

CSC 311: Introduction to Machine Learning

Lecture 3 - Bagging, Linear Models I

Rahul G. Krishnan Alice Gao

University of Toronto, Fall 2022

Outline

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

Announcements

- HW1 is due next Monday (10% late penalty for each late day, no credit after 3 days).
- We have arranged TA office hours (on website) for the assignment.
- Go to the earliest possible ones you can attend.
- **Manage your time well!** If you wait till the last TA session, you may a long wait to ask your question.

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.

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

Bias/Variance Decomposition

- prediction y at a query \mathbf{x} is a random variable (where the randomness comes from the choice of dataset),
- y_\star is the optimal deterministic prediction, and
- 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 Decomposition

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

Bias/variance decomposes the expected loss into three terms:

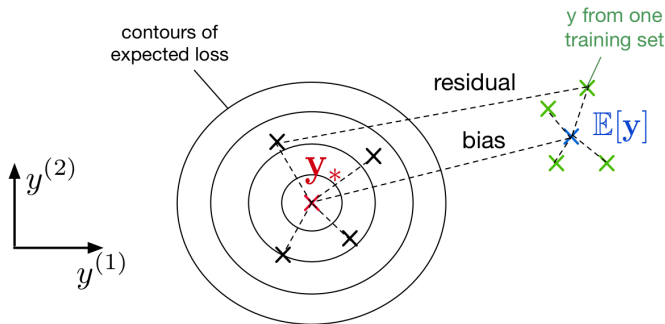
- **bias**: how wrong the expected prediction is
(corresponds to under-fitting)
- **variance**: the amount of variability in the predictions
(corresponds to over-fitting)
- Bayes error: the inherent unpredictability of the targets

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

Visualizing Bias/Variance Decomposition

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

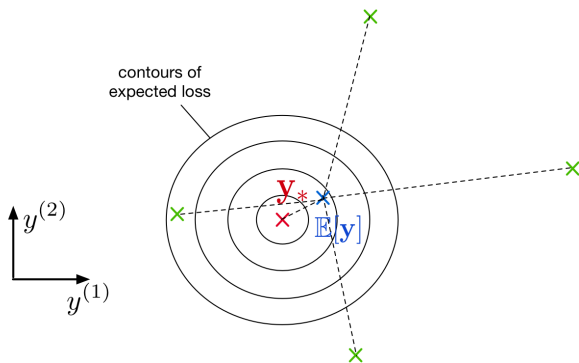
- **high bias**
(cannot capture the structure in the data)
- **low variance**
(enough data to get stable estimates)



Visualizing Bias/Variance Decomposition

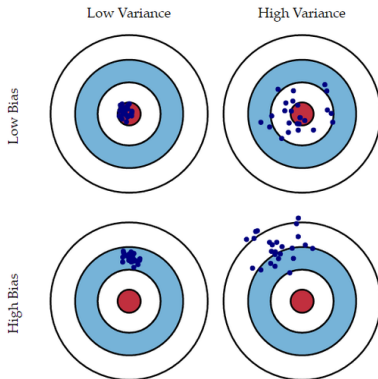
An overly **complex** model (e.g. KNN with $k = 1$) might have

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



Bias/Variance Decomposition: Another Visualization

- The following graphic summarizes the previous two slides:



A: Bayes error

- 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_{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

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

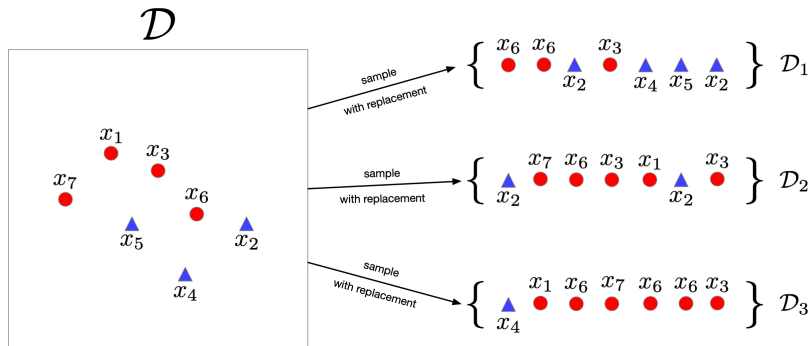
- ▶ **Bayes error:** unchanged,
since we have no control over it

Bagging: The Idea

- In practice, p_{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_{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

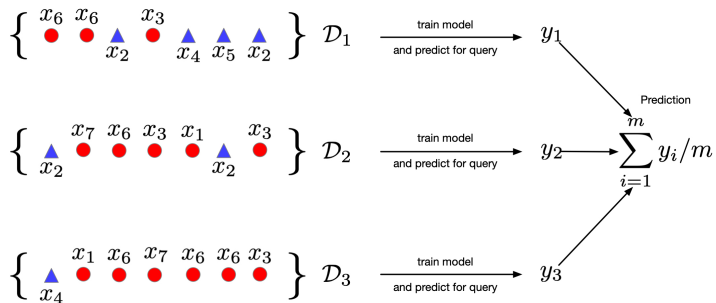
Create $m = 3$ datasets by sampling from D with replacement.
Each dataset contains $n = 7$ examples.



Bagging Example 2/2

Generate prediction y_i using dataset 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.

Random Forests

- 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:

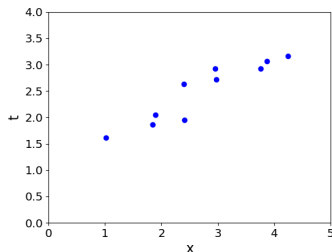
- Does not reduce bias in case of squared error.
- Correlation between classifiers means less variance reduction.
Add more randomness in Random Forests.
- Weighting members equally may not be the best.
Weighted ensembling often leads to better results if members are very different.

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

Overview

- Third 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 $f : \mathcal{X} \rightarrow \mathcal{T}$ such that $t \approx y = f(\mathbf{x})$ based on some data $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$.

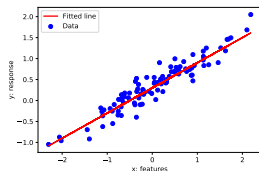
Linear Regression - Model

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

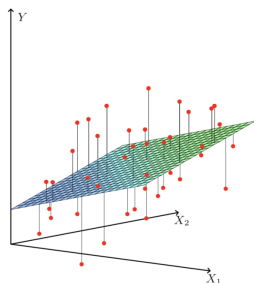
$$y = f(\mathbf{x}) = \sum_j w_j x_j + b$$

- ▶ 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*.

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

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.

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

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

- This is simpler and executes much faster:

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

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

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

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

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

Solving the Minimization Problem

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

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.

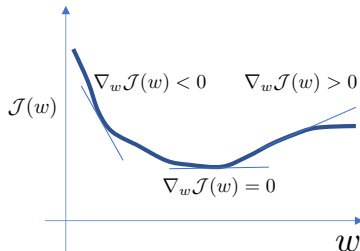
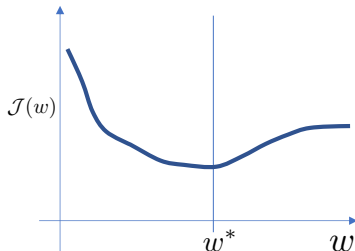
$$\nabla_{\mathbf{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

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.

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

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

See course notes for additional details.

- 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.

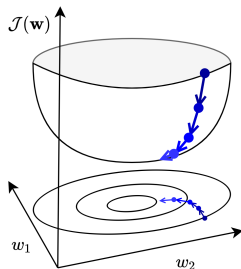
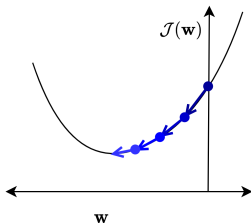
Iterative solution: Gradient Descent

- Many optimization problems don't have a direct solution.
- A more broadly applicable way to minimize the cost function: **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 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 α (unless $\partial \mathcal{J} / \partial w_j = 0$):

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

Gradient Descent

- 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 α 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 cost is the total loss rather than average loss, a smaller learning rate will be needed ($\alpha' = \alpha/N$).

Gradient Descent

- Gradient descent gets its name from the gradient.
Recall the definition of the gradient:

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

The gradient is the direction of fastest *increase* in \mathcal{J} .

- Update rule in vector form:

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

- 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. $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \mathbf{0}$.

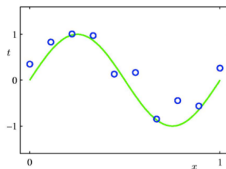
Why Use Gradient Descent?

- GD is applicable to a much broader set of models
- GD is easier to implement than direct solutions
- For regression in high-dimensional space, GD is more efficient than direct solution
 - ▶ For example, the linear regression direct solution $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$ requires matrix inversion, which is $\mathcal{O}(D^3)$.
 - ▶ Each GD update costs $\mathcal{O}(ND)$ or less with stochastic gradient descent.
 - ▶ Huge difference if $D \gg 1$

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

Feature Mapping (Basis Expansion)

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



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

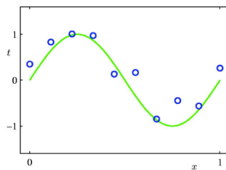
Polynomial Feature Mapping

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

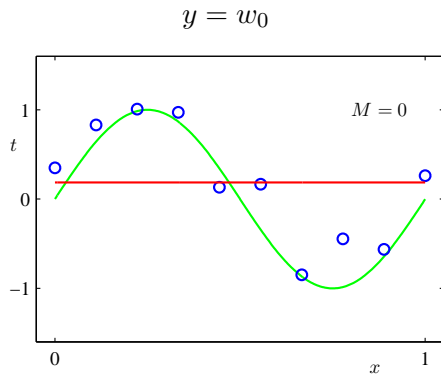
$$y = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{i=0}^M w_i x^i$$

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

Another example: $\psi(x) = [1, \sin(2\pi x), \cos(2\pi x), \sin(4\pi x), \dots]^\top$.

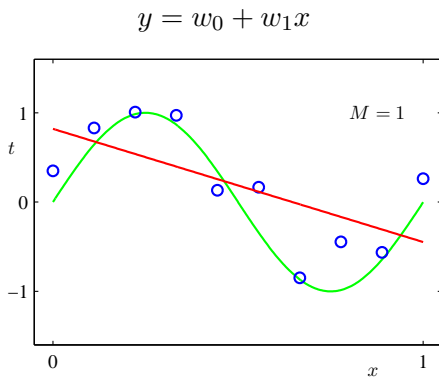


Polynomial Feature Mapping with $M = 0$



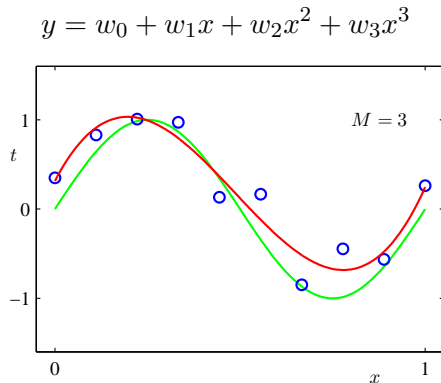
-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 1$



-Pattern Recognition and Machine Learning, Christopher Bishop.

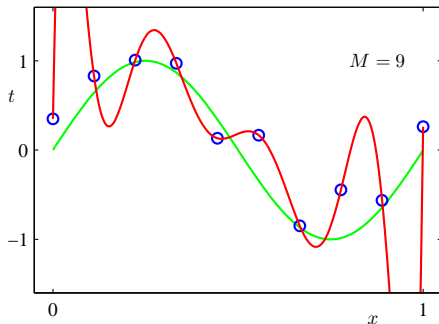
Polynomial Feature Mapping with $M = 3$



-Pattern Recognition and Machine Learning, Christopher Bishop.

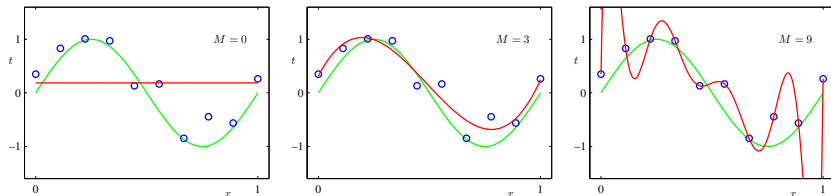
Polynomial Feature Mapping with $M = 9$

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



-Pattern Recognition and Machine Learning, Christopher Bishop.

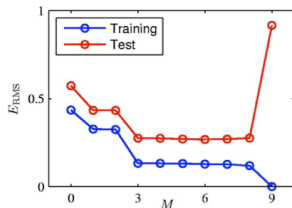
Model Complexity and Generalization



Under-fitting ($M=0$): Model is too simple, does not fit the data.

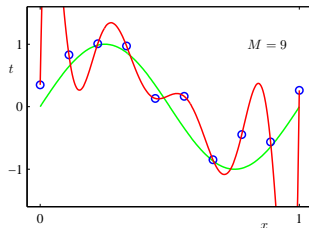
Good model ($M=3$): Achieves small test error, generalizes well.

Over-fitting ($M=9$): Model is too complex, fits perfectly.



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.

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

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

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

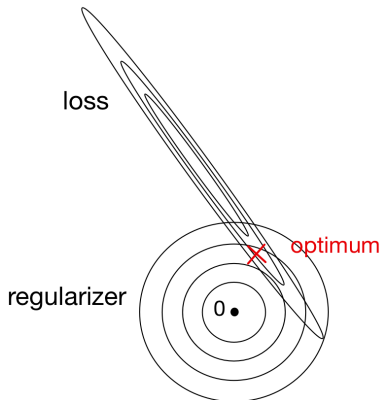
- The regularized cost function makes a tradeoff between the fit to the data and the norm of the weights.

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

- If 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

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

- The case $\lambda = 0$ (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**:

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

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**