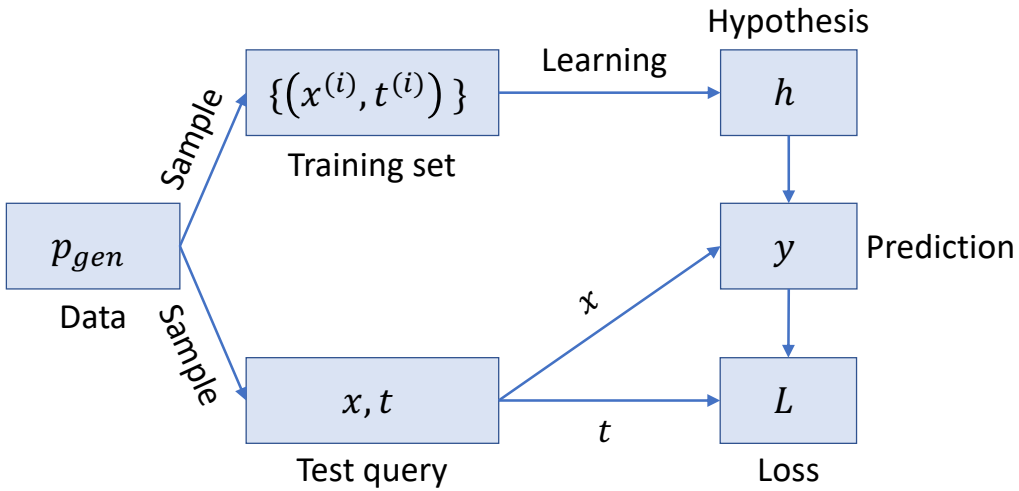
Here is the analogous setup for regression:



Since $y = h_{\mathcal{D}}(\mathbf{x})$ is a random variable, we can talk about its expectation, variance, etc. over the distribution of training sets p_{dataset}

Bias-Variance Decomposition: Basic Setup

- Recap of basic setup:



- Assume (for the moment) that t is deterministic given x!
- There is a distribution over the loss at $\mathbf{x}$, with expectation $\mathbb{E}_{\mathcal{D} \sim p_{\text{dataset}}}[L(h_{\mathcal{D}}(\mathbf{x}), t)]$.
- For each query point $\mathbf{x}$, the expected loss is different. We are interested in quantifying how well our classifier does over the distribution p_{sample}, averaging over training sets: $\mathbb{E}_{\mathbf{x} \sim p_{\text{sample}}, \mathcal{D} \sim p_{\text{dataset}}}[L(h_{\mathcal{D}}(\mathbf{x}), t)]$.

Bias-Variance Decomposition

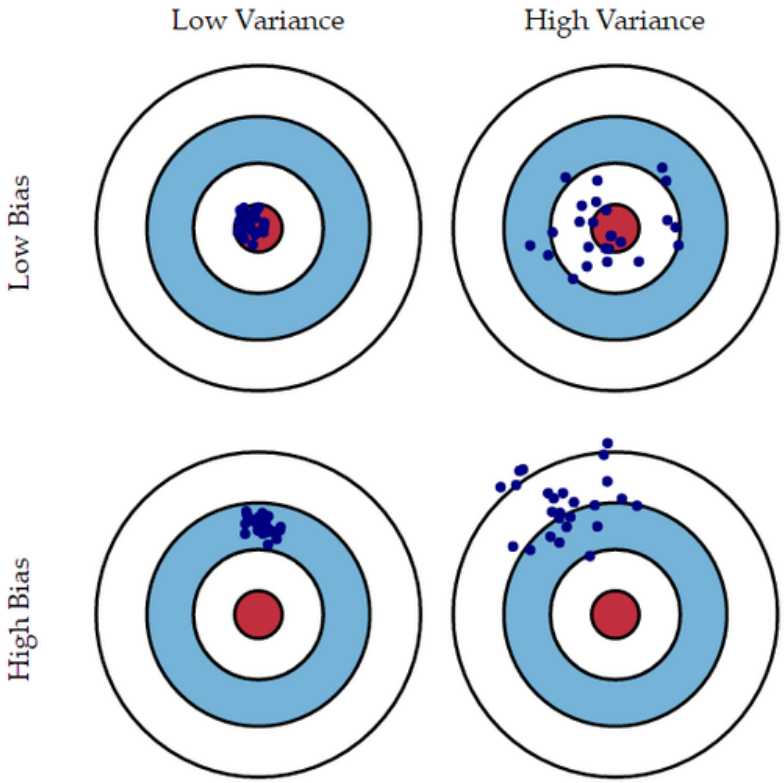
- For now, focus on squared error loss, $L(y,t) = \frac{1}{2}(y-t)^2$.
- We can decompose the expected loss (suppressing distributions $\mathbf{x}$, $\mathcal{D}$ drawn from for compactness) (using $\mathbb{E}[\mathbb{E}[X \mid Y]] = \mathbb{E}[X]$ in second step)

$$\begin{aligned}
\mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - t)^2] &= \mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}] + \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}] - t)^2] \\
&= \mathbb{E}_{\mathbf{x}}[\mathbb{E}_{\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}])^2 + (\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}] - t)^2 + \\
&\qquad 2(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}])(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}] - t) \mid \mathbf{x}]] \\
&= \underbrace{\mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}])^2]}_{\text{variance}} + \underbrace{\mathbb{E}_{\mathbf{x}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x}) \mid \mathbf{x}] - t)^2]}_{\text{bias}}
\end{aligned}$$

- Bias: On average, how close is our classifier to true target? (corresponds to underfitting)
- Variance: How widely dispersed are our predictions as we generate new datasets? (corresponds to overfitting)

Bias and Variance

- Throwing darts = predictions for each draw of a dataset



- What doesn't this capture?
- We average over points $\mathbf{x}$ from the data distribution

Bagging

Now, back to ensembles!
For now, we only consider bagging & random forests. We will talk about other ensemble methods such as boosting later in the course.

Bagging: Motivation

- Suppose we could somehow sample m independent training sets $\{\mathcal{D}_i\}_{i=1}^m$ from p_{dataset}.
- We could then learn a predictor $h_i := h_{\mathcal{D}_i}$ based on each one, and take the average $h = \frac{1}{m}\sum_{i=1}^m h_i$.
- How does this affect the terms of the expected loss?
  - **Bias: unchanged**, since the averaged prediction has the same expectation

$$\mathbb{E}_{\mathcal{D}_1,\ldots,\mathcal{D}_m \overset{iid}{\sim} p_{\text{dataset}}}[h(\mathbf{x})] = \frac{1}{m}\sum_{i=1}^m \mathbb{E}_{\mathcal{D}_i \sim p_{\text{dataset}}}[h_i(\mathbf{x})] = \mathbb{E}_{\mathcal{D}\sim p_{\text{dataset}}}[h_{\mathcal{D}}(\mathbf{x})]$$

  - **Variance: reduced**, since we're averaging over independent samples

$$\operatorname*{Var}_{\mathcal{D}_1,\ldots,\mathcal{D}_m}[h(\mathbf{x})] = \frac{1}{m^2}\sum_{i=1}^m \operatorname*{Var}_{\mathcal{D}_i}[h_i(\mathbf{x})] = \frac{1}{m}\operatorname*{Var}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})].$$

What if $m \to \infty$?

Bagging: The Idea

- In practice, we don't have access to the underlying data generating distribution p_{sample}.
- It is expensive to collect many i.i.d. datasets from p_{dataset}.
- Solution: **bootstrap aggregation**, or **bagging**.
  - Take a single dataset $\mathcal{D}$ with n examples.
  - Generate m new datasets, each by sampling n training examples from $\mathcal{D}$, with replacement.
  - Average the predictions of models trained on each of these datasets.

Bagging: The Idea

- Problem: the datasets are not independent, so we don't get the $1/m$ variance reduction.
  - Possible to show that if the sampled predictions have variance σ^2 and correlation ρ, then

$$\text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} h_i(\mathbf{x})\right) = \frac{1}{m}(1-\rho)\sigma^2 + \rho\sigma^2.$$

- Ironically, it can be advantageous to introduce *additional* variability into your algorithm, as long as it reduces the correlation between samples.
  - Intuition: you want to invest in a diversified portfolio, not just one stock.
  - Can help to use average over multiple algorithms, or multiple configurations of the same algorithm.

Random Forests

- **Random forests** = bagged decision trees, with one extra trick to decorrelate the predictions
- When choosing each node of the decision tree, choose a random set of d input features, and only consider splits on those features
- The main idea in random forests is to improve the variance reduction of bagging by reducing the correlation between the trees ($\sim \rho$).
- Random forests are probably the best black-box machine learning algorithm — they often work well with no tuning whatsoever.
  - one of the most widely used algorithms in Kaggle competitions

Bayes Optimality

- Let's return to quantifying expected loss and make the situation slightly more complicated (and realistic): what if t is not deterministic given $\mathbf{x}$? i.e. have $p(t|\mathbf{x})$
- We can no longer measure bias as expected distance from true target, since there's a distribution over targets!
- Instead, we'll measure distance from $y_*(\mathbf{x}) = \mathbb{E}[t \,|\, \mathbf{x}]$
  - This is the best possible prediction, in the sense that it minimizes the expected loss

Bayes Optimality

Want to show: $\text{argmin}_y \mathbb{E}[(y-t)^2 \,|\, \mathbf{x}] = y_*(\mathbf{x}) = \mathbb{E}[t \,|\, \mathbf{x}]$ (Distribution of $t \sim p(t|\mathbf{x})$)

- **Proof:** Start by conditioning on (fixing) $\mathbf{x}$.

$$\begin{aligned}
\mathbb{E}[(y-t)^2 \,|\, \mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2 \,|\, \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \,|\, \mathbf{x}] + \mathbb{E}[t^2 \,|\, \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \,|\, \mathbf{x}] + \mathbb{E}[t \,|\, \mathbf{x}]^2 + \text{Var}[t \,|\, \mathbf{x}] \\
&= y^2 - 2yy_*(\mathbf{x}) + y_*(\mathbf{x})^2 + \text{Var}[t \,|\, \mathbf{x}] \\
&= (y - y_*(\mathbf{x}))^2 + \text{Var}[t \,|\, \mathbf{x}]
\end{aligned}$$

- The first term is nonnegative, and can be made 0 by setting $y = y_*(\mathbf{x})$.
- The second term doesn't depend on y! Corresponds to the inherent unpredictability, or **noise**, of the targets, and is called the **Bayes error** or **irreducible error**.
  - This is the best we can ever hope to do with any learning algorithm. An algorithm that achieves it is **Bayes optimal**.

Bayes Optimality

- We can again decompose the expected loss, this time taking the distribution of t into account (check this!):

$$\mathbb{E}_{\mathbf{x},\mathcal{D},t|\mathbf{x}}[(h_{\mathcal{D}}(\mathbf{x}) - t)^2] =$$
$$\underbrace{\mathbb{E}_{\mathbf{x}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})] - y_*(\mathbf{x}))^2]}_{\text{bias}} + \underbrace{\mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})])^2]}_{\text{variance}} + \underbrace{\mathbb{E}_{\mathbf{x}}[\text{Var}[t \mid \mathbf{x}]]}_{\text{Bayes}}$$

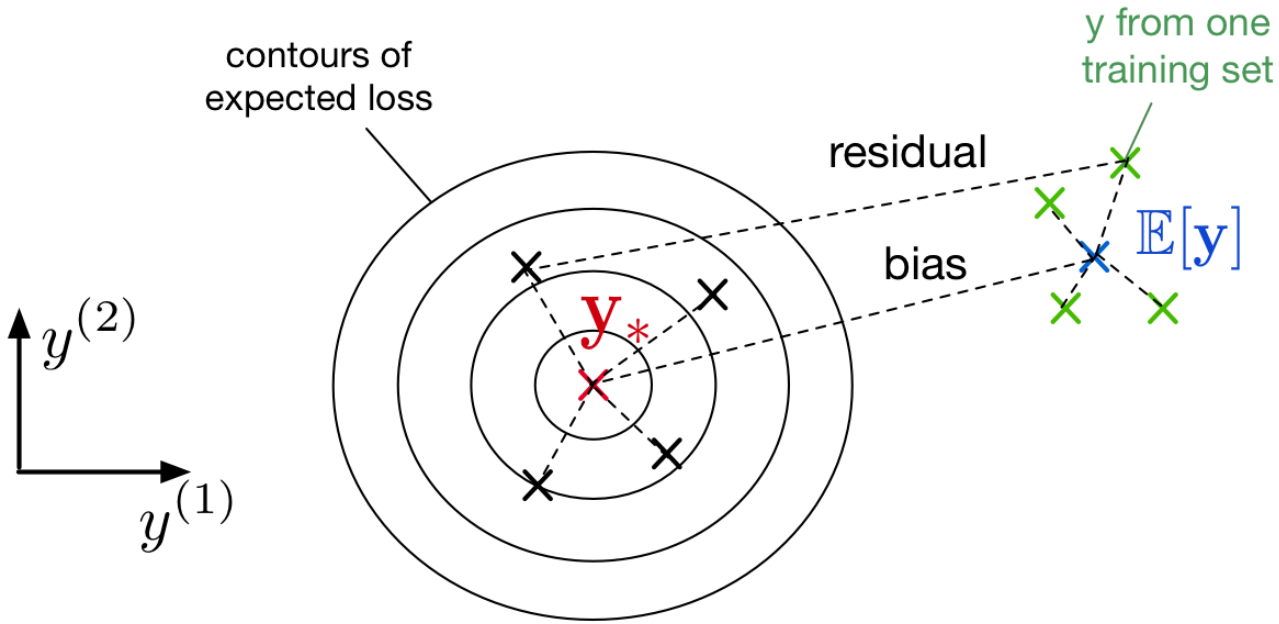
- Contrast if t is not random conditioned on $\mathbf{x}$:

$$\underbrace{\mathbb{E}_{\mathbf{x}}[(\mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})] - t)^2]}_{\text{bias}} + \underbrace{\mathbb{E}_{\mathbf{x},\mathcal{D}}[(h_{\mathcal{D}}(\mathbf{x}) - \mathbb{E}_{\mathcal{D}}[h_{\mathcal{D}}(\mathbf{x})])^2]}_{\text{variance}}$$

- We have no control over the Bayes error! In particular, bagging/boosting do not help.

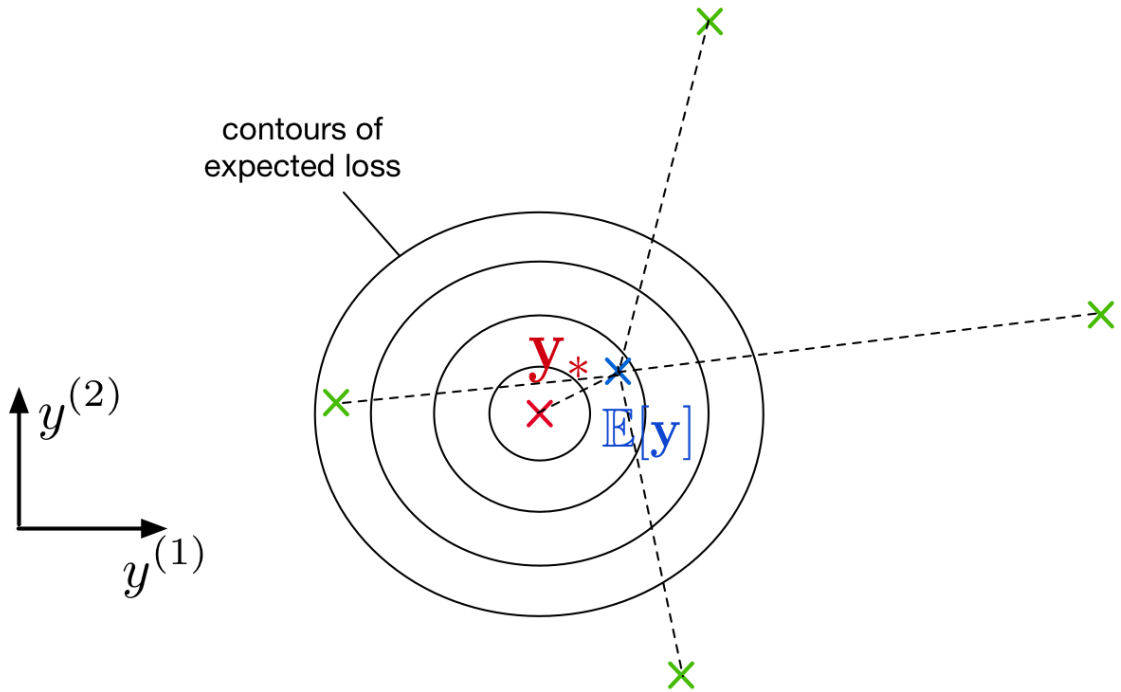
Bias/Variance Decomposition: Another Visualization

- We can visualize this decomposition in **output space**, where the axes correspond to predictions on the test examples.
- If we have an overly simple model (e.g. k-NN with large k), it might have
  - high bias (because it's too simplistic to capture the structure in the data)
  - low variance (because there's enough data to get a stable estimate of the decision boundary)



Bias/Variance Decomposition: Another Visualization

- If you have an overly complex model (e.g. k-NN with $k = 1$), it might have
  - low bias (since it learns all the relevant structure)
  - high variance (it fits the quirks of the data you happened to sample)



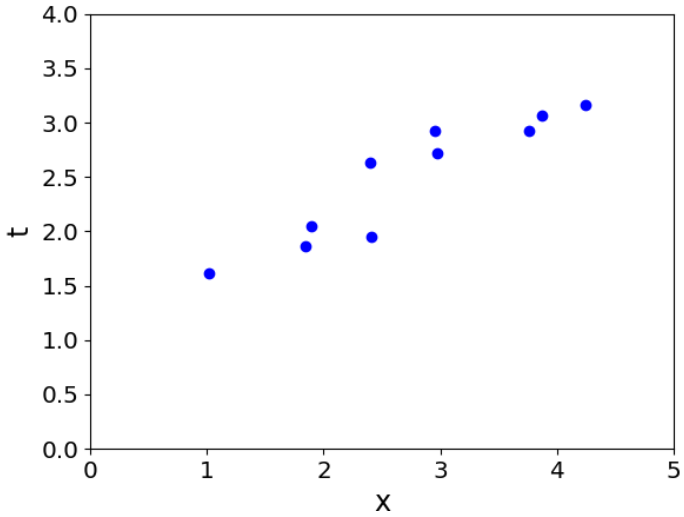
Summary

- Bagging reduces overfitting by averaging predictions.
- Used in most competition winners
  - Even if a single model is great, a small ensemble usually helps.
- Limitations:
  - Does not reduce bias.
  - There is still correlation between classifiers.
- Random forest solution: Add more randomness.

Summary so far

- So far, we've talked about *procedures* for learning.
  - KNN, decision trees, bagging, random forests
- For the remainder of this course, we'll take a more modular approach:
  - choose a model describing the relationships between variables of interest
  - define a loss function quantifying how bad is the fit to the data
  - choose a regularizer saying how much we prefer different candidate explanations
  - fit the model, e.g. using an optimization algorithm
- By mixing and matching these modular components, your ML skills become more powerful!

Recall the supervised learning setup



Recall that in supervised learning:

- There is target $t \in \mathcal{T}$ (also called response, outcome, output, class)
- There are features $x \in \mathcal{X}$ (also called inputs, covariates, design)
- Objective is to learn a function $f : \mathcal{X} \to \mathcal{T}$ such that

$$t \approx y = f(x)$$

based on some data $\mathcal{D} = \{(t^{(i)}, x^{(i)}) \text{ for } i = 1, 2, ..., N\}$.

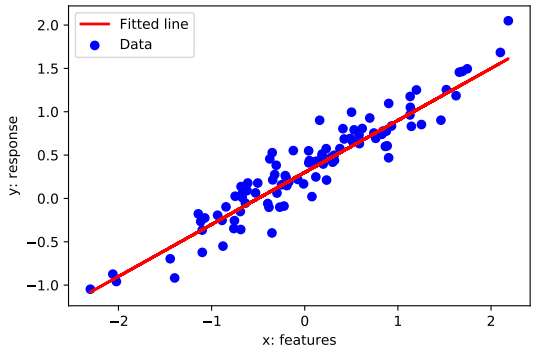
Problem Setup: linear regression

- Model: In linear regression, we use linear functions of the inputs $\mathbf{x} = (x_1, \ldots, x_D)$ to make predictions y of the target value t:

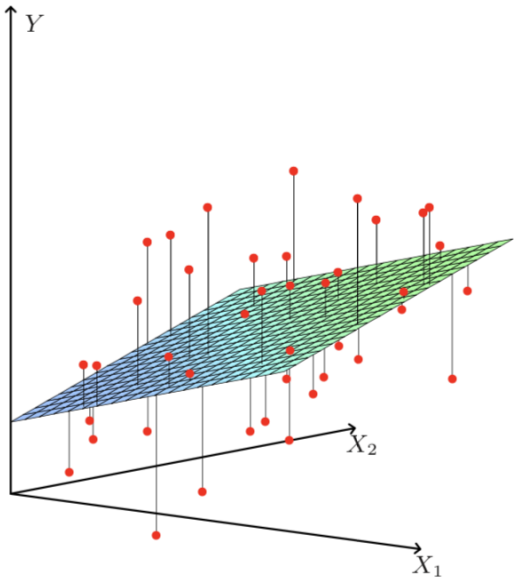
$$y = f(\mathbf{x}) = \sum_j w_j x_j + b$$

  - y is the prediction
  - $\mathbf{w}$ is the weights
  - b is the bias
- $\mathbf{w}$ and b together are the parameters
- We hope that our prediction is close to the target: $y \approx t$.

What is linear? 1 feature vs D features



- If we have only 1 feature:
  $y = wx + b$ where $w, x, b \in \mathbb{R}$.
- y is linear in x.



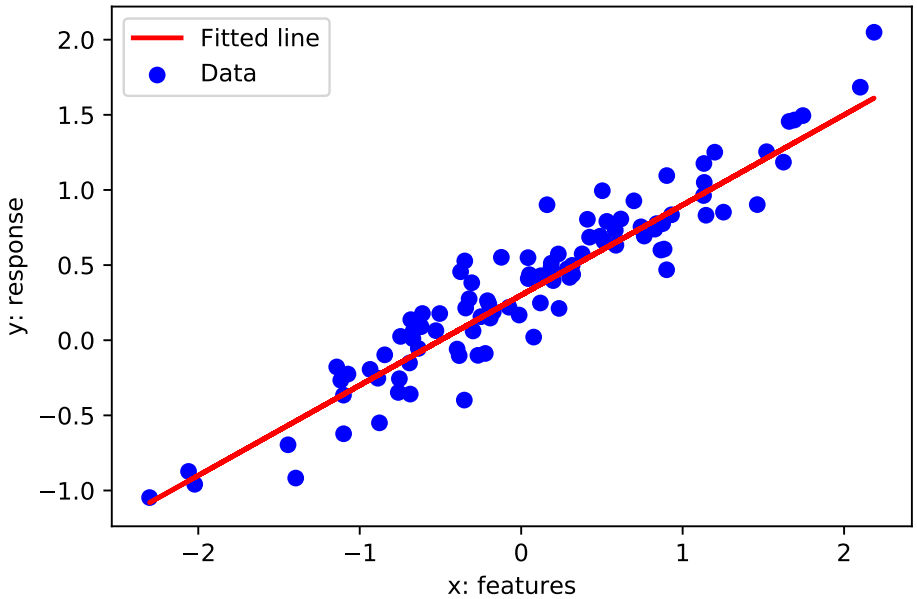
- If we have D features:
  $y = \mathbf{w}^\top \mathbf{x} + b$ where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^D$,
  $b \in \mathbb{R}$
- y is linear in $\mathbf{x}$.

Relation between the prediction y and inputs $\mathbf{x}$ is linear in both cases.

Linear Regression

We have a dataset $\mathcal{D} = \{(t^{(i)}, x^{(i)}) \text{ for } i = 1, 2, ..., N\}$ where,

- $t^{(i)} \in \mathbb{R}$ is the target or response (e.g. income),
- $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_D^{(i)})^\top \in \mathbb{R}^D$ are the inputs (e.g. age, height)
- predict $t^{(i)}$ with a linear function of $\mathbf{x}^{(i)}$:



- $t^{(i)} \approx y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$
- Find the "best" line $(\mathbf{w}, b)$.
- minimize$_{(\mathbf{w},b)} \sum_{i=1}^{N} \mathcal{L}(y^{(i)}, t^{(i)})$

Problem Setup

- Loss function: squared error (says how bad the fit is)

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- Cost function: loss function averaged over all training examples

$$\begin{aligned}\mathcal{J}(\mathbf{w}, b) &= \frac{1}{2}\sum_{i=1}^{N}\left(y^{(i)} - t^{(i)}\right)^2 \\ &= \frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)}\right)^2\end{aligned}$$

Vector notation

- We can organize all the training examples into a design matrix $\mathbf{X}$ with one row per training example, and all the targets into the target vector $\mathbf{t}$.

one feature across all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training example (vector)

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^T\mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^T\mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

Vectorization

- Computing the squared error cost across the whole dataset:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$
$$\mathcal{J} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$$

- We can also add a column of 1's to design matrix, combine the bias and the weights, and conveniently write

$$\mathbf{X} = \begin{bmatrix} 1 & [\mathbf{x}^{(1)}]^\top \\ 1 & [\mathbf{x}^{(2)}]^\top \\ 1 & \vdots \end{bmatrix} \in \mathbb{R}^{N \times D+1} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \end{bmatrix} \in \mathbb{R}^{D+1}$$

Then, our predictions reduce to $\mathbf{y} = \mathbf{X}\mathbf{w}$.

Solving the minimization problem

- We defined a cost function. This is what we'd like to minimize.
- Recall from calculus class: minimum of a smooth function (if it exists) occurs at a critical point, i.e. point where the derivative is zero.
- Multivariate generalization: set the partial derivatives to zero (or equivalently the gradient). We call this direct solution.

Direct solution

- Partial derivatives: derivatives of a multivariate function with respect to one of its arguments.

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \to 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- To compute, take the single variable derivatives, pretending the other arguments are constant.
- Example: partial derivatives of the prediction y

$$\begin{aligned}
\frac{\partial y}{\partial w_j} &= \frac{\partial}{\partial w_j} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\
&= x_j \\
\frac{\partial y}{\partial b} &= \frac{\partial}{\partial b} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\
&= 1
\end{aligned}$$

Direct solution

- Chain rule for derivatives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}y}\frac{\partial y}{\partial w_j} \\ &= \frac{\mathrm{d}}{\mathrm{d}y}\left[\frac{1}{2}(y-t)^2\right]\cdot x_j \\ &= (y-t)x_j \\ \frac{\partial \mathcal{L}}{\partial b} &= y-t\end{aligned}$$

- Cost derivatives (average over data points):

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)}-t^{(i)})\,x_j^{(i)}$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N}y^{(i)}-t^{(i)}$$

Direct solution

- The minimum must occur at a point where the partial derivatives are zero.

$$\frac{\partial \mathcal{J}}{\partial w_j} = 0 \qquad \frac{\partial \mathcal{J}}{\partial b} = 0.$$

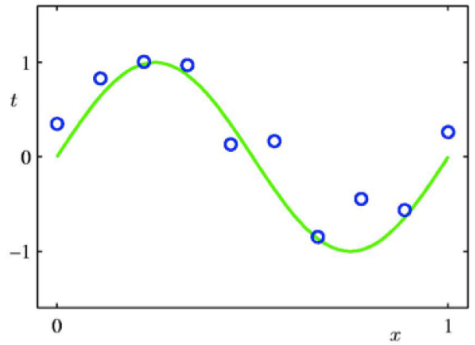
- If $\partial \mathcal{J} / \partial w_j \neq 0$, you could reduce the cost by changing w_j.
- This turns out to give a system of linear equations, which we can solve efficiently. **Full derivation in the preliminaries.pdf.**
- Optimal weights:

$$\mathbf{w}^{\text{LS}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$$

- Linear regression is one of only a handful of models in this course that permit direct solution.

What if it isn't linear: Polynomial curve fitting

If the relationship doesn't look linear, we can fit a polynomial.



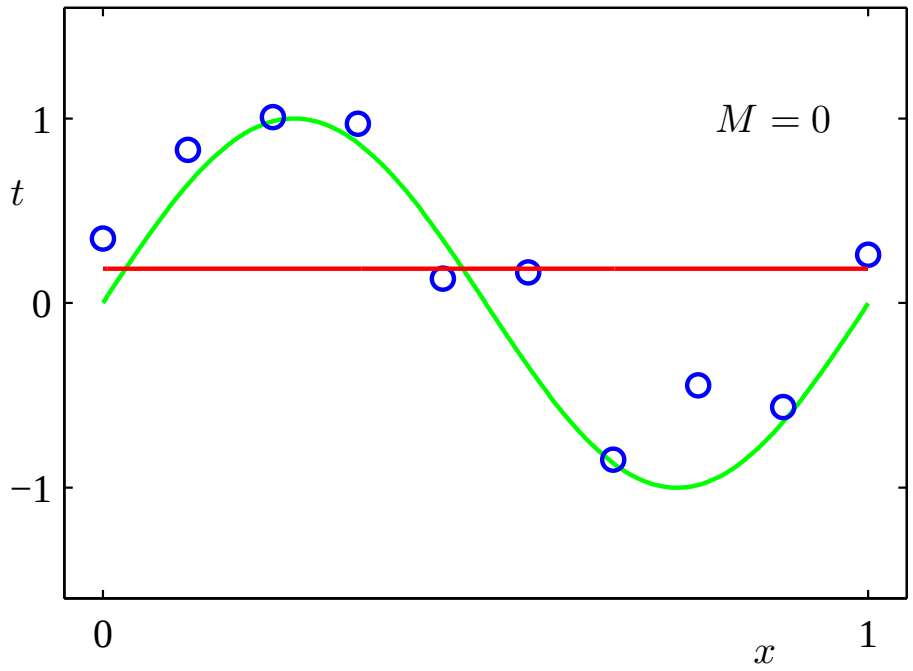
Fit the data using a degree-M polynomial function of the form:

$$y = w_0 + w_1 x + w_2 x^2 + ... + w_M x^M = \sum_{i=0}^{M} w_i x^i$$

- This is called **feature mapping:** $y = \mathbf{w}^\top \boldsymbol{\psi}(x)$ where $\boldsymbol{\psi}(x) = [1, x, x^2, ...]^\top$. In general, $\boldsymbol{\psi}$ can be any function.
- We can still use least squares since t is linear in $w_0, w_1, ...$
- Form a feature vector $\mathbf{x}' = (1, x, x^2, ..., x^M)$ and solve the least squares problem.

Fitting polynomials: $M = 0$

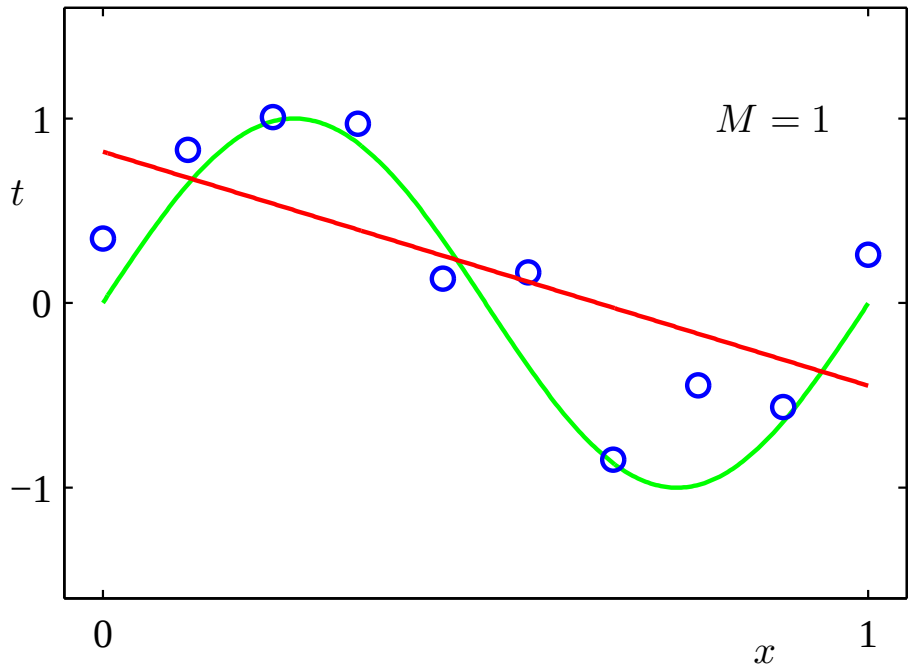
$$y = w_0$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials: $M = 1$

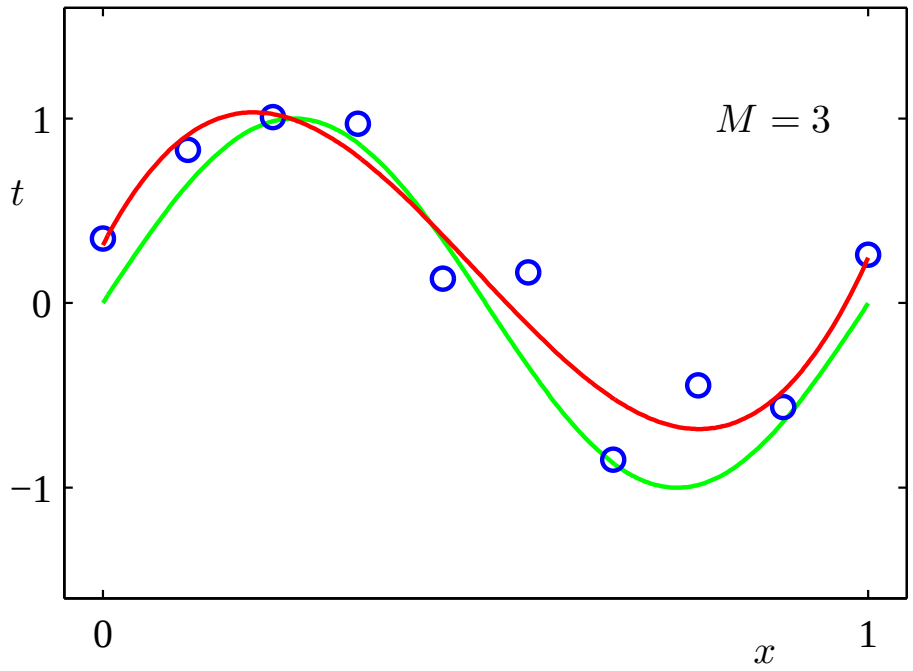
$$y = w_0 + w_1 x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials: $M = 3$

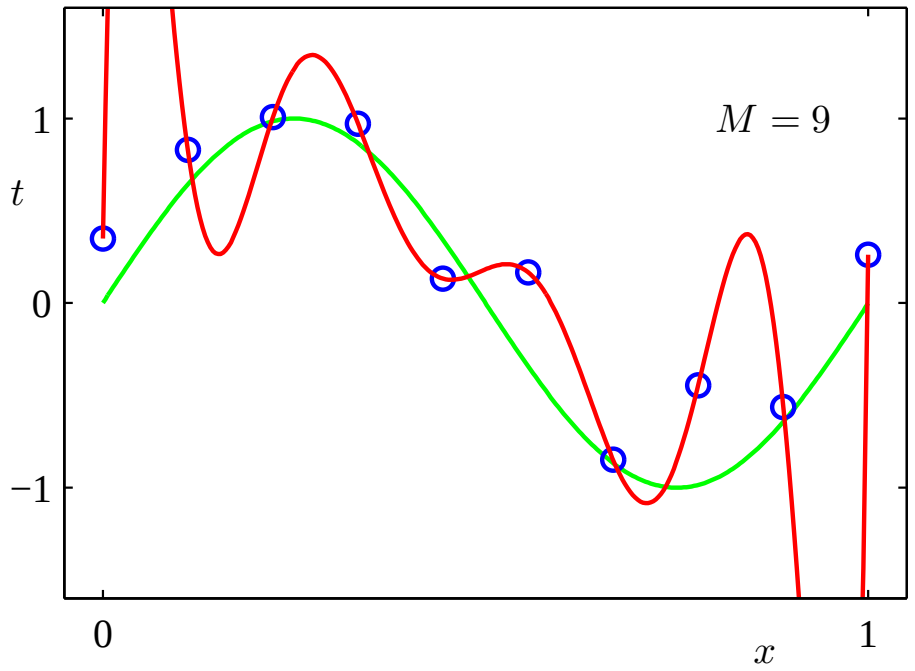
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials: $M = 9$

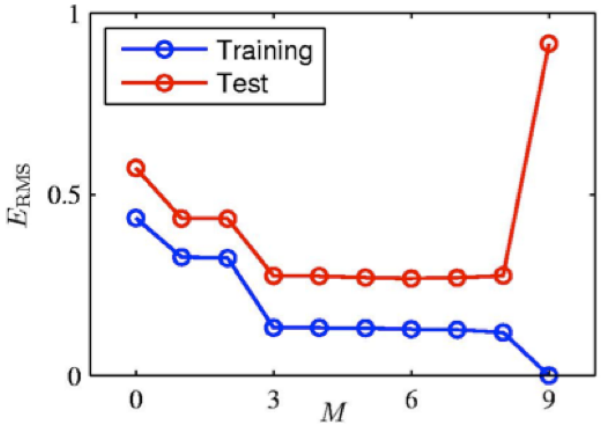
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$

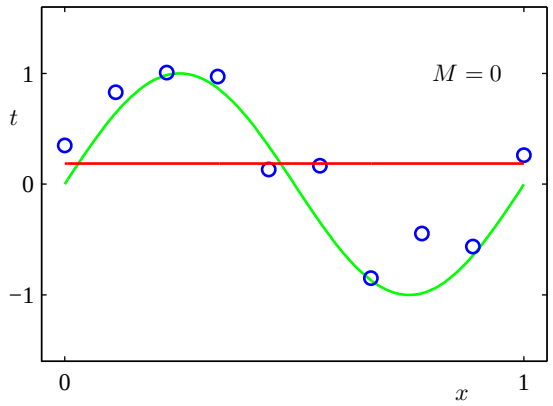


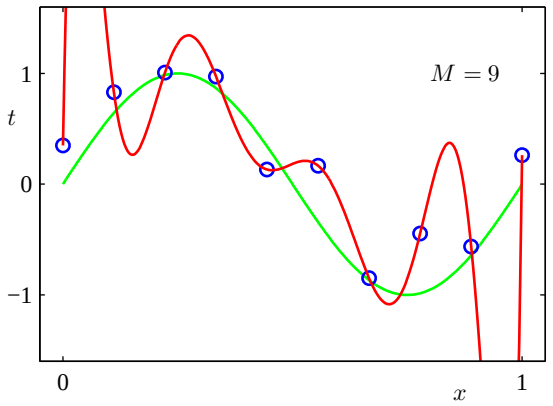
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

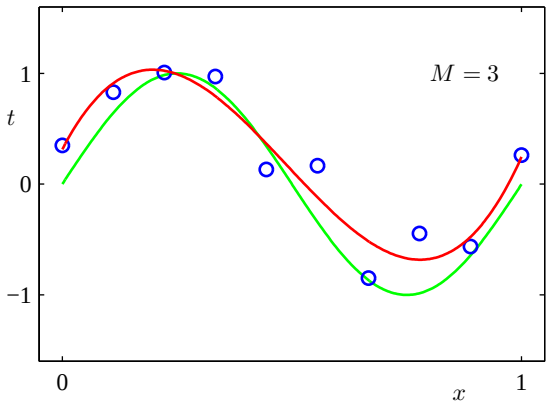
Underfitting (M=0): model is too simple — does not fit the data.
Overfitting (M=9): model is too complex — fits perfectly.





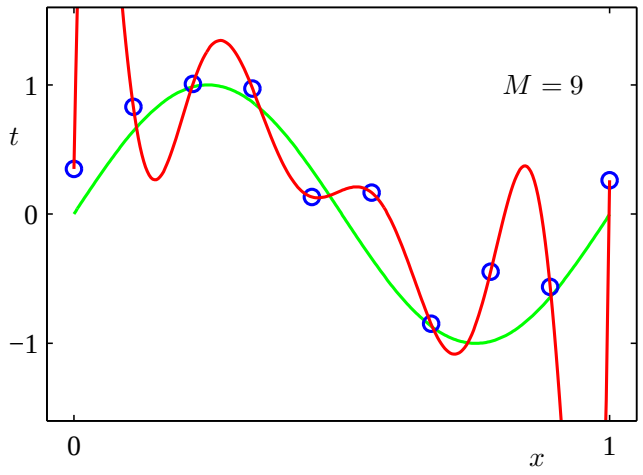


Good model (M=3): Achieves small test error (generalizes well).



Generalization

	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43



- As M increases, the magnitude of coefficients gets larger.
- For $M = 9$, the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

Regularization

- The degree of the polynomial M is a hyperparameter, just like k in KNN. We can tune it using a validation set.
- But restricting the size of the model is a crude solution, since you'll never be able to learn a more complex model, even if the data support it.
- Another approach: keep the model large, but regularize it
  - Regularizer: a function that quantifies how much we prefer one hypothesis vs. another

L^2 (or ℓ_2) Regularization

- We can encourage the weights to be small by choosing as our regularizer the L^2 penalty.

$$\mathcal{R}(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2 = \frac{1}{2}\sum_j w_j^2.$$

  - Note: to be pedantic, the L^2 norm is Euclidean distance, so we're really regularizing the *squared* L^2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

$$\mathcal{J}_{\text{reg}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \lambda\mathcal{R}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \frac{\lambda}{2}\sum_j w_j^2$$

- If you fit training data poorly, $\mathcal{J}$ is large. If your optimal weights have high values, $\mathcal{R}$ is large.
- Here, λ is a hyperparameter that we can tune with a validation set.
- Large λ penalizes weight values more.

L^2 Regularized least squares: Ridge regression

For the least squares problem, we have $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$.

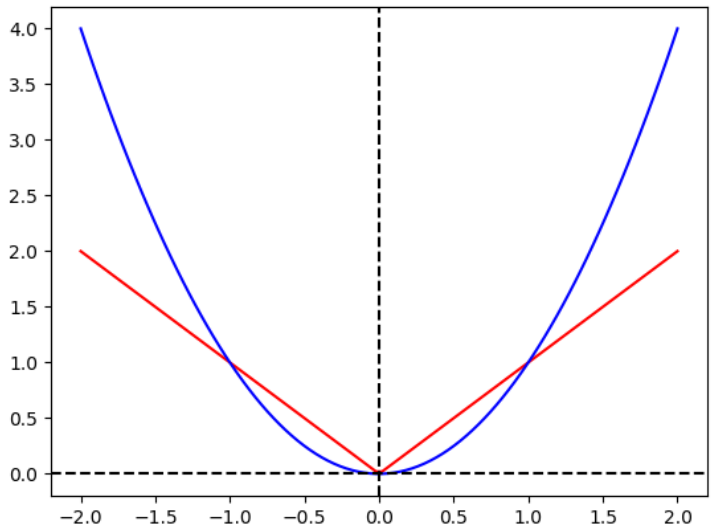
- When $\lambda > 0$ (with regularization), regularized cost gives

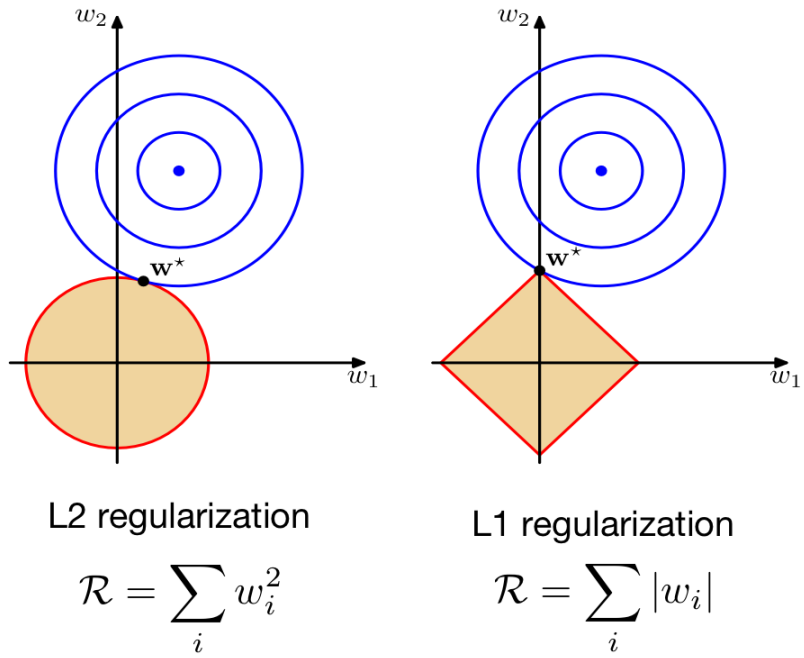
$$\begin{aligned}\mathbf{w}_\lambda^{Ridge} = \operatorname*{argmin}_{\mathbf{w}} \mathcal{J}_{\text{reg}}(\mathbf{w}) &= \operatorname*{argmin}_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \\ &= (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{t}\end{aligned}$$

- The case $\lambda = 0$ (no regularization) reduces to least squares solution!

L^1 vs. L^2 Regularization

- The L^1 norm, or sum of absolute values, is another regularizer that encourages weights to be exactly zero. (How can you tell?)
- We can design regularizers based on whatever property we'd like to encourage.





— Bishop, *Pattern Recognition and Machine Learning*

Conclusion

Linear regression exemplifies recurring themes of this course:

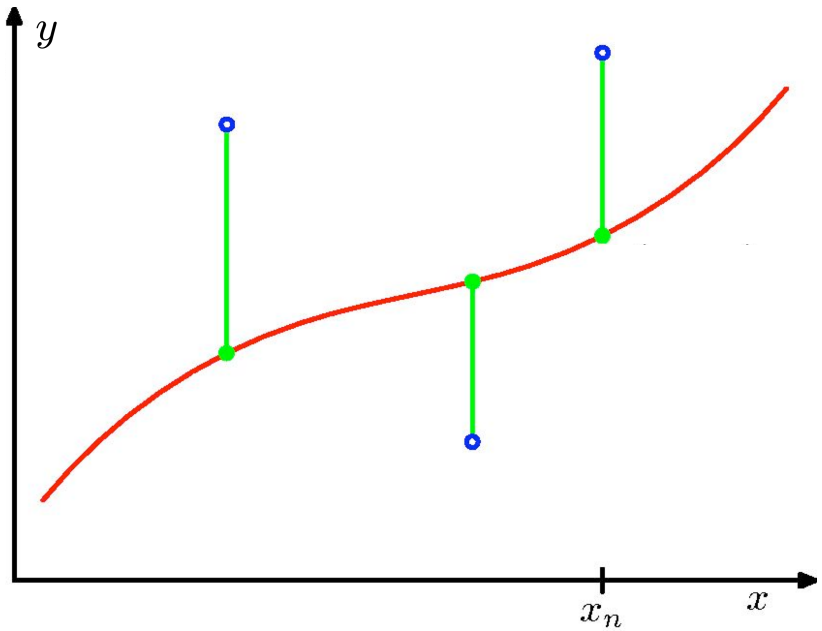
- choose a model and a loss function
- formulate an optimization problem
- solve the minimization problem using one of two strategies
  - direct solution (set derivatives to zero)
  - gradient descent (see appendix)
- vectorize the algorithm, i.e. represent in terms of linear algebra
- make a linear model more powerful using features
- improve the generalization by adding a regularizer

Appendix

Probabilistic Interpretation

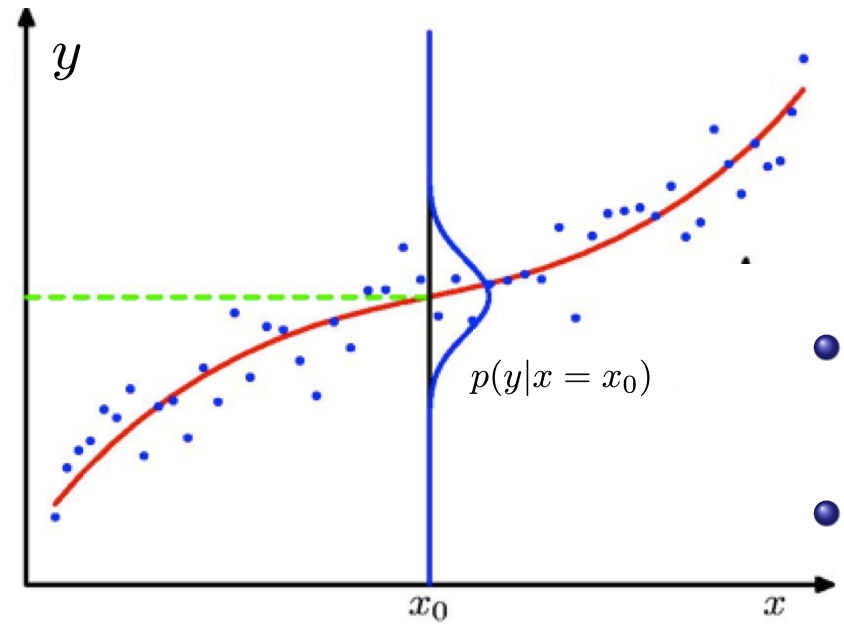
For the least squares: we minimize the sum of the squares of the errors between the predictions for each data point $x^{(i)}$ and the corresponding target values $t^{(i)}$, i.e.,

$$\underset{(\mathbf{w},\mathbf{w}_0)}{\text{minimize}} \sum_{i=1}^{n} (t^{(i)} - \mathbf{w}^\top x^{(i)} + b)^2$$



- $t \approx x^\top \mathbf{w} + b$, $(\mathbf{w}, b) \in \mathbb{R}^D \times \mathbb{R}$
- So far we saw that polynomial curve fitting can be expressed in terms of error minimization.
- We now view it from probabilistic perspective.

Probabilistic interpretation



- Suppose that our model arose from a statistical model ($b=0$ for simplicity):

$$y^{(i)} = \mathbf{w}^\top x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ is independent of anything else.

- Thus, $y^{(i)}|x^{(i)} \sim p(y|x^{(i)}, \mathbf{w}) = \mathcal{N}(\mathbf{w}^\top x^{(i)}, \sigma^2)$.
- So far we saw that polynomial curve fitting can be expressed in terms of error minimization.
- We now view it from probabilistic perspective.

Maximum Likelihood Estimation

- If the samples $z^{(i)} = (y^{(i)}|x^{(i)}, \mathbf{w})$ are assumed to be independently distributed (**not i.i.d assumption**),
- and drawn from a distribution

$$y^{(i)} \sim p(y|x^{(i)}, \mathbf{w})$$

where $\mathbf{w}$ is a parameter to be estimated,

- then joint density takes the form

$$p(y^{(1)}, y^{(2)}, ..., y^{(n)}|x^{(1)}, x^{(2)}, ..., x^{(n)}, \mathbf{w}) = \prod_{i=1}^{n} p(y^{(i)}|x^{(i)}, \mathbf{w}) = L(\mathbf{w})$$

which is called the likelihood (which doesn't refer to joint density!).

Maximum likelihood estimation: after observing the data samples $z^{(i)}$ for $i = 1, 2, ..., n$ we should choose $\mathbf{w}$ that maximizes the likelihood.

Probabilistic Interpretation

Product of n terms is not easy to minimize. Taking log reduces it to a sum! Two objectives are equivalent since log is strictly increasing.

Maximizing the likelihood is equivalent to minimizing the negative log-likelihood:

$$\ell(\mathbf{w}) = -\log L(\mathbf{w}) = -\log \prod_{i=1}^{n} p(z^{(i)}|\mathbf{w}) = -\sum_{i=1}^{n} \log p(z^{(i)}|\mathbf{w})$$

Maximum Likelihood Estimator (MLE)

After observing $z^{(i)}$ for $i = 1, ..., n$ i.i.d. samples from $p(z|\mathbf{w})$, MLE is

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\text{argmin}} \quad l(\mathbf{w}) = -\sum_{i=1}^{n} \log p(z^{(i)}|\mathbf{w})$$

Back to Linear Regression

- Suppose that our model arose from a statistical model:

$$y^{(i)} = \mathbf{w}^\top x^{(i)} + \epsilon^{(i)}$$

  where $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ is independent of anything else.
- $p(y^{(i)}|x^{(i)}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top x^{(i)})^2\right\}$
- $\log p(y^{(i)}|x^{(i)}, \mathbf{w}) = -\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top x^{(i)})^2 - \log(\sqrt{2\pi\sigma^2})$
- $\mathbf{w}^{\text{MLE}} = \text{argmin}_{\mathbf{w}} \ \mathcal{L}(\mathbf{w}) = \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y^{(i)} - \mathbf{w}^\top x^{(i)})^2 + C$ where C and σ doesn't depend on $\mathbf{w}$, so don't contribute to the minimization.

$\mathbf{w}^{\text{MLE}} = \mathbf{w}^{\text{LS}}$ when we work with Gaussian densities!

Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: gradient descent.
- Gradient descent is an iterative algorithm, which means we apply an update repeatedly until some criterion is met.
- We initialize the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the direction of steepest descent.

Gradient descent

- Observe:
  - if $\partial \mathcal{J} / \partial w_j > 0$, then increasing w_j increases $\mathcal{J}$.
  - if $\partial \mathcal{J} / \partial w_j < 0$, then increasing w_j decreases $\mathcal{J}$.
- The following update decreases the cost function:

$$\begin{aligned} w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\ &= w_j - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, x_j^{(i)} \end{aligned}$$

- α is a learning rate. The larger it is, the faster $\mathbf{w}$ changes.
  - We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001

Gradient descent

- This gets its name from the gradient:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

  - This is the direction of fastest increase in $\mathcal{J}$.
- Update rule in vector form:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}} \\ &= \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \, \mathbf{x}^{(i)} \end{aligned}$$

- Hence, gradient descent updates the weights in the direction of fastest *decrease*.

Gradient descent

- Why gradient descent, if we can find the optimum directly?
  - GD can be applied to a much broader set of models
  - GD can be easier to implement than direct solutions, especially with automatic differentiation software
  - For regression in high-dimensional spaces, GD is more efficient than direct solution (matrix inversion is an $\mathcal{O}(D^3)$ algorithm).

Gradient descent under L^2 Regularization

- Recall the gradient descent update:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

- The gradient descent update of the regularized cost has an interesting interpretation as weight decay:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\
&= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}
\end{aligned}$$

Brief Matrix/vector calculus

- For a function $f : \mathbb{R}^p \to \mathbb{R}$, $\nabla f(z)$ denotes the gradient at z which points in the direction of the greatest rate of increase.
- $\nabla f(x) \in \mathbb{R}^p$ is a vector with $[\nabla f(x)]_i = \frac{\partial}{\partial x_i} f(x)$.
- $\nabla^2 f(x) \in \mathbb{R}^{p \times p}$ is a matrix with $[\nabla^2 f(x)]_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$
- At any minimum of a function f, we have $\nabla f(\mathbf{w}) = 0$, $\nabla^2 f(\mathbf{w}) \succeq 0$.
- Consider the problem $\underset{\mathbf{w}}{\text{minimize}}\ \ell(\mathbf{w}) = \frac{1}{2}\|y - X\mathbf{w}\|_2^2$,
- $\nabla \ell(\mathbf{w}) = X^\top (X\mathbf{w} - y) = 0 \implies \hat{\mathbf{w}} = (X^\top X)^{-1} X^\top y$ (assuming $X^\top X$ is invertible)

At an arbitrary point x (old/new observation), our prediction is $y = \hat{\mathbf{w}}^\top x$.

Vectorization

- Computing the prediction using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- For-loops in Python are slow, so we vectorize algorithms by expressing them in terms of vectors and matrices.

$$\mathbf{w} = (w_1, \ldots, w_D)^T \qquad \mathbf{x} = (x_1, \ldots, x_D)$$

$$y = \mathbf{w}^T \mathbf{x} + b$$

- This is simpler and much faster:

```
y = np.dot(w, x) + b
```

Vectorization

Why vectorize?

- The equations, and the code, will be simpler and more readable. Gets rid of dummy variables/indices!
- Vectorized code is much faster
  - Cut down on Python interpreter overhead
  - Use highly optimized linear algebra libraries
  - Matrix multiplication is very fast on a Graphics Processing Unit (GPU)