CSC 311: Introduction to Machine Learning

Lecture 3 - Bagging, Linear Models I

Roger Grosse Rahul G. Krishnan Guodong Zhang

University of Toronto, Fall 2021

Today

- Today we will introduce ensembling methods that combine multiple models and can perform better than the individual members.
  - We've seen many individual models (KNN, decision trees)
- We will see bagging:
  - Train models independently on random "resamples" of the training data.
- We will introduce linear regression, our first parametric learning algorithm.
  - This will exemplify how we'll think about learning algorithms for the rest of the course.

Bias/Variance Decomposition

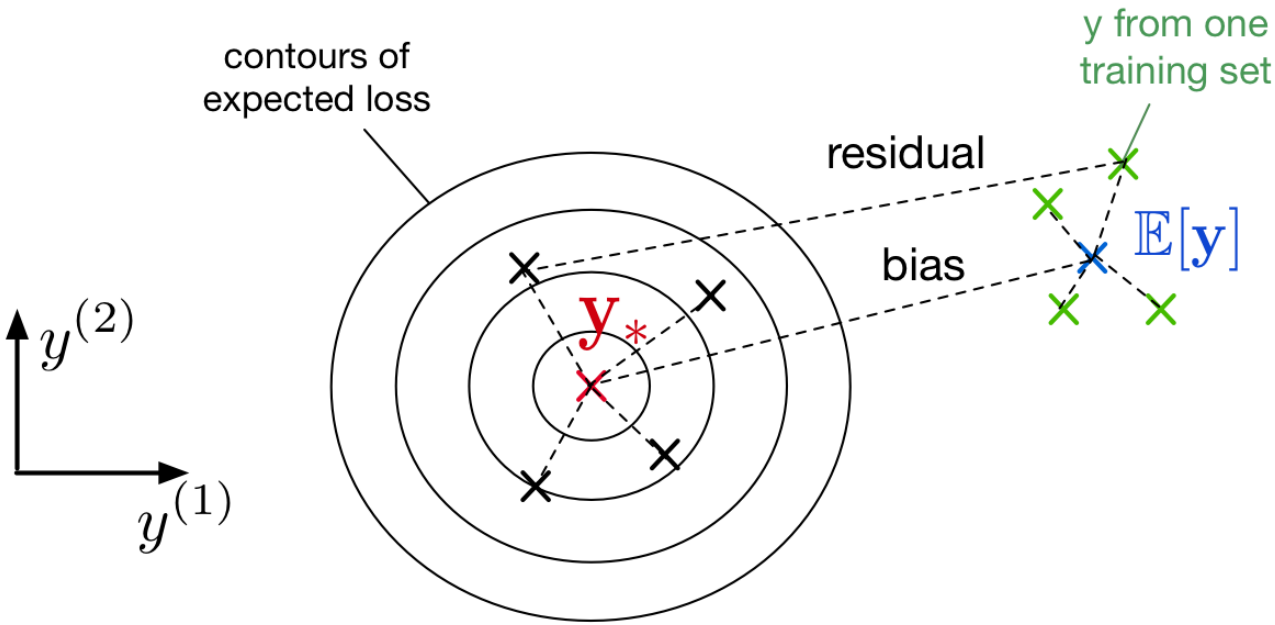
- Recall, we treat predictions y at a query $\mathbf{x}$ as a random variable (where the randomness comes from the choice of dataset), $y_\star$ is the optimal deterministic prediction, t is a random target sampled from the true conditional $p(t|\mathbf{x})$.

$$\mathbb{E}[(y-t)^2] = \underbrace{(y_\star - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\text{Var}(y)}_{\text{variance}} + \underbrace{\text{Var}(t)}_{\text{Bayes error}}$$

- Bias/variance decomposes the expected loss into three terms:
  - bias: how wrong the expected prediction is (corresponds to underfitting)
  - variance: the amount of variability in the predictions (corresponds to overfitting)
  - Bayes error: the inherent unpredictability of the targets
- Even though this analysis only applies to squared error, we often loosely use "bias" and "variance" as synonyms for "underfitting" and "overfitting".

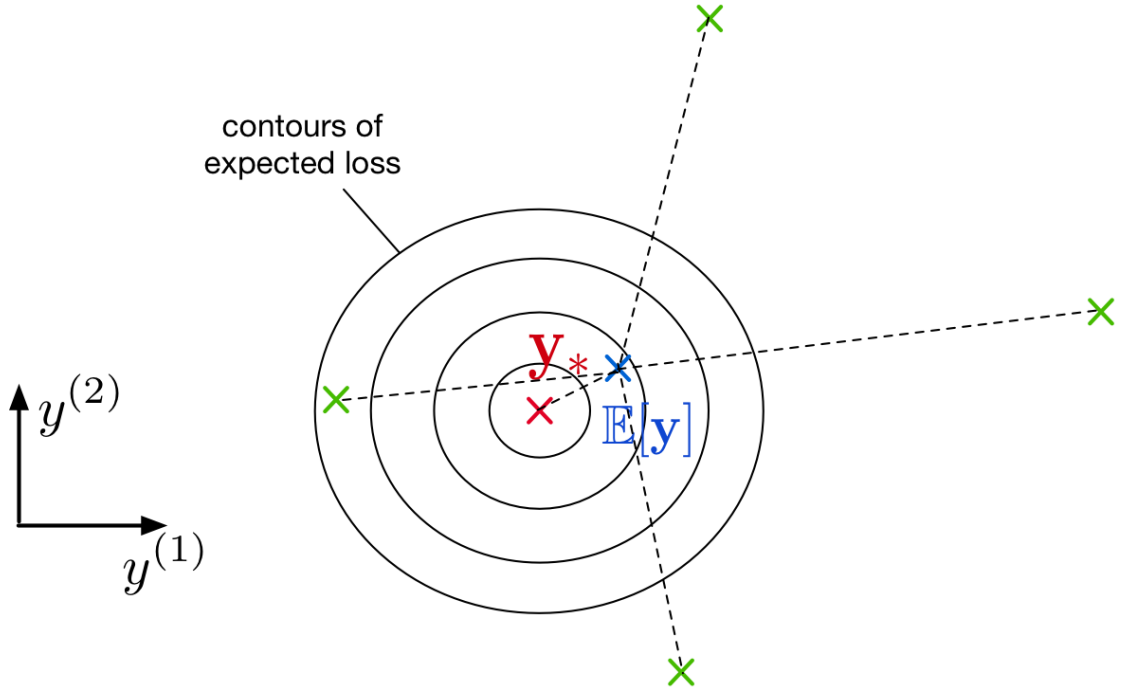
Bias/Variance Decomposition: Another Visualization

- 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. KNN with large k), it might have
  - high bias (because it cannot capture the structure in the data)
  - low variance (because there's enough data to get stable estimates)



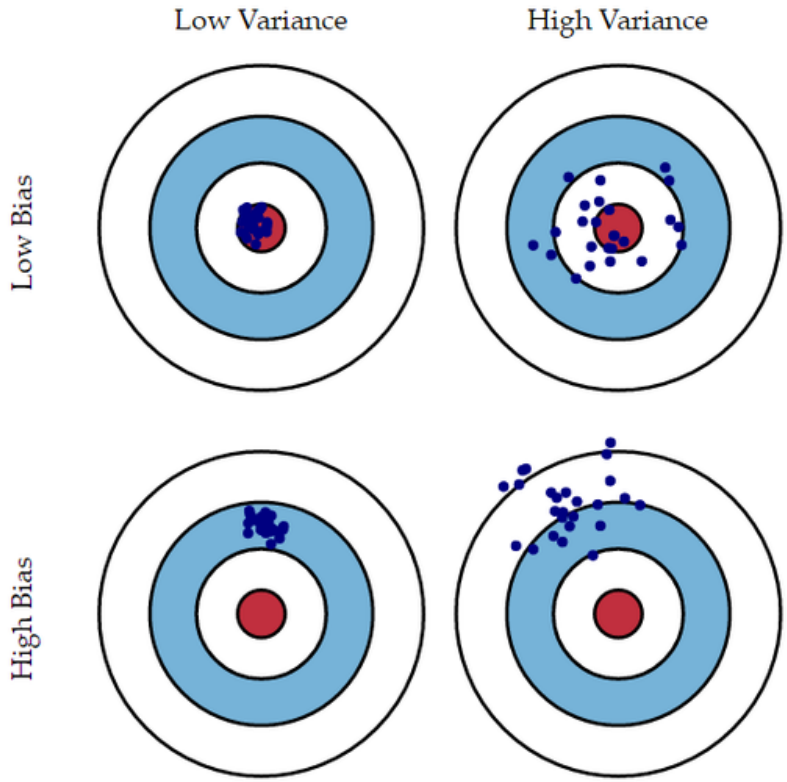
Bias/Variance Decomposition: Another Visualization

- If you have an overly complex model (e.g. KNN 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)



Bias/Variance Decomposition: Another Visualization

- The following graphic summarizes the previous two slides:



- What doesn't this capture?

A: Bayes error

Bagging: Motivation

- Suppose we could somehow sample m independent training sets from p_{sample}.
- We could then compute the prediction y_i based on each one, and take the average $y = \frac{1}{m}\sum_{i=1}^{m} y_i$.
- How does this affect the three terms of the expected loss?
  - **Bayes error: unchanged**, since we have no control over it
  - **Bias: unchanged**, since the averaged prediction has the same expectation

$$\mathbb{E}[y] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} y_i\right] = \mathbb{E}[y_i]$$

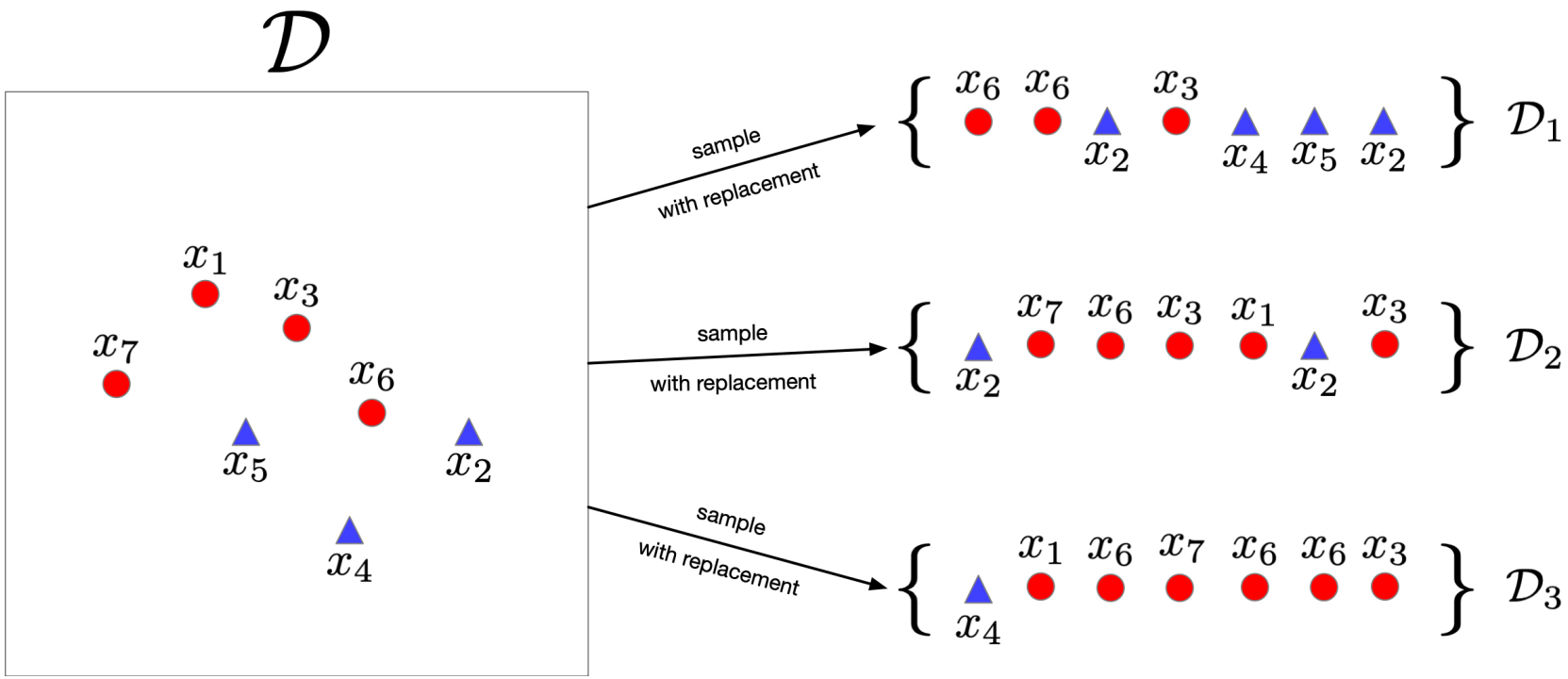
  - **Variance: reduced**, since we're averaging over independent samples

$$\text{Var}[y] = \text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} y_i\right] = \frac{1}{m^2}\sum_{i=1}^{m}\text{Var}[y_i] = \frac{1}{m}\text{Var}[y_i].$$

Bagging: The Idea

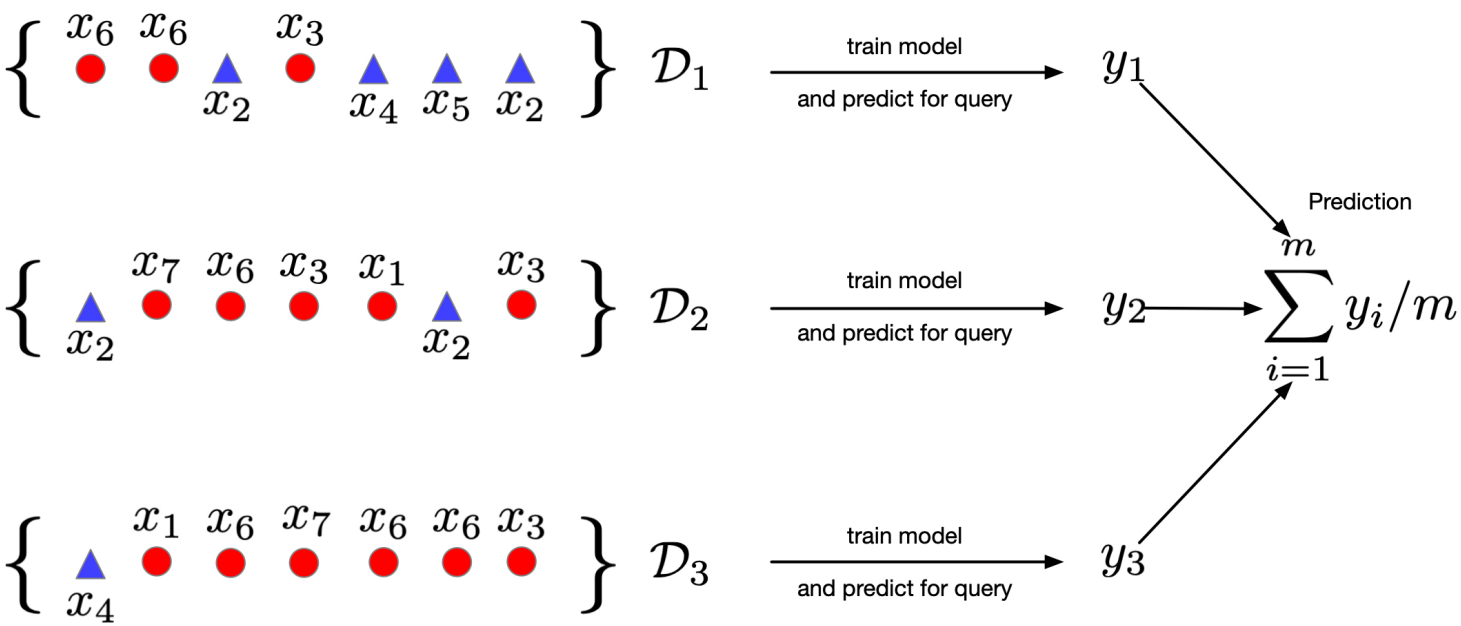
- In practice, the sampling distribution p_{sample} is often finite or expensive to sample from.
- So training separate models on independently sampled datasets is very wasteful of data!
  - Why not train a single model on the union of all sampled datasets?
- Solution: given training set $\mathcal{D}$, use the empirical distribution $p_{\mathcal{D}}$ as a proxy for p_{sample}. This is called bootstrap aggregation, or bagging .
  - Take a single dataset $\mathcal{D}$ with n examples.
  - Generate m new datasets ("resamples" or "bootstrap samples"), each by sampling n training examples from $\mathcal{D}$, with replacement.
  - Average the predictions of models trained on each of these datasets.
- The bootstrap is one of the most important ideas in all of statistics!
  - Intuition: As $|\mathcal{D}| \to \infty$, we have $p_{\mathcal{D}} \to p_{\text{sample}}$.

Bagging



in this example $n = 7$, $m = 3$

Bagging



predicting on a query point x

Bagging for Binary Classification

- If our classifiers output real-valued probabilities, $z_i \in [0, 1]$, then we can average the predictions before thresholding:

$$y_{\text{bagged}} = \mathbb{I}(z_{\text{bagged}} > 0.5) = \mathbb{I}\left(\sum_{i=1}^{m} \frac{z_i}{m} > 0.5\right)$$

- If our classifiers output binary decisions, $y_i \in \{0, 1\}$, we can still average the predictions before thresholding:

$$y_{\text{bagged}} = \mathbb{I}\left(\sum_{i=1}^{m} \frac{y_i}{m} > 0.5\right)$$

  This is the same as taking a majority vote.

- A bagged classifier can be stronger than the average underyling model.
  - E.g., individual accuracy on "Who Wants to be a Millionaire" is only so-so, but "Ask the Audience" is quite effective.

Bagging: Effect of Correlation

- 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

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

- 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
- 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

Bagging Summary

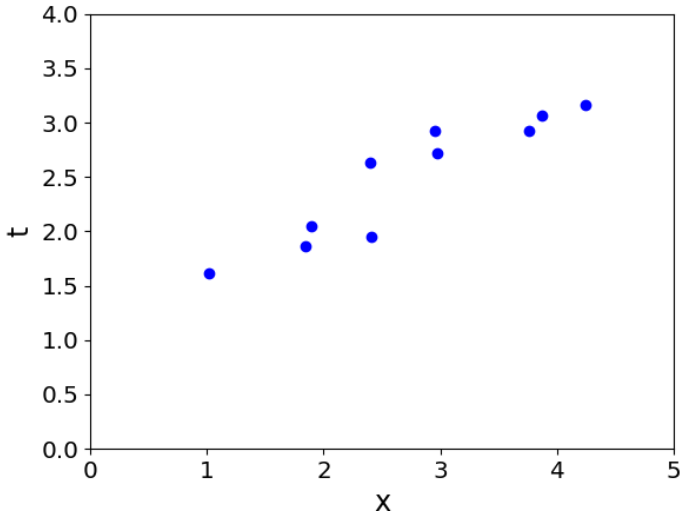
- Bagging reduces overfitting by averaging predictions.
- Used in most competition winners
  - Even if a single model is great, a small ensemble usually helps.
- Limitations:
  - Does not reduce bias in case of squared error.
  - There is still correlation between classifiers.
    - Random forest solution: Add more randomness.
  - Naive mixture (all members weighted equally).
    - If members are very different (e.g., different algorithms, different data sources, etc.), we can often obtain better results by using a principled approach to weighted ensembling.

Linear Regression

Overview

- Second learning algorithm of the course: linear regression.
  - Task: predict scalar-valued targets (e.g. stock prices)
  - Architecture: linear function of the inputs
- While KNN was a complete algorithm, linear regression exemplifies a modular approach that will be used throughout this course:
  - choose a model describing the relationships between variables of interest
  - define a loss function quantifying how bad the fit to the data is
  - choose a regularizer saying how much we prefer different candidate models (or explanations of data)
  - fit a model that minimizes the loss function and satisfies the constraint/penalty imposed by the regularizer, possibly using an optimization algorithm
- Mixing and matching these modular components give us a lot of new ML methods.

Supervised Learning Setup



In supervised learning:

- There is input $\mathbf{x} \in \mathcal{X}$, typically a vector of features (or covariates)
- There is target $t \in \mathcal{T}$ (also called response, outcome, output, class)
- Objective is to learn a function $f : \mathcal{X} \to \mathcal{T}$ such that $t \approx y = f(\mathbf{x})$ based on some data $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, ..., N\}$.

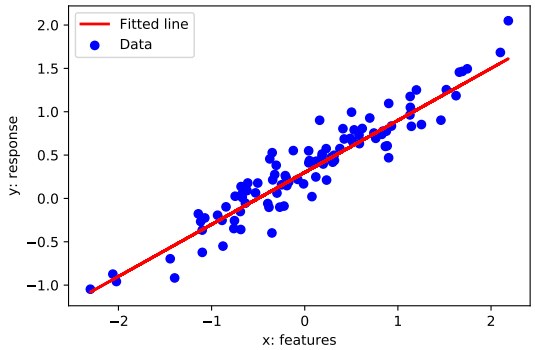
Linear Regression - Model

- Model: In linear regression, we use a *linear* function of the features $\mathbf{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$ to make predictions y of the target value $t \in \mathbb{R}$:

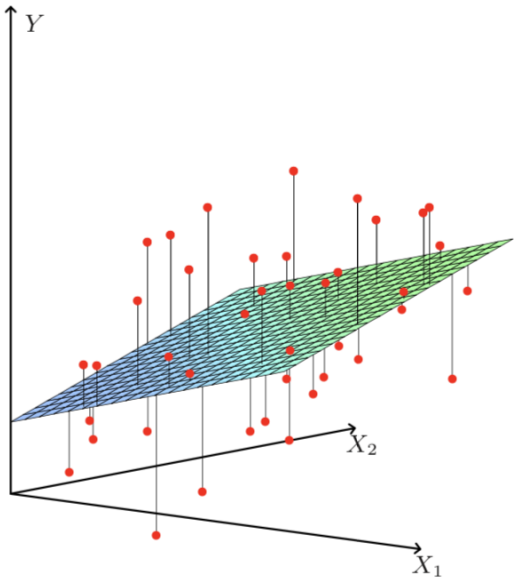
$$y = f(\mathbf{x}) = \sum_j w_j x_j + b$$

  - y is the prediction
  - $\mathbf{w}$ is the weights
  - b is the bias (or intercept)
- $\mathbf{w}$ 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 - Loss Function

- A loss function $\mathcal{L}(y, t)$ defines how bad it is if, for some example $\mathbf{x}$, the algorithm predicts y, but the target is actually t.
- Squared error loss function:

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

- $y - t$ 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

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

- Terminology varies. Some call "cost" *empirical* or *average loss*.

Vectorization

Vectorization

- The prediction for one data point can be computed using a for loop:

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

- Excessive super/sub scripts are hard to work with, and Python loops are slow, so we vectorize algorithms by expressing them in terms of vectors and matrices.

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

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

- This is simpler and executes 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 (hardware support)
  - Matrix multiplication very fast on GPU (Graphics Processing Unit)

Switching in and out of vectorized form is a skill you gain with practice

- Some derivations are easier to do element-wise
- Some algorithms are easier to write/understand using for-loops and vectorize later for performance

Vectorization

- 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:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2$$

- Sometimes we may use $\mathcal{J} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$, without a normalizer. This would correspond to the sum of losses, and not the averaged loss. The minimizer does not depend on N (but optimization might!).
- We can also add a column of 1's to design matrix, combine the bias and the weights, and conveniently write

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

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

Optimization

Solving the Minimization Problem

We defined a cost function $\mathcal{J}(\mathbf{w})$. This is what we'd like to minimize.

Recall from calculus: the 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 $\partial \mathcal{J} / \partial w_j$ to zero.
- Equivalently, we can set the gradient to zero. The gradient is the vector of partial derivatives:

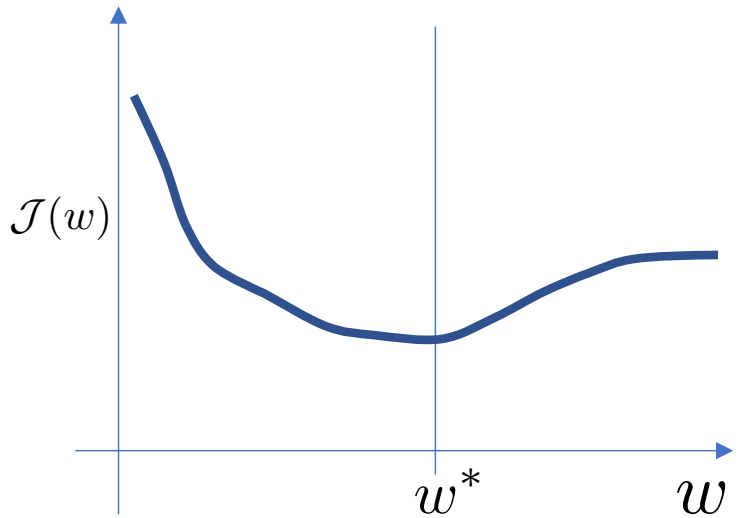
$$\nabla_{\mathbf{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

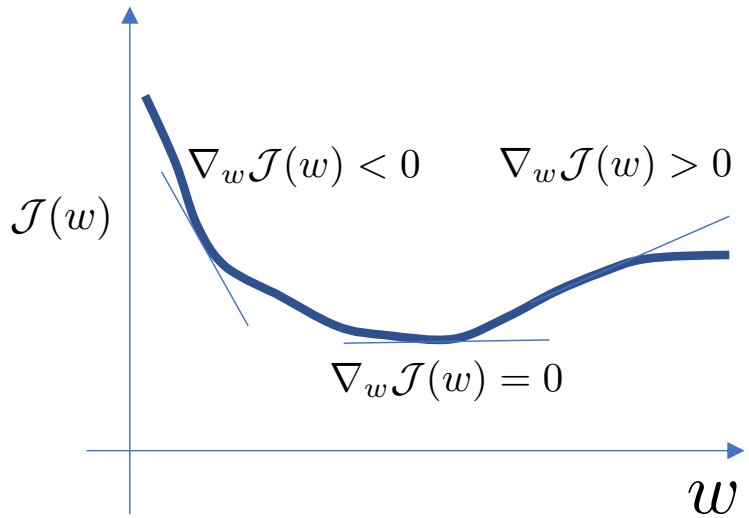
Solutions may be direct or iterative

- Sometimes we can directly find provably optimal parameters (e.g. set the gradient to zero and solve in closed form). We call this a direct solution.
- Iterative solution methods repeatedly apply an update rule that gradually takes us closer to the soltuion.

Direct Solution: Calculus

- Lets consider a cartoon visualization of $\mathcal{J}(w)$ where w is single dimensional
- **Left** We seek $w = w^*$ that minimizes $\mathcal{J}(w)$
- **Right** The gradients of a function can tell us where the maxima and minima of functions lie
- **Strategy:** Write down an algebraic expression for $\nabla_w \mathcal{J}(w)$. Set equation to 0. Solve for w





Direct Solution: Calculus

- We seek $\mathbf{w}$ to minimize $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$
- Taking the gradient with respect to $\mathbf{w}$ (**see course notes for additional details**) and setting it to $\mathbf{0}$, we get:

$$\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}) = \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{t} = \mathbf{0}$$

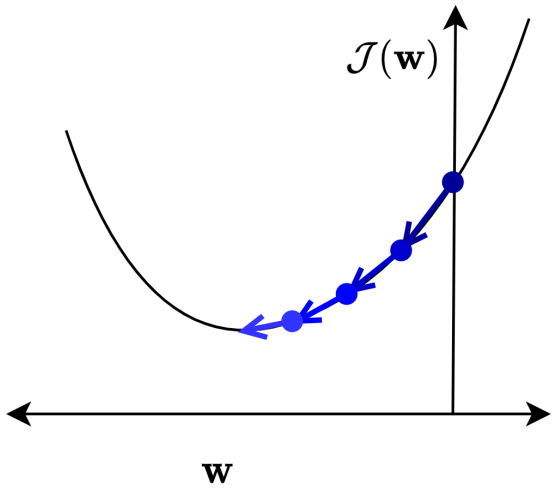
- Optimal weights:

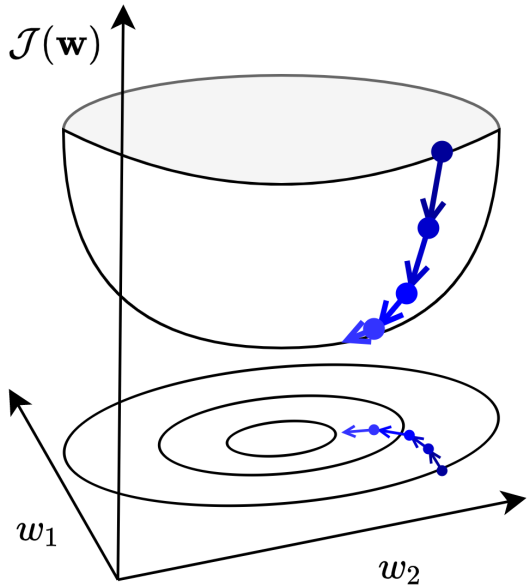
$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

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

Iterative solution: Gradient Descent

- Most optimization problems we cover in this course don't have a direct solution.
- 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 always decreases the cost function for small enough α (unless $\partial \mathcal{J}/\partial w_j = 0$):

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

- $\alpha > 0$ is a learning rate (or step size). 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.
  - If cost is the sum of N individual losses rather than their average, smaller learning rate will be needed ($\alpha' = \alpha/N$).

Gradient Descent

- This gets its name from the gradient. Recall the definition:

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

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

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

  And for linear regression we have:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \, \mathbf{x}^{(i)}$$

- So gradient descent updates $\mathbf{w}$ in the direction of fastest *decrease*.
- Observe that once it converges, we get a critical point, i.e. $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \mathbf{0}$.

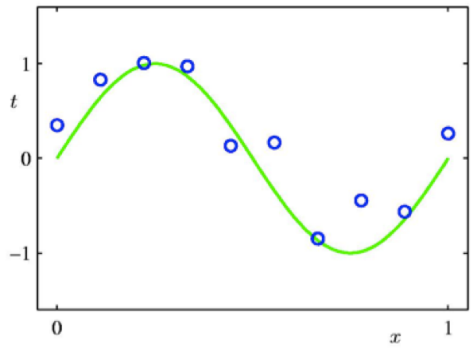
Gradient Descent for Linear Regression

- Even for linear regression, where there is a direct solution, we sometimes need to use GD.
- 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
  - For regression in high-dimensional space, GD is more efficient than direct solution
    - Linear regression solution: $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$
    - Matrix inversion is an $\mathcal{O}(D^3)$ algorithm
    - Each GD update costs $\mathcal{O}(ND)$
    - Or less with stochastic gradient descent (SGD, covered next week)
    - Huge difference if $D \gg 1$

Feature Mappings

Feature Mapping (Basis Expansion)

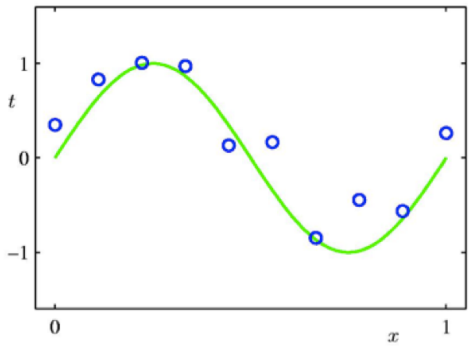
The relation between the input and output may not be linear.



- We can still use linear regression by mapping the input features to another space using feature mapping (or basis expansion). $\boldsymbol{\psi}(\mathbf{x}) : \mathbb{R}^D \to \mathbb{R}^d$ and treat the mapped feature (in $\mathbb{R}^d$) as the input of a linear regression procedure.
- Let us see how it works when $\mathbf{x} \in \mathbb{R}$ and we use a polynomial feature mapping.

Polynomial Feature Mapping

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



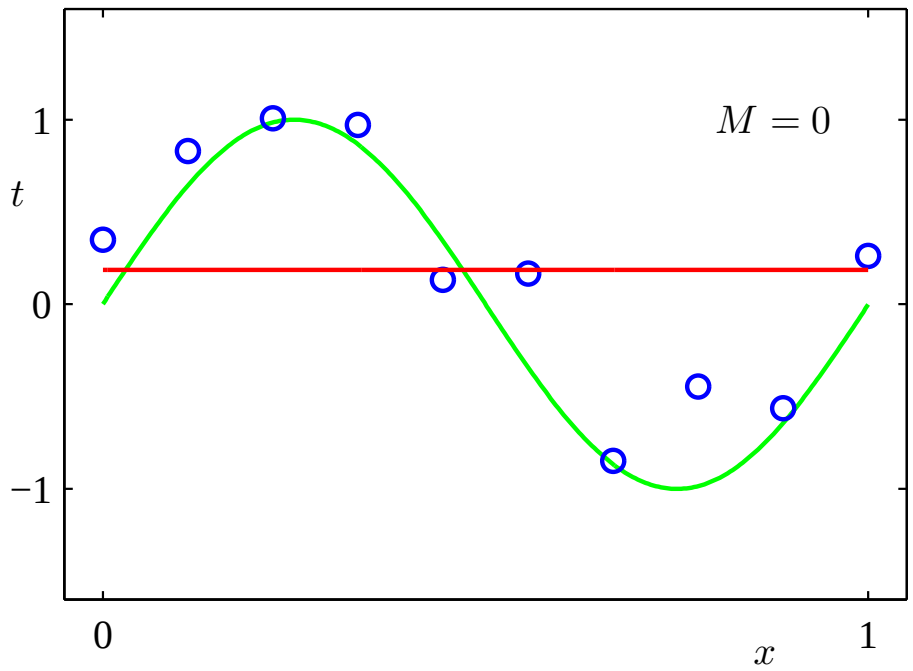
Fit the data using a degree-M polynomial function of the form:

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

- Here the feature mapping is $\boldsymbol{\psi}(x) = [1, x, x^2, ..., x^M]^\top$.
- We can still use linear regression to find $\mathbf{w}$ since $y = \boldsymbol{\psi}(x)^\top \mathbf{w}$ is linear in $w_0, w_1, ...$
- In general, $\boldsymbol{\psi}$ can be any function. Another example: $\boldsymbol{\psi}(x) = [1, \sin(2\pi x), \cos(2\pi x), \sin(4\pi x), ...]^\top$.

Polynomial Feature Mapping with $M = 0$

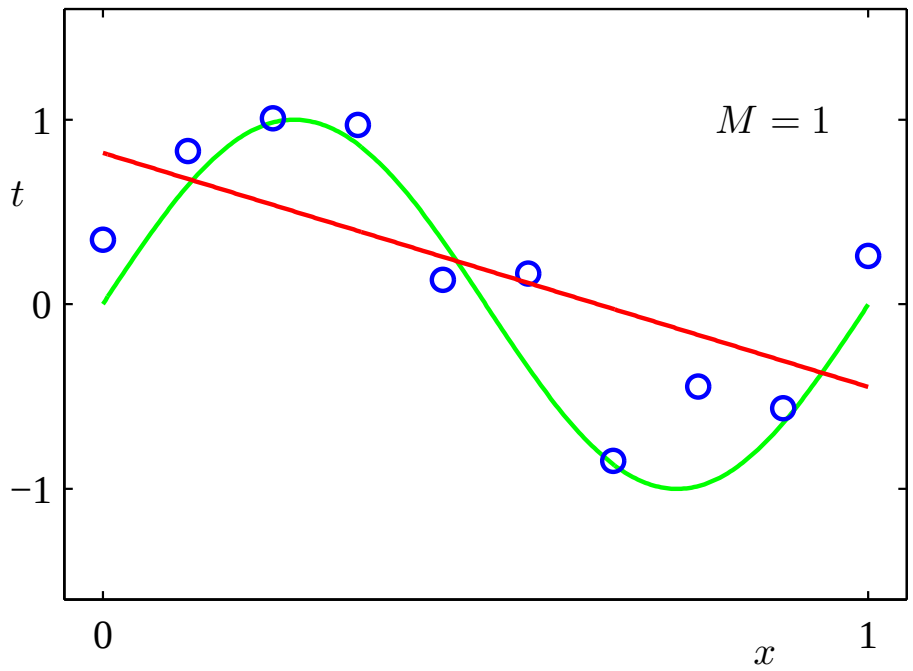
$$y = w_0$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 1$

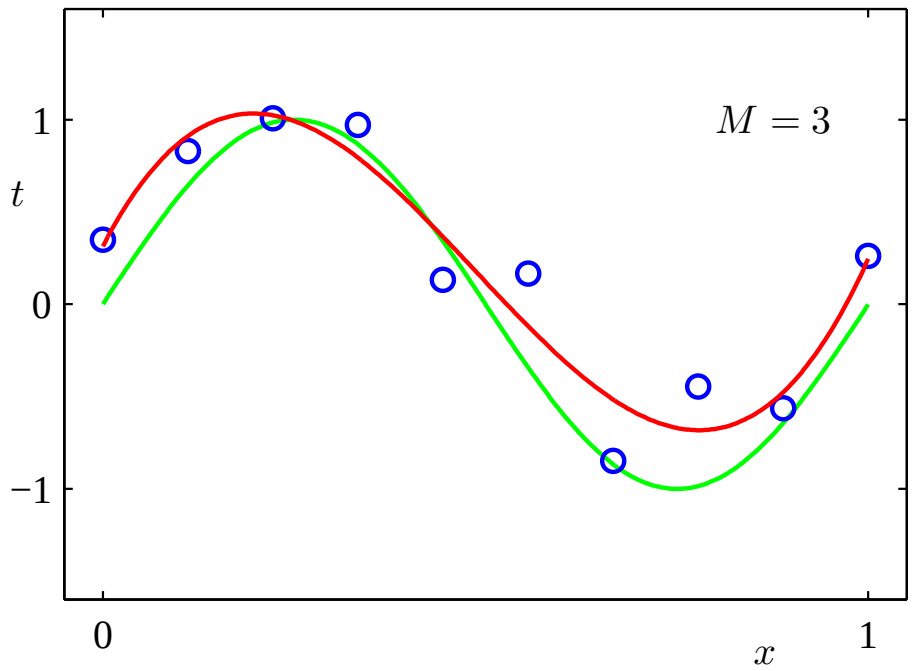
$$y = w_0 + w_1 x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 3$

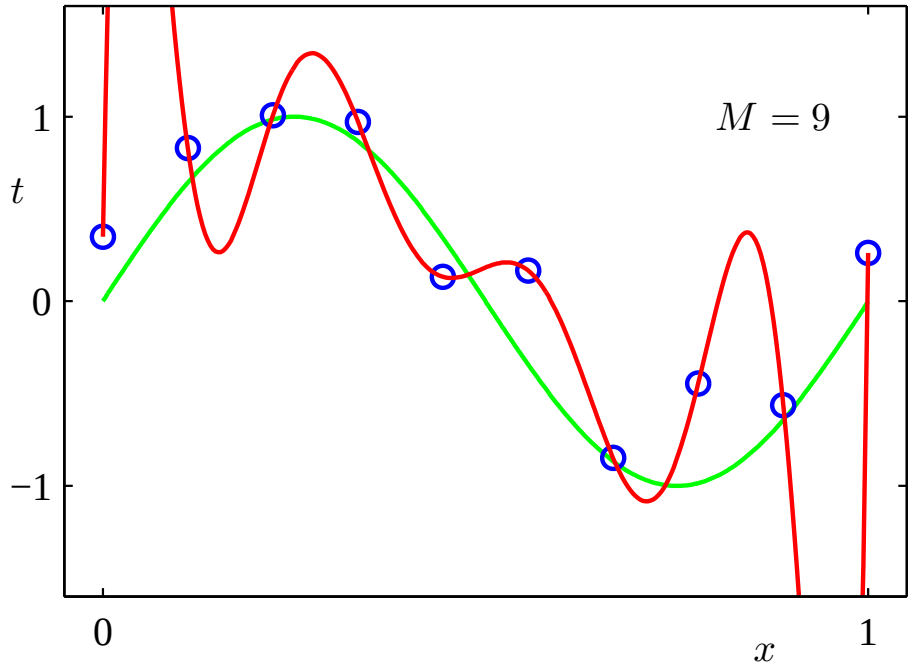
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 9$

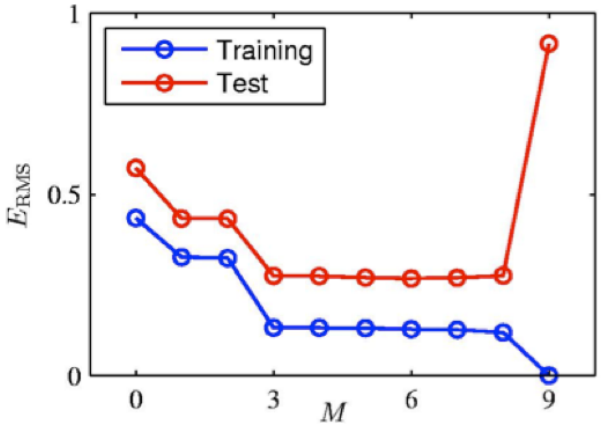
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$

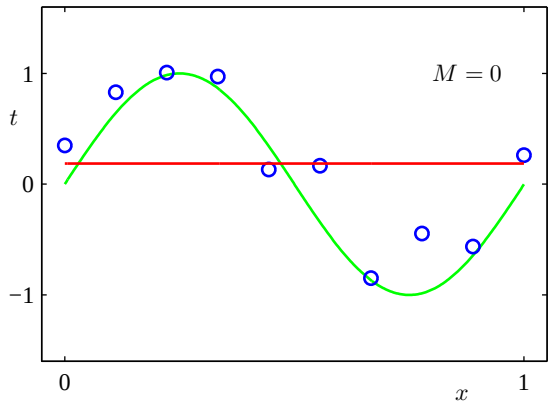


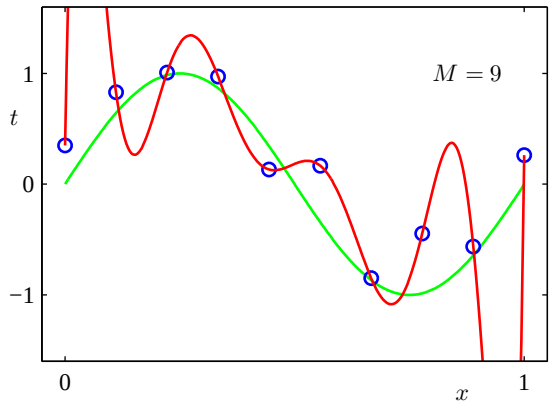
-Pattern Recognition and Machine Learning, Christopher Bishop.

Model Complexity and Generalization

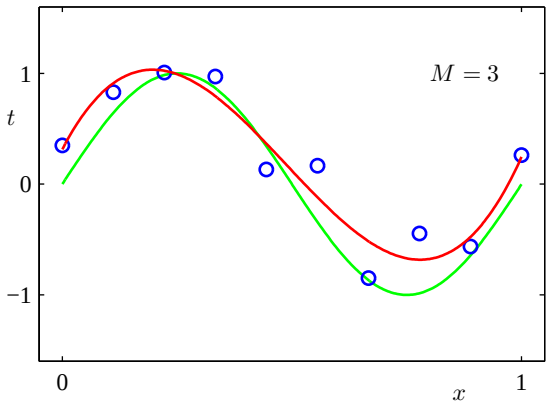
Underfitting (M=0): model is too simple — does not fit the data.
Overfitting (M=9): model is too complex — fits perfectly.





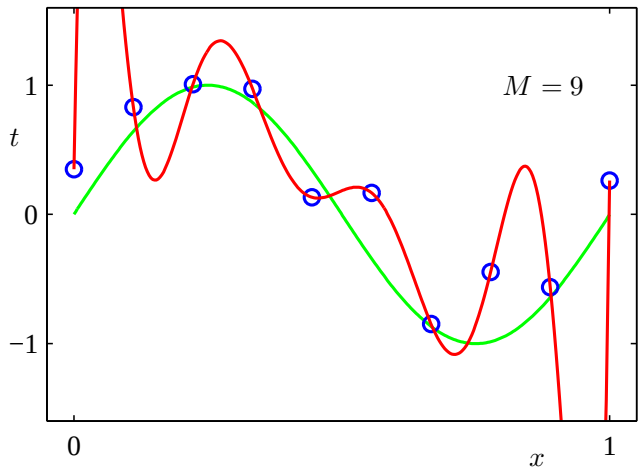


Good model (M=3): Achieves small test error (generalizes well).



Model Complexity and 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

Regularization

- The degree M of the polynomial controls the model's complexity.
- The value of M is a hyperparameter for polynomial expansion, just like k in KNN. We can tune it using a validation set.
- Restricting the number of parameters / basis functions (M) is a crude approach to controlling the model complexity.
- 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.

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

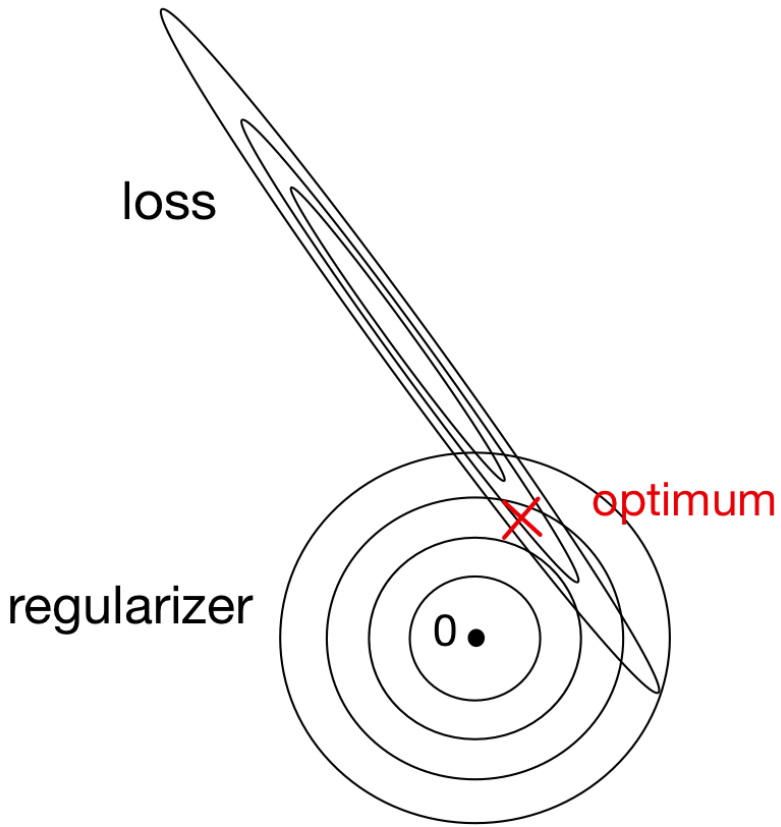
  - Note: To be precise, the L^2 norm $\|\mathbf{w}\|_2$ is Euclidean distance, so we're regularizing the *squared* L^2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

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

- If you fit training data poorly, $\mathcal{J}$ is large. If the weights are large in magnitude, $\mathcal{R}$ is large.
- Large λ penalizes weight values more.
- λ is a hyperparameter we can tune with a validation set.

L^2 (or ℓ_2) Regularization

- The geometric picture:



L^2 Regularized Least Squares: Ridge regression

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

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

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

- The case $\lambda = 0$ (no regularization) reduces to least squares solution!
- Note that it is also common to formulate this problem as $\operatorname{argmin}_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$ in which case the solution is $\mathbf{w}_\lambda^{\text{Ridge}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{t}$.

Gradient Descent under the L^2 Regularization

- Gradient descent update to minimize $\mathcal{J}$:

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

- The gradient descent update to minimize the L^2 regularized cost $\mathcal{J} + \lambda \mathcal{R}$ results in weight decay:

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

Conclusion so far

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
- 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