

# CSC 311: Introduction to Machine Learning

## Lecture 3 - Bagging, Linear Models I

---

Amanjit Singh Kainth

University of Toronto, Summer 2026

# Outline

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization

# Announcements

- HW1 released last week and is due next Friday.
- Go to the earliest possible TA OH you can attend.
- **Manage your time well!** If you wait till the last TA session, you may have a long wait to ask your question.

# Introduction

---

# Bias-Variance Decomposition

---

- 1 Introduction
- 2 Bias-Variance Decomposition**
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization

# Interpretations

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

Bias/variance decomposes the expected loss into three terms:

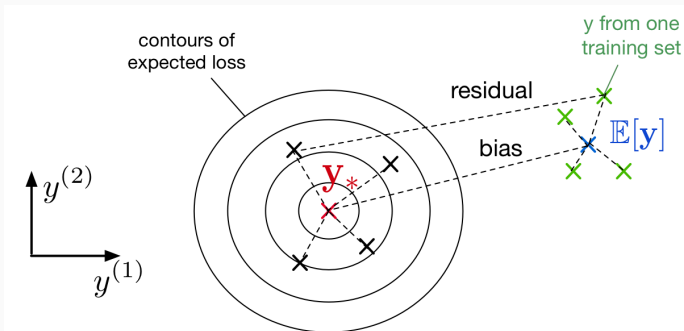
- **bias**: how wrong the expected prediction is (corresponds to under-fitting)
- **variance**: the amount of variability in the predictions (corresponds to over-fitting)
- **Bayes error**: the inherent unpredictability of the targets

Often loosely use “bias” for “under-fitting” and “variance” for “over-fitting”.

# Overly Simple Model

An overly **simple** model (e.g. KNN with large  $k$ ) might have

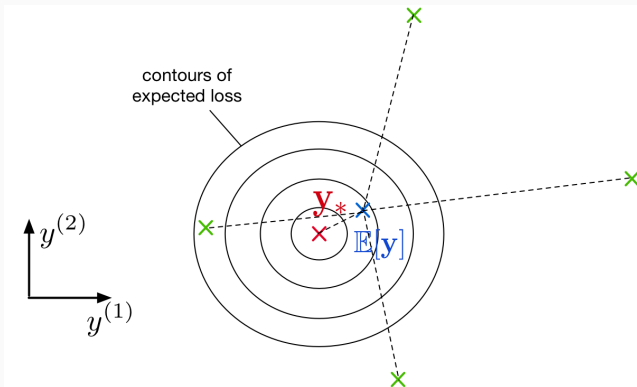
- **high bias**  
(cannot capture the structure in the data)
- **low variance**  
(enough data to get stable estimates)



# Overly Complex Model

An overly **complex** model (e.g. KNN with  $k = 1$ ) might have

- **low bias**  
(learns all the relevant structure)
- **high variance**  
(fits the quirks of the data you happened to sample)



# Bagging

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging**
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization

# Bagging Motivation

- Sample  $m$  independent training sets from  $p_{\text{sample}}$ .
- Compute the prediction  $y_i$  using each training set.
- Compute the average prediction  $y = \frac{1}{m} \sum_{i=1}^m y_i$ .
- How does this affect the three terms of the expected loss?
  - ▶ **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 are averaging over independent predictions

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

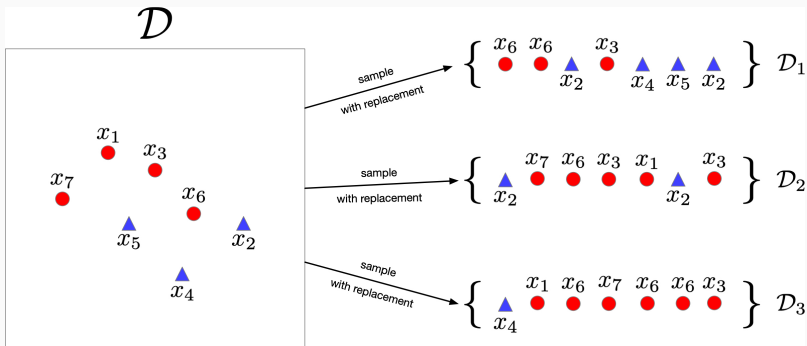
- ▶ **Bayes error:** unchanged,  
since we have no control over it

# Bagging: The Idea

- In practice,  $p_{\text{sample}}$  is often expensive to sample from. So training separate models on independently sampled datasets is very wasteful of data!
- Given training set  $\mathcal{D}$ , use the empirical distribution  $p_{\mathcal{D}}$  as a proxy for  $p_{\text{sample}}$ . This is called **bootstrap aggregation** or **bagging**.
  - ▶ Take a dataset  $\mathcal{D}$  with  $n$  examples.
  - ▶ Generate  $m$  new datasets (“resamples” or “bootstrap samples”)
  - ▶ Each dataset has  $n$  examples sampled from  $\mathcal{D}$  with replacement.
  - ▶ Average the predictions of models trained on the  $m$  datasets.
- One of the most important ideas in statistics!
  - ▶ Intuition: As  $|\mathcal{D}| \rightarrow \infty$ , we have  $p_{\mathcal{D}} \rightarrow p_{\text{sample}}$ .

# Bagging Example 1/2

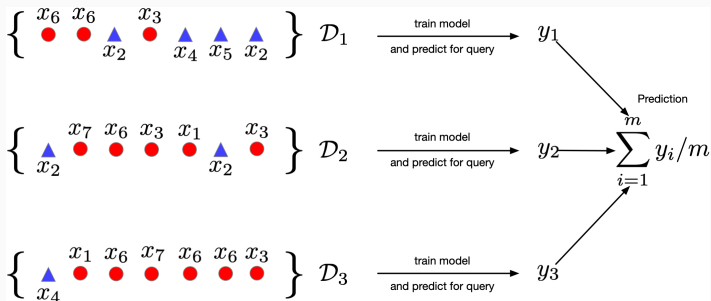
Create  $m = 3$  datasets by sampling from  $\mathcal{D}$  with replacement. Each dataset contains  $n = 7$  examples.



## Bagging Example 2/2

Generate prediction  $y_i$  using dataset  $\mathcal{D}_i$ .

Average the predictions.



## Aggregating Predictions for Binary Classification

- Classifier  $i$  outputs a prediction  $y_i$
- $y_i$  can be real-valued  $y_i \in [0, 1]$  or a binary value  $y_i \in \{0, 1\}$
- Average the predictions and apply a threshold.

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

- Same as majority vote.

# Bagging Properties

- A bagged classifier can be stronger than the average model.
  - ▶ E.g. on “Who Wants to be a Millionaire”, “Ask the Audience” is much more effective than “Phone a Friend”.
- But, if  $m$  datasets are NOT independent, don't get the  $\frac{1}{m}$  variance reduction.
- Reduce correlation between datasets by introducing *additional variability*
  - ▶ Invest in a diversified portfolio, not just one stock.
  - ▶ Average over multiple algorithms, or multiple configurations of the same algorithm.

- A trick to reduce correlation between bagged decision trees:  
For each node, choose a random subset of features  
and consider splits on these features only.
- Probably the best black-box machine learning algorithm.
  - ▶ works well with no tuning.
  - ▶ widely used in Kaggle competitions.

# Bagging Summary

Reduces over-fitting by averaging predictions.

In most competition winners.

A small ensemble often better than a single great model.

Limitations:

- Does not reduce bias in case of squared error.
- Correlation between classifiers means less variance reduction.  
Add more randomness in Random Forests.
- Weighting members equally may not be the best.  
Weighted ensembling often leads to better results if members are very different.

# Linear Regression

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression**
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization

## Step 1 & 2: Linear Regression

- Define the task and a strategy on solving it
- **Task**: predict scalar-valued targets (e.g. stock prices)
- **Architecture**: linear function of the inputs

## Step 3: A Modular Approach to ML

- choose a **model** describing relationships between variables
- define a **loss function** quantifying how well the model fits the data
- choose a **regularizer** expressing preference over different models
- fit a model that minimizes the loss function and satisfies the regularizer's constraint/penalty, possibly using an **optimization algorithm**

Mixing and matching these modular components give us a lot of different ML methods.

# Supervised Learning Setup

- Input  $\mathbf{x} \in \mathcal{X}$  (a vector of features)
- Target  $t \in \mathcal{T}$
- Data  $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$
- Objective: learn a function  $f : \mathcal{X} \rightarrow \mathcal{T}$  based on the data such that  $t \approx y = f(\mathbf{x})$

# Model

**Model:** a *linear* function of the features  $\mathbf{x} = (x_1, \dots, x_D) \in \mathbb{R}^D$  to make prediction  $y \in \mathbb{R}$  of the target  $t \in \mathbb{R}$ :

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

- **Parameters** are weights  $\mathbf{w}$  and the bias/intercept  $b$
- Want the prediction to be close to the target:  $y \approx t$ .
- Highly interpretable model, useful for debugging.

# Loss Function

**Loss function**  $\mathcal{L}(y, t)$  defines how badly the algorithm's prediction  $y$  fits the target  $t$  for some example  $\mathbf{x}$ .

**Squared error loss function:**  $\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$

- $y - t$  is the **residual**, and we want to minimize this magnitude
- $\frac{1}{2}$  makes calculations convenient.

**Cost function:** loss function averaged over all training examples also called *empirical* or *average loss*.

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

# Vectorization

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization**
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization

## Loops v.s. Vectorized Code

- We can compute prediction for one data point using a for loop:

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

- But, excessive super/sub scripts are hard to work with, and Python loops are slow.
- Instead, we express algorithms using vectors and matrices.

$$\mathbf{w} = (w_1, \dots, w_D)^\top \quad \mathbf{x} = (x_1, \dots, x_D)^\top$$
$$y = \mathbf{w}^\top \mathbf{x} + b$$

- This is simpler and executes much faster:

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

# Benefits of Vectorization

Why vectorize?

- The code is simpler and more readable. No more 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

You will practice switching in and out of vectorized form.

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

## Predictions for the Dataset

- Put training examples into a **design matrix**  $\mathbf{X}$ .
- Put targets into the **target vector**  $\mathbf{t}$ .
- We can compute the predictions for the whole dataset.

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

$$\begin{pmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix} + b \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix}$$

## Computing Squared Error Cost

We can compute 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!).

## Combining Bias and Weights

We can combine the bias and the weights and add a column of 1's to design matrix.

Our predictions become

$$\mathbf{y} = \mathbf{X}\mathbf{w}.$$

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

# Optimization

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization**
- 7 Feature Mappings
- 8 Regularization

## Step 4: Solving the Minimization Problem

Goal is to minimize the cost function  $\mathcal{J}(\mathbf{w})$ .

Recall: the minimum of a smooth function (if it exists) occurs at a **critical point**, i.e. point where the derivative is zero.

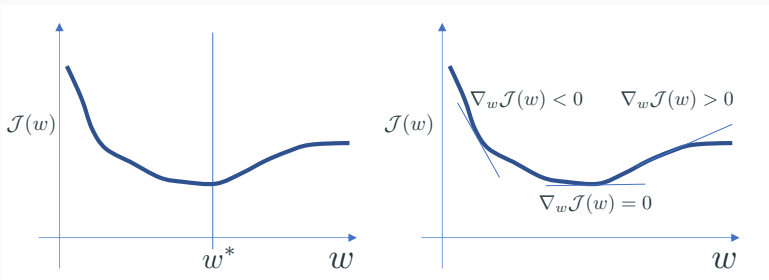
$$\nabla_{\mathbf{w}}\mathcal{J} = \frac{\partial\mathcal{J}}{\partial\mathbf{w}} = \begin{pmatrix} \frac{\partial\mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial\mathcal{J}}{\partial w_D} \end{pmatrix}$$

Solutions may be direct or iterative.

- **Direct solution**: set the gradient to zero and solve in closed form — directly find provably optimal parameters.
- **Iterative solution**: repeatedly apply an update rule that gradually takes us closer to the solution.

# Minimizing 1D Function

- Consider  $\mathcal{J}(w)$  where  $w$  is 1D.
- Seek  $w = w^*$  to minimize  $\mathcal{J}(w)$ .
- The gradients can tell us where the maxima and minima of functions lie
- **Strategy:** Write down an algebraic expression for  $\nabla_w \mathcal{J}(w)$ . Set  $\nabla_w \mathcal{J}(w) = 0$ . Solve for  $w$ .



# Direct Solution for Linear Regression

- 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  $\mathbf{w}$  and setting it to  $\mathbf{0}$ , we get:

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

See course notes for derivation.

- Optimal weights:

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

- Few models (like linear regression) permit direct solution.

# Iterative Solution: Gradient Descent

- Many optimization problems don't have a direct solution.
- A more broadly applicable strategy is **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**.

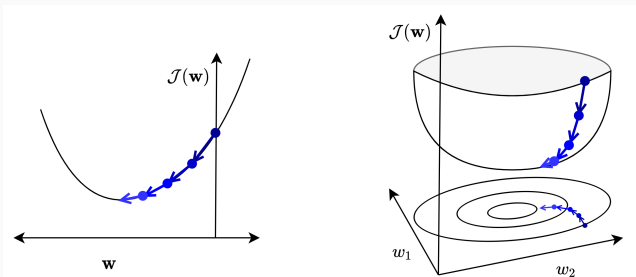
# Deriving Update Rule

Observe:

- if  $\partial\mathcal{J}/\partial w_j > 0$ , then decreasing  $\mathcal{J}$  requires decreasing  $w_j$ .
- if  $\partial\mathcal{J}/\partial w_j < 0$ , then decreasing  $\mathcal{J}$  requires increasing  $w_j$ .

The following update always decreases the cost function for small enough  $\alpha$  (unless  $\partial\mathcal{J}/\partial w_j = 0$ ):

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



# Setting Learning Rate

Gradient descent update rule:

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

$\alpha > 0$  is a **learning rate** (or step size).

- The larger  $\alpha$  is, the faster  $\mathbf{w}$  changes.
- Values are typically small, e.g. 0.01 or 0.0001.
- We'll see later how to tune the learning rate.
- If minimizing total loss rather than average loss, needs a smaller learning rate ( $\alpha' = \alpha/N$ ).

# Gradient Descent Intuition

- Gradient descent gets its name from the gradient, the direction of fastest *increase*.

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

- Update rule in vector form:

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

Update rule for linear regression:

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

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

# Why Use Gradient Descent?

- Applicable to a much broader set of models.
- Easier to implement than direct solutions.
- More efficient than direct solution for regression in high-dimensional space.
  - ▶ The linear regression direct solution  $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$  requires matrix inversion, which is  $\mathcal{O}(D^3)$ .
  - ▶ Gradient descent update costs  $\mathcal{O}(ND)$  or less with stochastic gradient descent.
  - ▶ Huge difference if  $D$  is large.

## Feature Mappings

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings**
- 8 Regularization

# Steps for linear regression

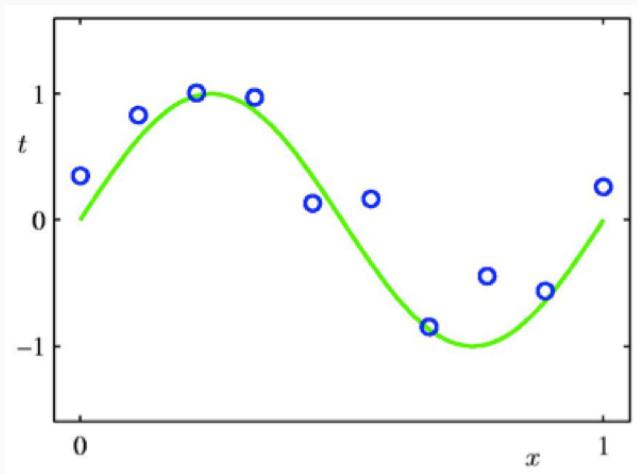
Below is a categorization of ML problems that you will see time, and time-again throughout this semester.

- Step 1: Understand the problem (is it prediction, learning a good representation). **Regression**
- Step 2: Formulate the problem mathematically (create notation for your inputs and outcomes and model). **Linear function of inputs**
- Step 3: Formulate an objective function that represents success for your model. **Mean squared error**
- Step 4: Find a strategy to solve the optimization problem on pencil and paper. **Direct or gradient based optimization**
- Step 5: Translate the algorithm into code. **Part of future homework excercises**
- Step 6: Analyze, iterate, improve design choices in your model and algorithm

Can we use linear regression to model a non-linear relationship?

- Map the input features to another space  $\psi(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}^d$ .
- Treat the mapped feature (in  $\mathbb{R}^d$ ) as the input of a linear regression procedure.

## Modeling a Non-Linear Relationship

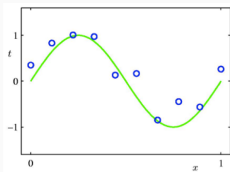


# Polynomial Feature Mapping

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

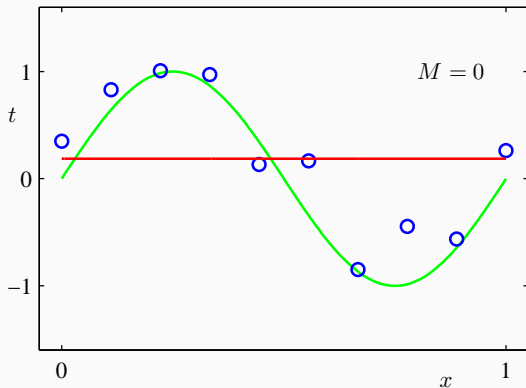
$$y = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{i=0}^M w_ix^i$$

- The feature mapping is  $\boldsymbol{\psi}(x) = [1, x, x^2, \dots, x^M]^\top$ .
- $y = \boldsymbol{\psi}(x)^\top \mathbf{w}$  is linear in  $w_0, w_1, \dots$
- Use linear regression to find  $\mathbf{w}$ .



# Polynomial Feature Mapping with $M = 0$

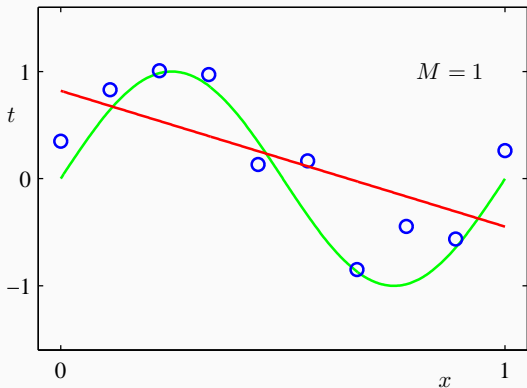
$$y = w_0$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

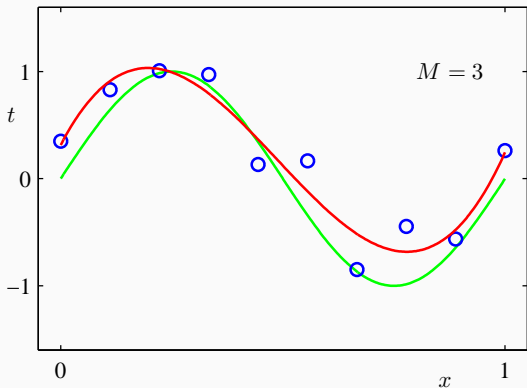
# Polynomial Feature Mapping with $M = 1$

$$y = w_0 + w_1x$$



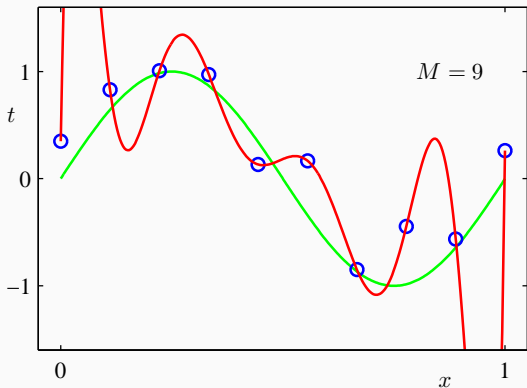
# Polynomial Feature Mapping with $M = 3$

$$y = w_0 + w_1x + w_2x^2 + w_3x^3$$



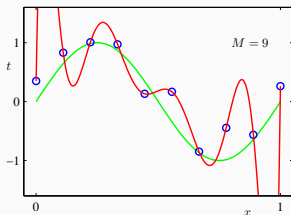
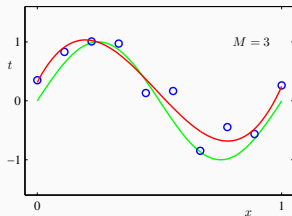
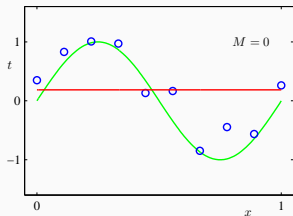
# Polynomial Feature Mapping with $M = 9$

$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$



[Pattern Recognition and Machine Learning, Christopher Bishop.]

# Model Complexity and Generalization



## Under-fitting

( $M = 0$ ):

Model is too simple,  
doesn't fit data well.

## Good model

( $M = 3$ ):

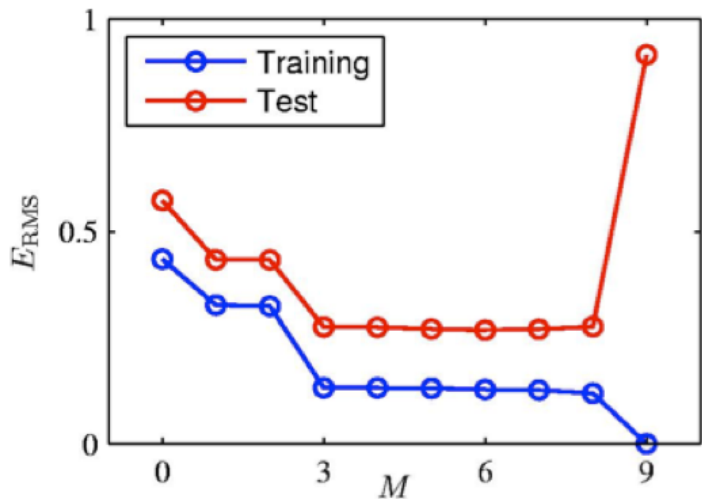
Small test error,  
generalizes well.

## Over-fitting

( $M = 9$ ):

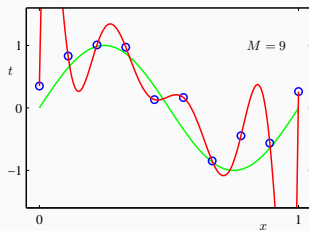
Model is too complex,  
fits data perfectly. 46

# Model Complexity and Generalization



# Model Complexity and Generalization

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

---

- 1 Introduction
- 2 Bias-Variance Decomposition
- 3 Bagging
- 4 Linear Regression
- 5 Vectorization
- 6 Optimization
- 7 Feature Mappings
- 8 Regularization**

# Controlling Model Complexity

How can we control the model complexity?

- A crude approach: restrict # of parameters / basis functions. For polynomial expansion, tune  $M$  using a validation set.
- Another approach: **regularize** the model. **Regularizer** is a function that quantifies how much we prefer one hypothesis vs. another.

## $L^2$ (or $\ell_2$ ) Regularization

- Encourage the weights to be small by choosing the  $\ell_2$  **penalty** as our regularizer.

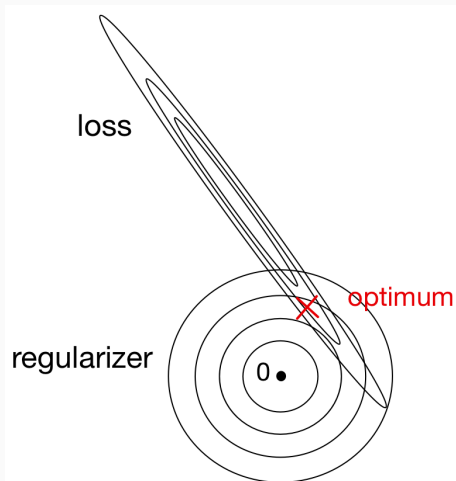
$$\mathcal{R}(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|_2^2 = \frac{1}{2} \sum_j w_j^2.$$

- The regularized cost function makes a trade-off between the 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 the model fits training data poorly,  $\mathcal{J}$  is large. If the weights are large in magnitude,  $\mathcal{R}$  is large.
- Large  $\lambda$  penalizes weight values more.
- Tune hyperparameter  $\lambda$  with a validation set.

# $L^2$ Regularization Picture



## $L^2$ Regularized Least Squares: Ridge regression

For the least squares problem, we have  $\mathcal{J}(\mathbf{w}) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2$ .

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

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

- $\lambda = 0$  (no regularization) reduces to least squares solution!
- Can also formulate the problem as

$$\underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

with solution

$$\mathbf{w}_\lambda^{\text{Ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{t}.$$

## 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**:

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

# Conclusions

Linear regression exemplifies recurring themes of this course:

- choose a **model** and a **loss function**
- formulate an **optimization problem**
- solve the minimization problem using **direction solution** or **gradient descent**.
- **vectorize** the algorithm, i.e. represent in terms of linear algebra
- make a linear model more powerful using **feature mappings**
- improve the generalization by adding a **regularizer**