STA 314: Statistical Methods for Machine Learning I

Lecture 4 - Ensembles & Linear Regression

Chris J. Maddison

University of Toronto

Today

- We will cover ensembling methods that combine multiple models and can perform better than the individual members.
  - We've seen individual models (KNN, decision trees)
- We will study bagging in particular, which trains models independently on random "resamples" of the training data.
- We will start our study of linear predictors, starting with linear regression.
- Highly recommend the course notes on the suggested reading: linear regression and calculus.

Bagging: Motivation

- What if we could somehow sample m independent training sets from p_{data}?
- In the previous discussion, we would have picked a separate predictor

$$\hat{y}_n^\star = \arg\min_{y \in \mathcal{H}} \hat{\mathcal{R}}[y, \mathcal{D}_n^{train}]$$

  averaged the losses $L(\hat{y}_n^\star(\mathbf{x}^{(i)}), t^{(i)})$ on the test set.
- What if instead we used the average prediction?

$$\bar{y}^\star(\mathbf{x}) = \frac{1}{m} \sum_{n=1}^{m} \hat{y}_n^\star(\mathbf{x})$$

Bagging: Motivation

- 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

$$\mathbb{E}[\bar{y}^\star] = \mathbb{E}\left[\frac{1}{m}\sum_{n=1}^{m}\hat{y}_n^\star\right] = \mathbb{E}[\hat{y}_n^\star]$$

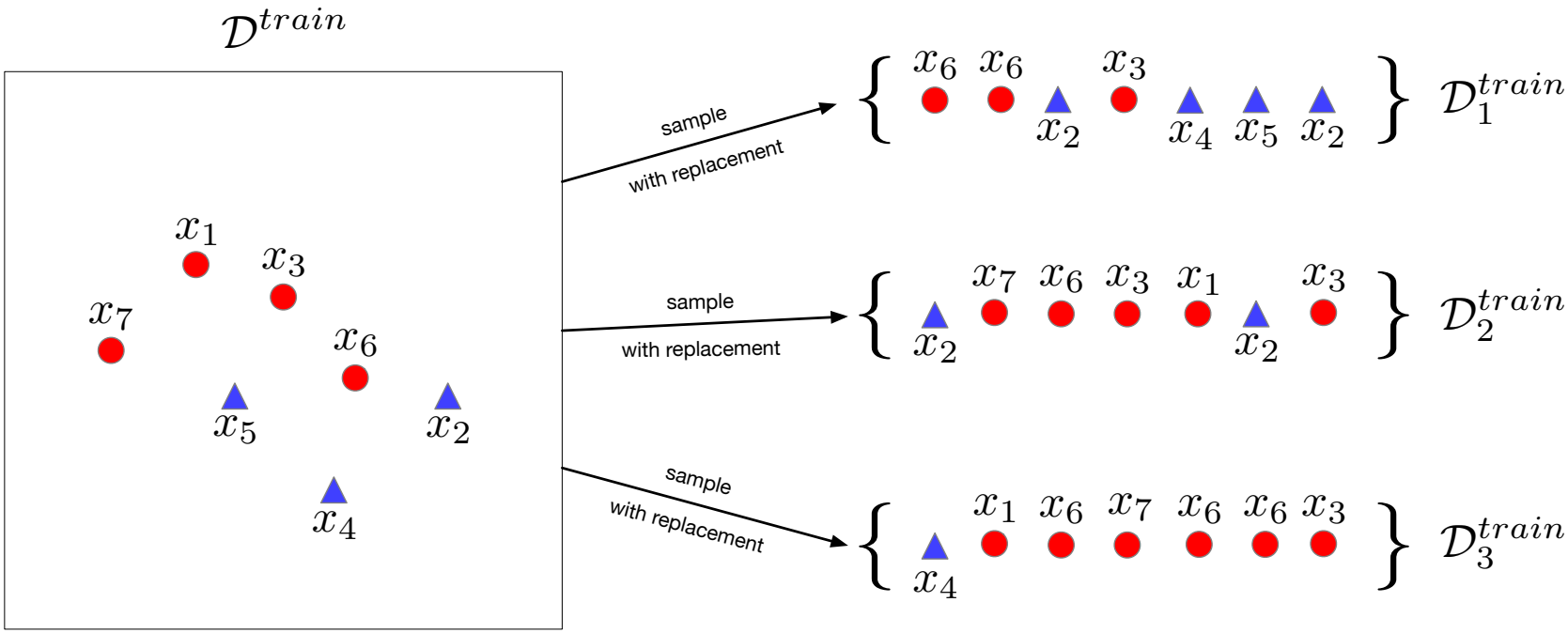
  - **Variance:** reduced, since we're averaging over independent samples

$$\mathrm{Var}[\bar{y}^\star] = \mathrm{Var}\left[\frac{1}{m}\sum_{n=1}^{m}\hat{y}_n^\star\right] = \frac{1}{m^2}\sum_{n=1}^{m}\mathrm{Var}[\hat{y}_n^\star] = \frac{1}{m}\mathrm{Var}[\hat{y}_n^\star].$$

Bagging: The Idea

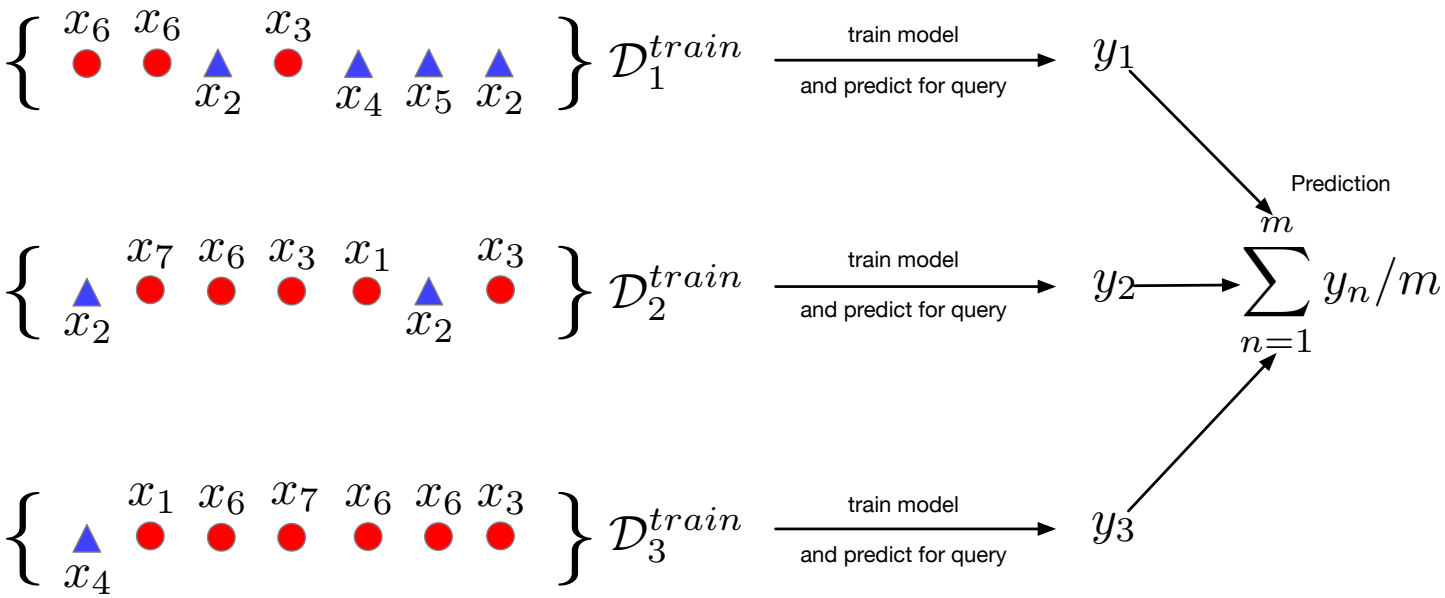
- In practice, the sampling distribution p_{data} is often finite or expensive to sample from.
- So training separate models on independently sampled datasets is very wasteful of data!
  - Why not train a single model on the union of all sampled datasets?
- Solution: given training set $\mathcal{D}^{train}$, use the empirical distribution $p_{\mathcal{D}^{train}}$ as a proxy for p_{data}. This is called bootstrap aggregation, or bagging .
  - Take a single dataset $\mathcal{D}^{train}$ with N examples.
  - Generate m new datasets ("resamples" or "bootstrap samples"), each by sampling N training examples from $\mathcal{D}^{train}$, 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!

Bagging



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

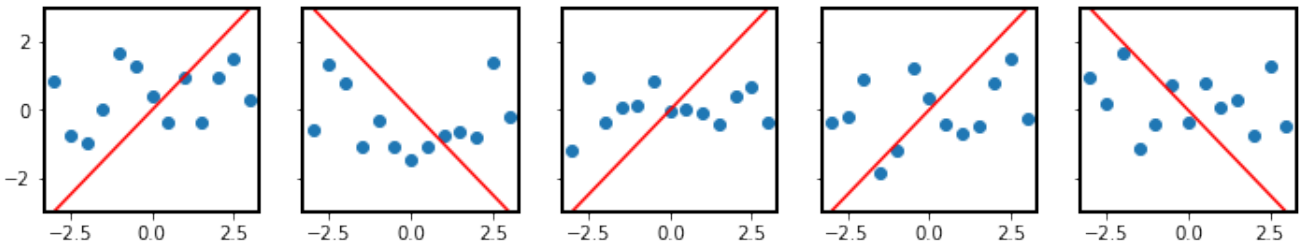
Bagging



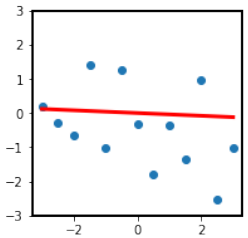
predicting on a query point x

Bagging: Effect on Hypothesis Space

- We saw that in case of squared error, bagging does not affect bias.
- But it can change the hypothesis space $\mathcal{H}$.
- Illustrative example:
  - $x \sim \mathcal{U}(-3, 3)$, $t \sim \mathcal{N}(0, 1)$
  - $\mathcal{H} = \{wx \mid w \in \{-1, 1\}\}$
  - Sampled datasets & fitted hypotheses:



  - Ensembled hypotheses (mean over 1000 samples):



    - The ensembled hypothesis is not in the original hypothesis space!

- This effect is often more pronounced when combining classifiers.

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

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

- 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

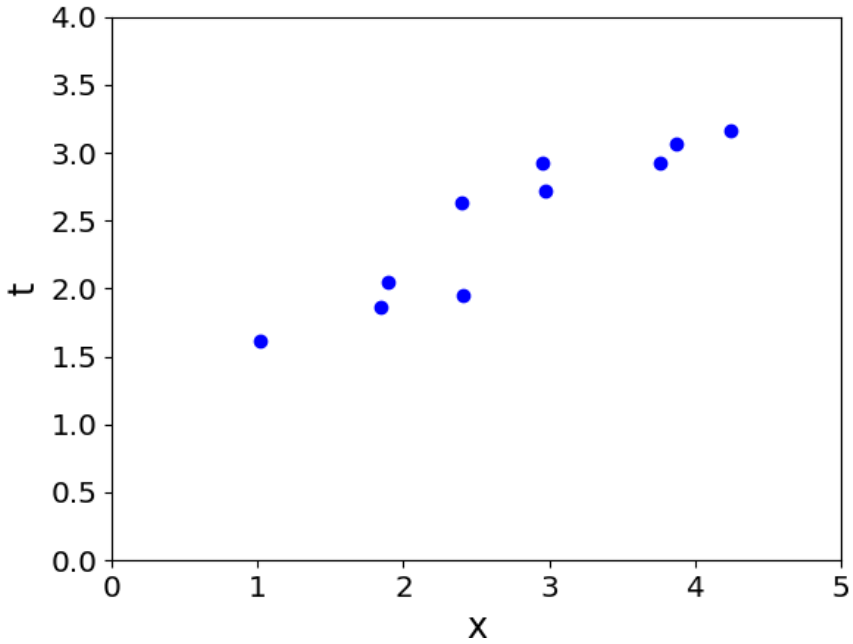
Bagging Summary

- 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.

Overview

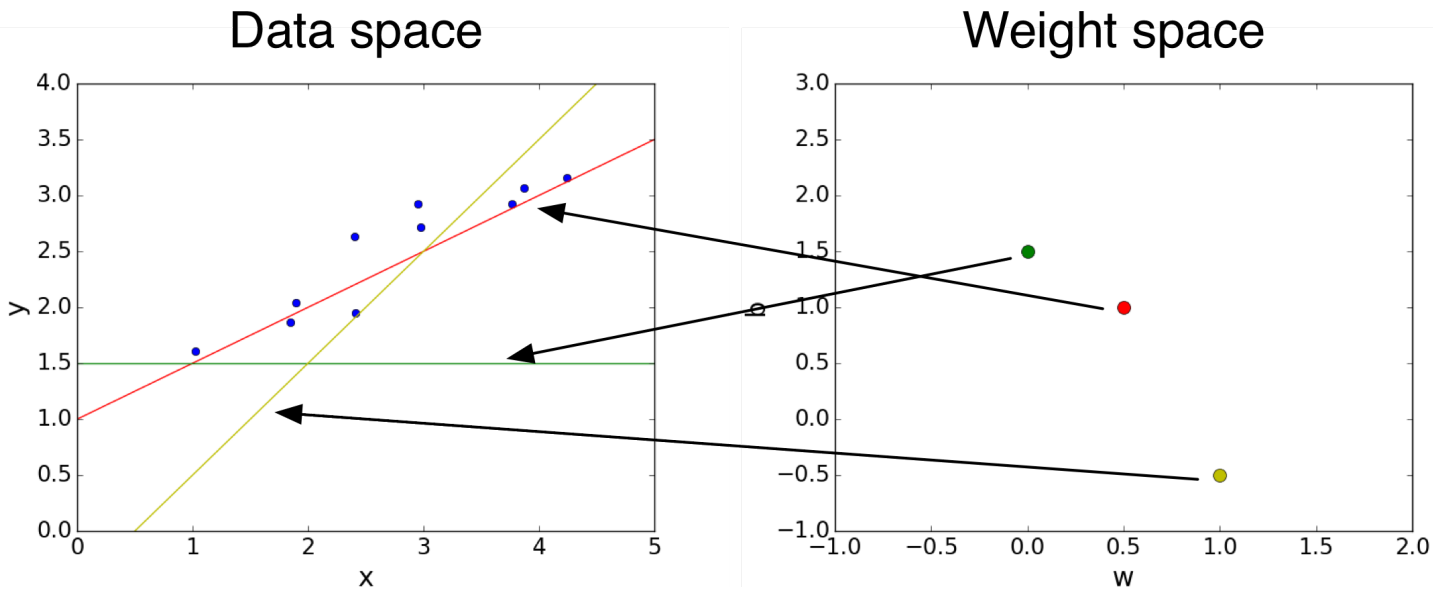
- So far, we've talked about *procedures* for learning.
  - KNN, decision trees, bagging
- For the remainder of this course, we'll take a more modular approach:
  - choose a model describing the relationships between variables of interest
  - define a loss function quantifying how bad is the fit to the 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 combinatorially more powerful!

Problem Setup



- Want to predict a scalar t as a function of a scalar x
- Given a dataset of pairs $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^{N}$
- The $\mathbf{x}^{(i)}$ are inputs, and the $t^{(i)}$ are targets.

Problem Setup



- Model: y is a linear function of x:

$$y = wx + b$$

- y is the prediction
- w is the weight
- b is the bias
- w and b together are the parameters
- Settings of the parameters are the hypotheses

Problem Setup

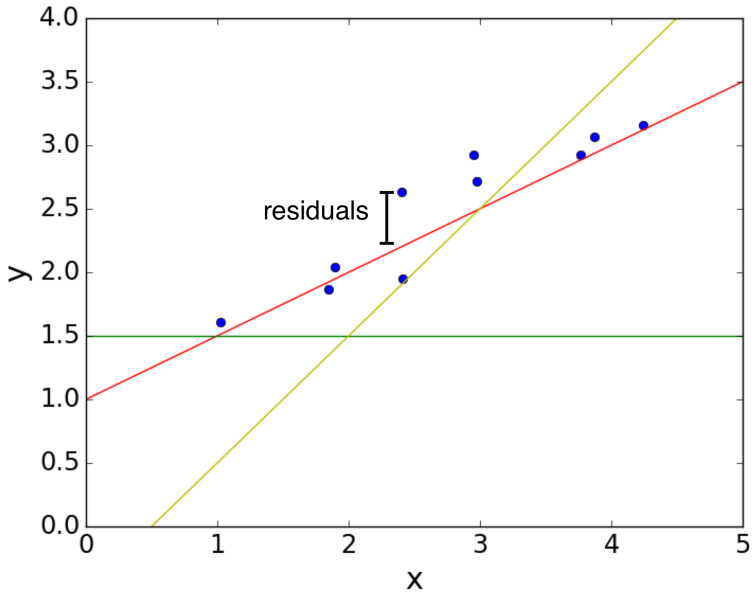
- Loss function: squared error (says how bad the fit is)

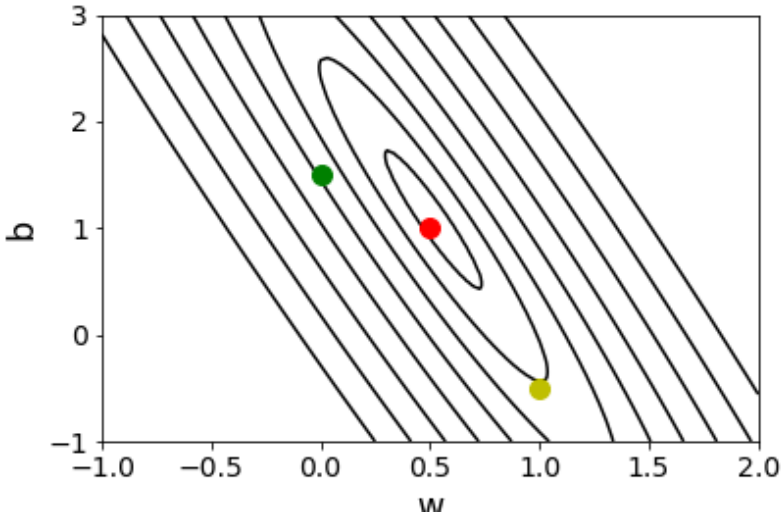
$$L(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- Average loss function (sometimes called the cost):

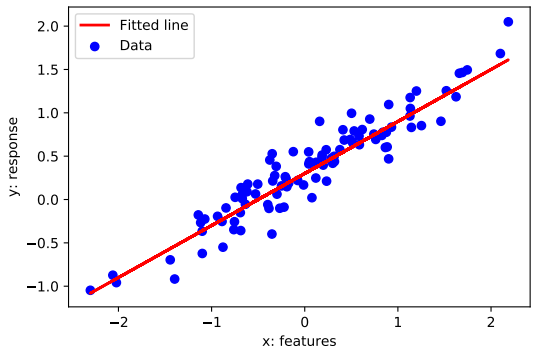
$$\begin{aligned}\hat{\mathcal{R}}(w, b) &= \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right)^2 \\ &= \frac{1}{2N} \sum_{i=1}^{N} \left(wx^{(i)} + b - t^{(i)} \right)^2\end{aligned}$$

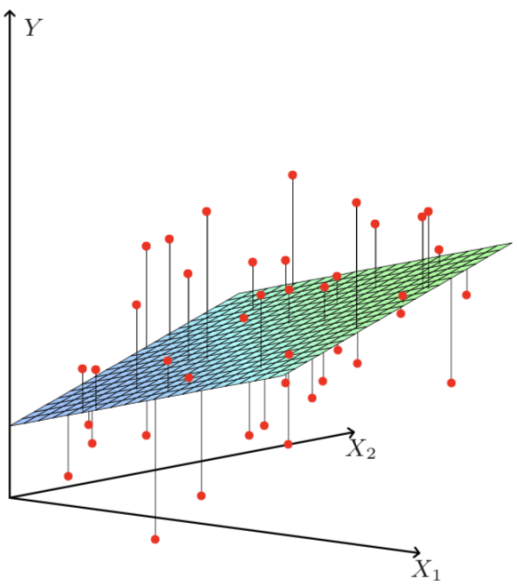
Problem Setup





What is linear? 1 feature vs D features





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

$$\hat{\mathcal{R}}(w, b) = \frac{1}{2N} \sum_{i=1}^{N} \left(wx^{(i)} + b - t^{(i)} \right)^2$$

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

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

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

Vectorization

- Notation-wise, $\frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)}-t^{(i)}\right)^2$ gets messy if we expand $y^{(i)}$:

$$\frac{1}{2N}\sum_{i=1}^{N}\left(\left(\sum_{j=1}^{D}w_j x_j^{(i)}+b\right)-t^{(i)}\right)^2$$

- The code equivalent is to compute the prediction 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:

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

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

- Sometimes we may use $\hat{\mathcal{R}} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$, 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 $\mathbf{y} = \mathbf{X}\mathbf{w}$.

Solving the Minimization Problem

We defined a cost function. This is what we'd like to minimize.

Two commonly applied mathematical approaches:

- Algebraic, e.g., using inequalities:
  - to show z^* minimizes $f(z)$, show that $\forall z, f(z) \geq f(z^*)$
  - to show that $a = b$, show that $a \geq b$ and $b \geq a$
- Calculus: 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).

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.
- We may also use optimization techniques that iteratively get us closer to the solution. We will get back to this soon.

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:

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

- Cost derivatives (average over data points):

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

Direct solution

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

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

- If $\partial \hat{\mathcal{R}} / \partial w_j \neq 0$, you could reduce the cost by changing w_j.
- This turns out to give a system of linear equations, which we can solve efficiently.
- Let's see what this looks like, assuming for simplicity that we set $b = 0$ (we can always at a dummy dimension to our data)

Direct solution

- We seek $\mathbf{w}$ to minimize $\hat{\mathcal{R}}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$
- Consider the vector of partial derivatives, or gradient:

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

- Setting this to 0 (**see course notes for additional details**) we get:

$$\frac{\partial \hat{\mathcal{R}}}{\partial \mathbf{w}} = \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{t} = \mathbf{0}$$

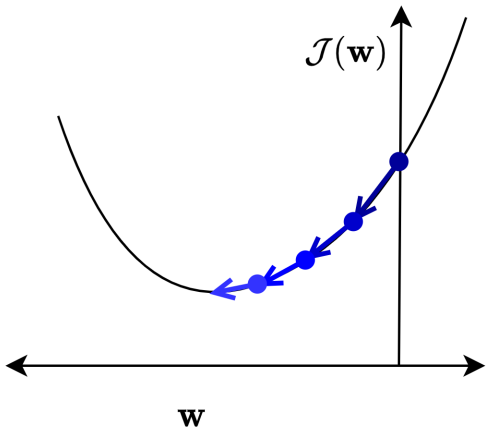
- From which we get the following optimal weights:

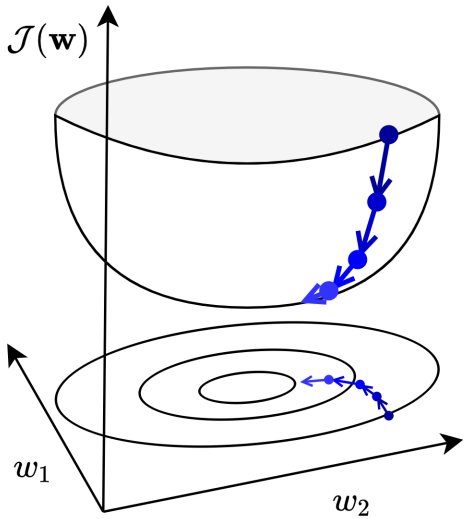
$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

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

Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: gradient descent.
- Many times, we do not have a direct solution: Taking derivatives of $\hat{\mathcal{R}}$ w.r.t $\mathbf{w}$ and setting them to 0 doesn't have an explicit solution.
- 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 $\partial\hat{\mathcal{R}}/\partial w_j > 0$, then increasing w_j increases $\hat{\mathcal{R}}$.
  - if $\partial\hat{\mathcal{R}}/\partial w_j < 0$, then increasing w_j decreases $\hat{\mathcal{R}}$.
- The following update always decreases the cost function for small enough α (unless $\partial\hat{\mathcal{R}}/\partial w_j = 0$):

$$w_j \leftarrow w_j - \alpha \frac{\partial \hat{\mathcal{R}}}{\partial w_j}$$

- $\alpha > 0$ 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

- Update rule in vector form:

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

- We know from calculus that the directional derivative of $\hat{\mathcal{R}}$ at $\mathbf{w}$ in the direction of $\boldsymbol{v}$ is

$$-\left\| \frac{\partial \hat{\mathcal{R}}}{\partial \mathbf{w}} \right\| \|\boldsymbol{v}\| \leq \frac{\partial \hat{\mathcal{R}}}{\partial \mathbf{w}}^{\top} \boldsymbol{v} \leq \left\| \frac{\partial \hat{\mathcal{R}}}{\partial \mathbf{w}} \right\| \|\boldsymbol{v}\|$$

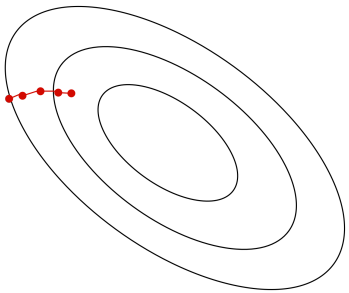
- So, if we consider unit vectors $\boldsymbol{v}$, the direction of greatest increase in $\hat{\mathcal{R}}$ at $\mathbf{w}$ is in the direction of the gradient. So, gradient descent updates the weights in the direction of fastest *decrease*.

Gradient Descent for Linear Regression

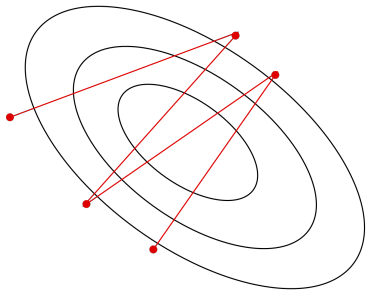
- The squared error loss of linear regression is a convex function.
- Even for linear regression, where there is a direct solution, we sometimes need to use GD.
- Why gradient descent, if we can find the optimum directly?
  - GD can be applied to a much broader set of models
  - GD can be easier to implement 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 GD (SGD, in a few slides)
    - Huge difference if $D \gg 1$

Learning Rate (Step Size)

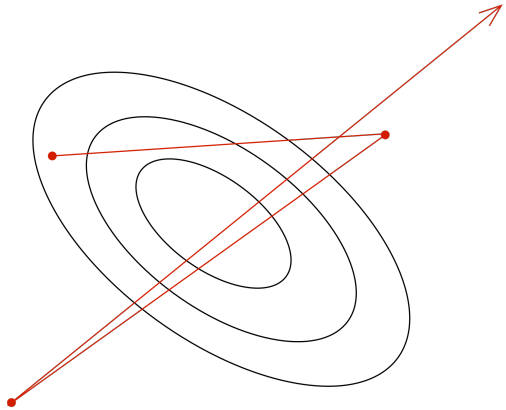
- In gradient descent, the learning rate α is a hyperparameter we need to tune. Here are some things that can go wrong:



α too small:
slow progress



α too large:
oscillations

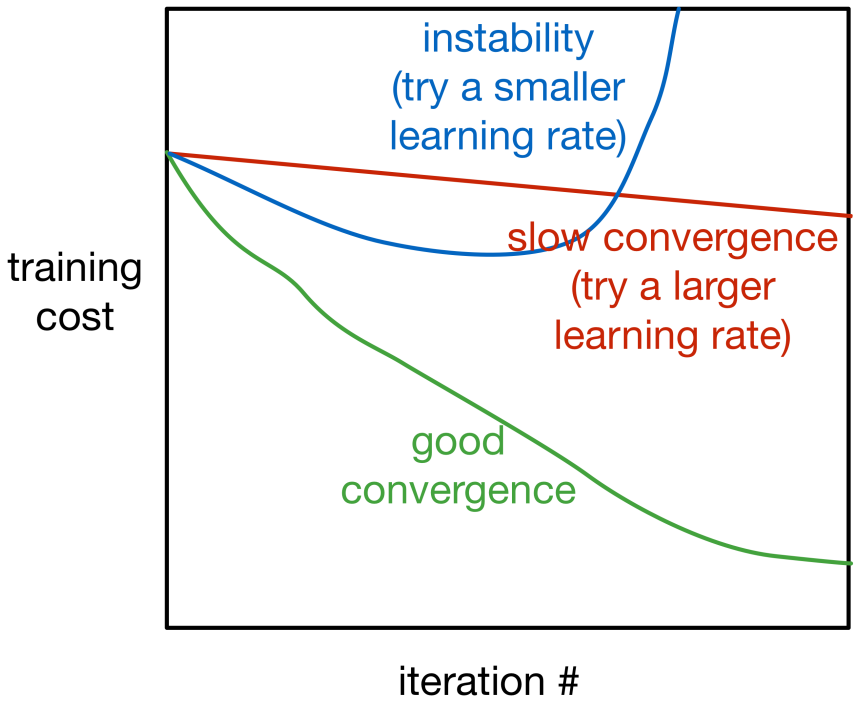


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.



- Warning: in general, it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.

Gradient descent

Visualization:

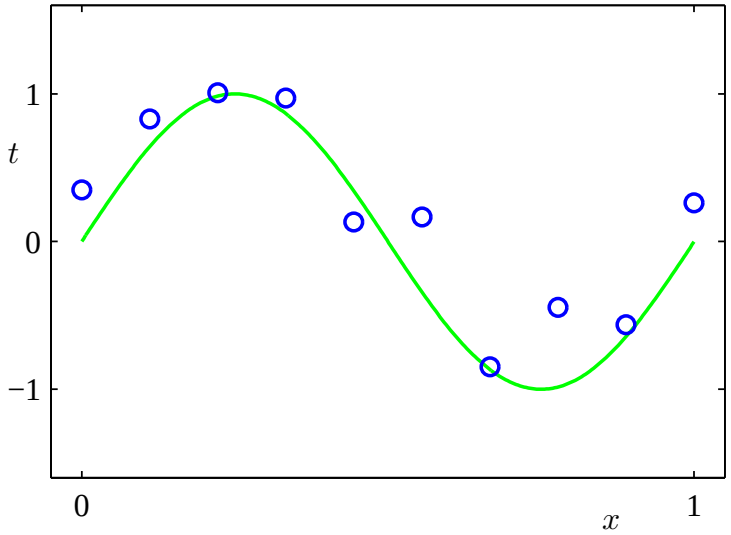
http://www.cs.toronto.edu/~guerzhoy/321/lec/W01/linear_regression.pdf#page=21

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).

Feature mappings

- Suppose we want to model the following data



-Pattern Recognition and Machine Learning, Christopher Bishop.

- One option: fit a low-degree polynomial; this is known as polynomial regression

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

- Do we need to derive a whole new algorithm?

Feature mappings

- We get polynomial regression for free!
- Define the feature map

$$\boldsymbol{\psi}(x) = \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

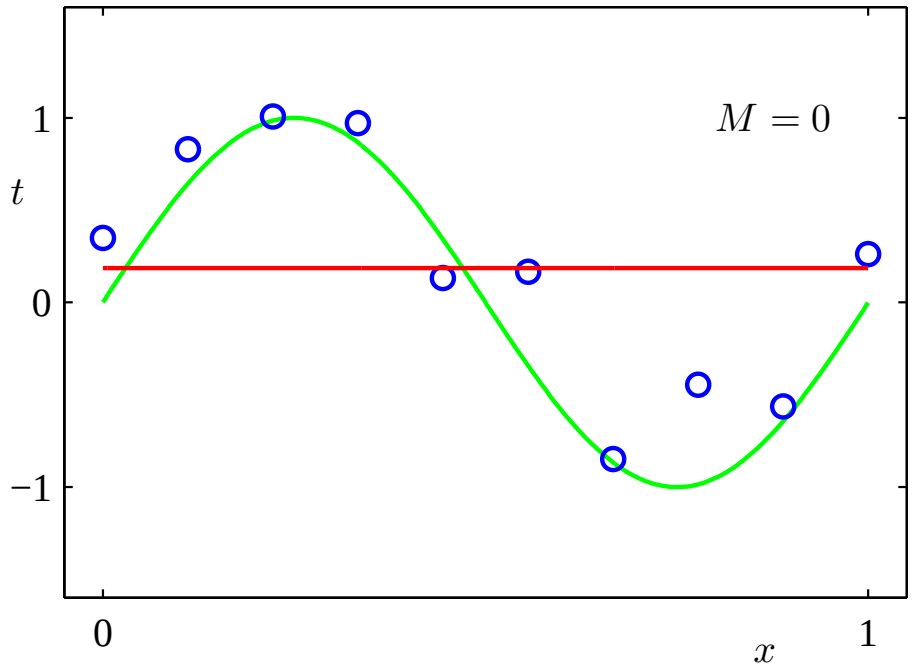
- Polynomial regression model:

$$y = \mathbf{w}^\top \boldsymbol{\psi}(x)$$

- All of the derivations and algorithms so far in this lecture remain exactly the same!

Fitting polynomials

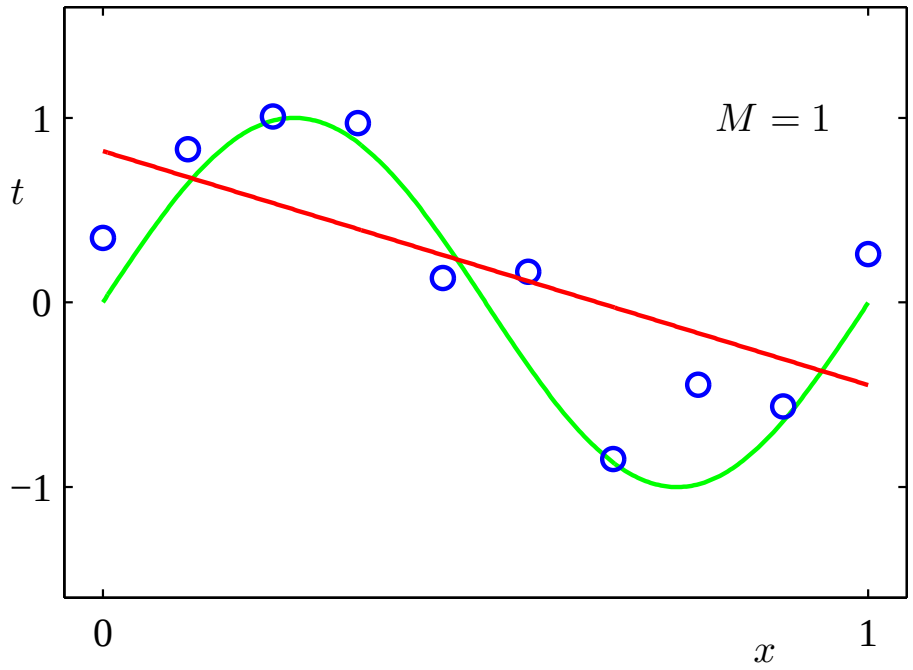
$y = w_0$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

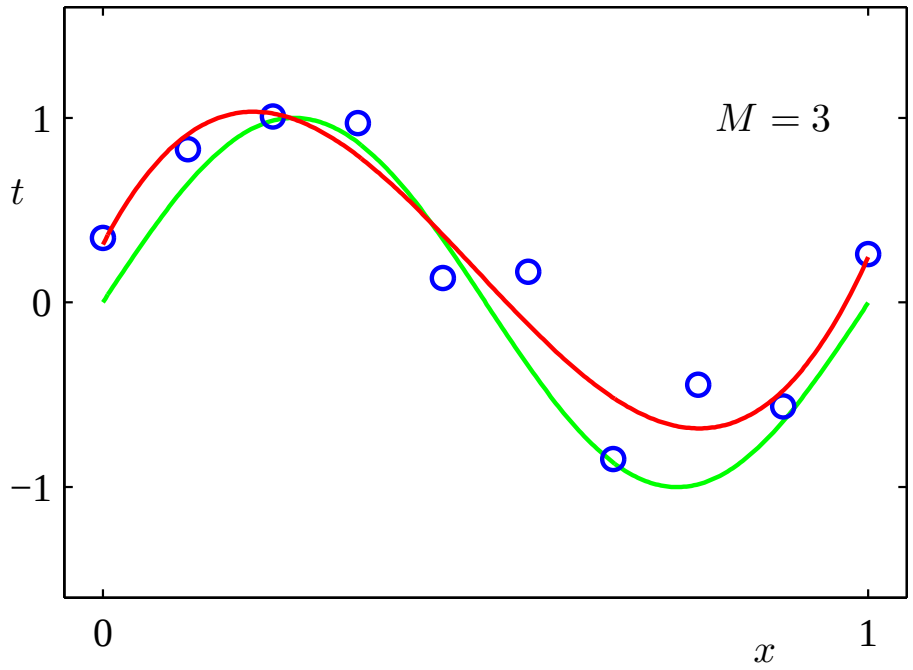
$$y = w_0 + w_1 x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

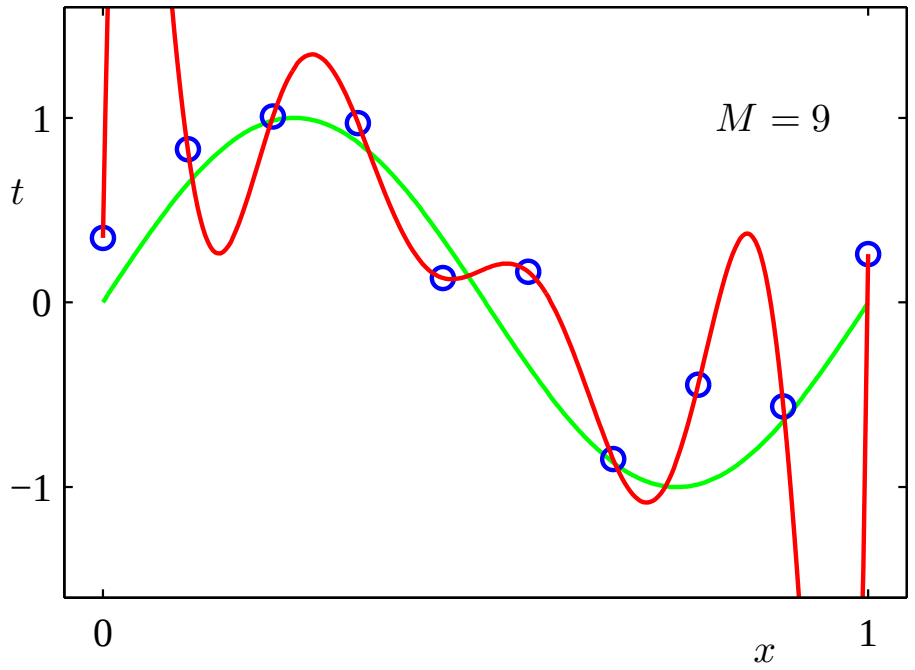
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

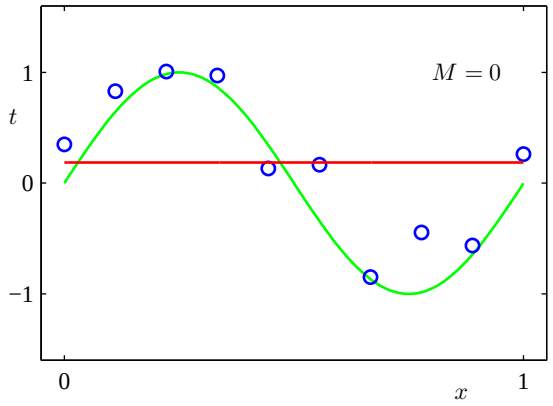
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$



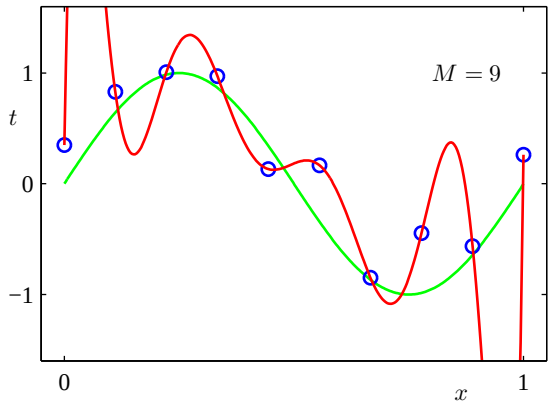
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

Underfitting : model is too simple — does not fit the data.

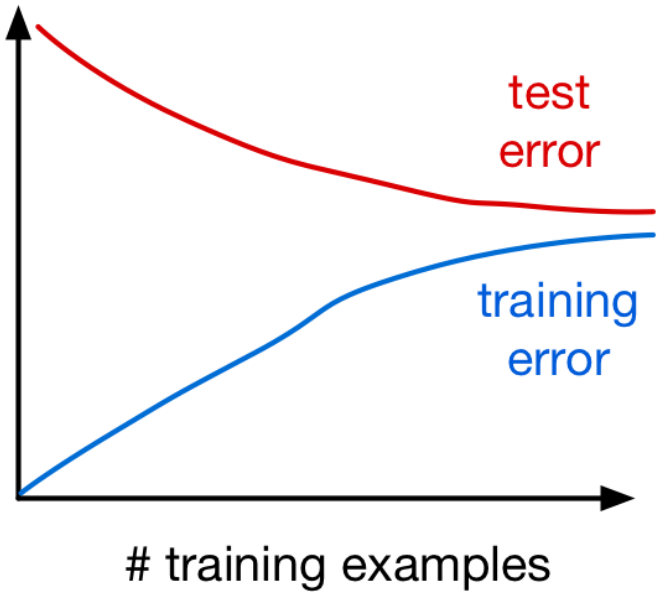


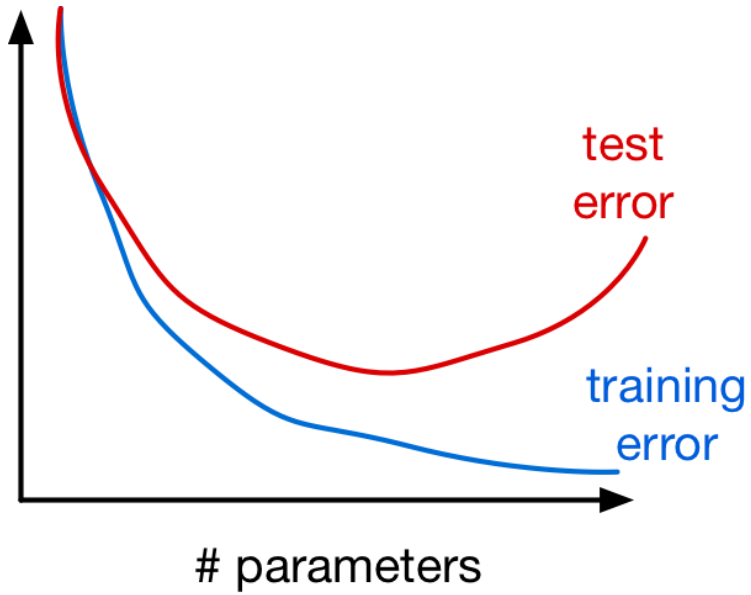
Overfitting : model is too complex — fits perfectly, does not generalize.



Generalization

- Training and test error as a function of # training examples and # parameters:



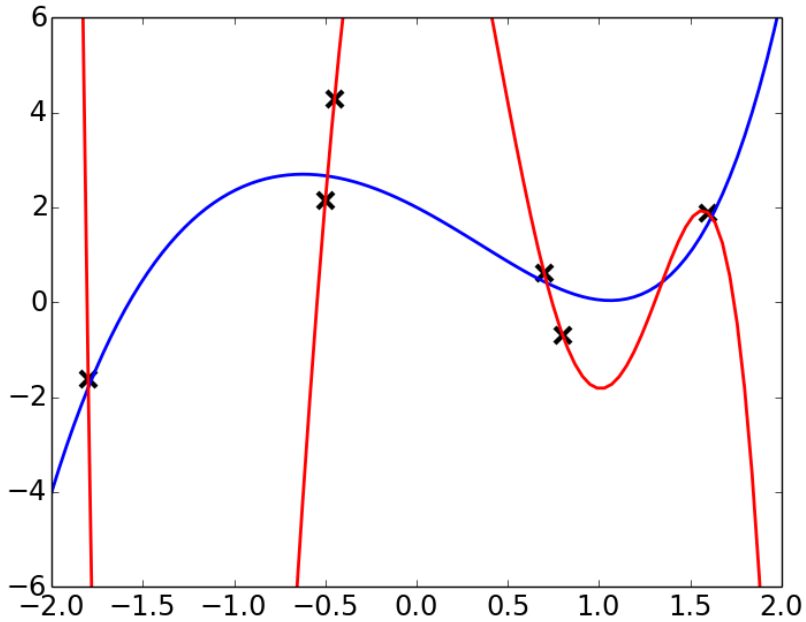


Regularization

- The degree of the polynomial 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 Regularization

Observation: polynomials that overfit often have large coefficients.



$$y = 0.1x^5 + 0.2x^4 + 0.75x^3 - x^2 - 2x + 2$$

$$y = -7.2x^5 + 10.4x^4 + 24.5x^3 - 37.9x^2 - 3.6x + 12$$

So let's try to keep the coefficients small.

L^2 Regularization

Another reason we want weights to be small:

- Suppose inputs x_1 and x_2 are nearly identical for all training examples. The following two hypotheses make nearly the same predictions:

$$\mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} -9 \\ 11 \end{pmatrix}$$

- But the second network might make weird predictions if the test distribution is slightly different (e.g. x_1 and x_2 match less closely).

L^2 (or ℓ_2) Regularization

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

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

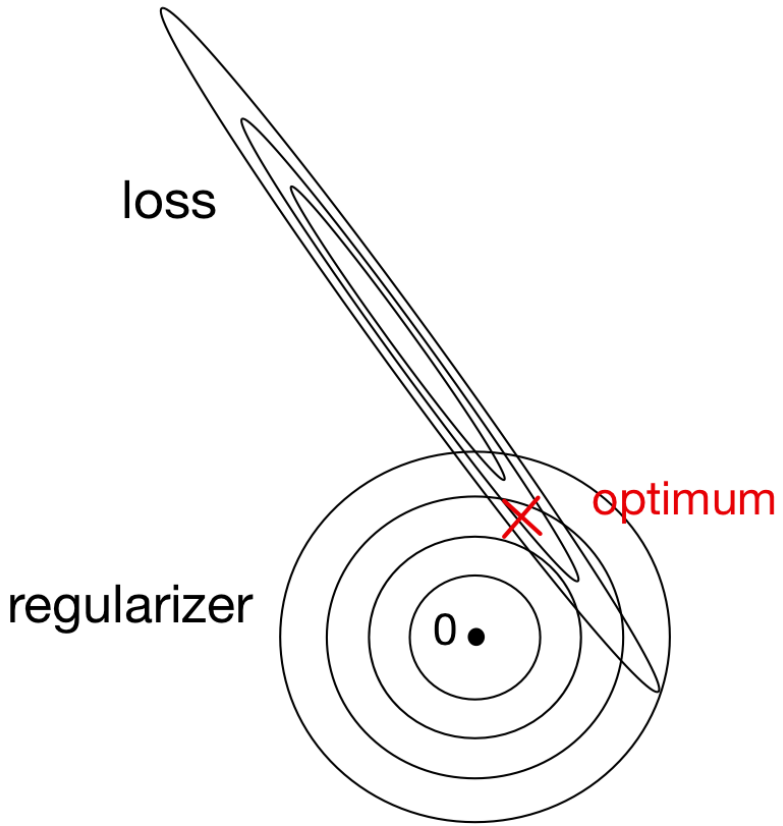
  - Note: To be precise, the L^2 norm is Euclidean distance, so we're regularizing the *squared* L^2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

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

- If you fit training data poorly, $\hat{\mathcal{R}}$ is large. If your optimal weights have high values, ϕ is large.
- Large λ penalizes weight values more.
- Like M, 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 $\hat{\mathcal{R}}(\mathbf{w}) = \frac{1}{2N}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$.

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

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

- 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 $\hat{\mathcal{R}}$:

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

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

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

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 (next topic)
- 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