CS184A/284A

AI in Biology and Medicine

Linear Regression
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression
Supervised learning

Notation
- Features $x$
- Targets $y$
- Predictions $\hat{y} = f(x; \theta)$
- Parameters $\theta$

Program ("Learner")
Characterized by some "parameters" $\theta$
Procedure (using $\theta$) that outputs a prediction

Learning algorithm
Change $\theta$
Improve performance

Training data (examples)
Features

Feedback / Target values

"predict"
Score performance ("cost function")

"train"
Linear regression

- Define form of function $f(x)$ explicitly
- Find a good $f(x)$ within that family

"Predictor": Evaluate line:

$$r = \theta_0 + \theta_1 x_1$$

return $r$
Define \( x_0 = 1 \) (constant)

Then

\[
\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots
\]

\[
\hat{y}(x) = \theta x^T
\]

\[
\theta = [\theta_0, \ldots, \theta_n]
\]

\[
x = [1, x_1, \ldots, x_n]
\]
Supervised learning

- Notation
  - Features $x$
  - Targets $y$
  - Predictions $\hat{y} = f(x; \theta)$
  - Parameters $\theta$

Program ("Learner")
Characterized by some "parameters" $\theta$
Procedure (using $\theta$) that outputs a prediction

Learning algorithm
Change $\theta$
Improve performance

Training data (examples)
Feedback / Target values

"predict"
Score performance ("cost function")

"train"
Measuring error

Prediction $\hat{y}$

Observation $y$

Error or “residual”

$y - \hat{y}(x) = (y - \theta \cdot x^T)$
Mean squared error

- How can we quantify the error?

\[
\text{MSE}, \ J(\theta) = \frac{1}{m} \sum_j (y^{(j)} - \hat{y}(x^{(j)}))^2
\]

\[
= \frac{1}{m} \sum_j (y^{(j)} - \theta \cdot x^{(j)T})^2
\]

- Could choose something else, of course…
  - Computationally convenient (more later)
  - Measures the variance of the residuals
  - Corresponds to likelihood under Gaussian model of “noise”

\[
\mathcal{N}(y ; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left\{ -\frac{1}{2\sigma^2} (y - \mu)^2 \right\}
\]
MSE cost function

- Rewrite using matrix form

\[
\text{MSE, } J(\theta) = \frac{1}{m} \sum_j (y^{(j)} - \hat{y}(x^{(j)}))^2
\]

\[
= \frac{1}{m} \sum_j (y^{(j)} - \theta \cdot x^{(j)T})^2
\]

\[
\theta = [\theta_0, \ldots, \theta_n]
\]

\[
y = [y^{(1)}, \ldots, y^{(m)}]^T
\]

\[
x^{(1)} \ldots x^{(1)}
\]

\[
x_0 \ldots x_n
\]

\[
x^{(m)} \ldots x^{(m)}
\]

\[
J(\theta) = \frac{1}{m} (y^T - \theta X^T) \cdot (y^T - \theta X^T)^T
\]

# Python / NumPy:

e = Y - X.dot( theta.T );
J = e.T.dot( e ) / m  # = np.mean( e ** 2 )
Supervised learning

- **Notation**
  - Features $x$
  - Targets $y$
  - Predictions $\hat{y} = f(x; \theta)$
  - Parameters $\theta$

Program ("Learner")
Characterized by some "parameters" $\theta$
Procedure (using $\theta$) that outputs a prediction

Learning algorithm
Change $\theta$
Improve performance

Training data (examples)
- Features
- Feedback / Target values

"predict"
Score performance ("cost function")

"train"
Visualizing the cost function

\[ J(\theta) \]

\[ \theta_0 \quad \theta_1 \]
Finding good parameters

- Want to find parameters which minimize our error...

- Think of a cost "surface": error residual for that $\theta$ ...

\[
\hat{\theta} = \arg \min_{\theta} J(\theta)
\]
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression
Gradient descent

- How to change $\theta$ to improve $J(\theta)$?
- Choose a direction in which $J(\theta)$ is decreasing
Gradient descent

- How to change $\theta$ to improve $J(\theta)$?
- Choose a direction in which $J(\theta)$ is decreasing
- Derivative $\frac{\partial J(\theta)}{\partial \theta}$
  - Positive $\Rightarrow$ increasing
  - Negative $\Rightarrow$ decreasing
Gradient descent in more dimensions

- Gradient vector

\[ \nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} & \frac{\partial J(\theta)}{\partial \theta_1} & \cdots \end{bmatrix} \]

Indicates direction of steepest ascent (negative = steepest descent)
Gradient descent

- Initialization
- Step size $\alpha$
  - Can change over iterations
- Gradient direction
- Stopping condition

Initialize $\theta$

Do{
  $\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$
}

while ($\alpha \| \nabla_\theta J \| > \varepsilon$)
Gradient for the MSE

- **MSE**

\[
J(\theta) = \frac{1}{m} \sum_j (y^{(j)} - \theta \cdot x^{(j)T})^2
\]

- \( \nabla J = ? \)

\[
J(\theta) = \frac{1}{m} \sum_j (y^{(j)} - \theta_0 x_0^{(j)} - \theta_1 x_1^{(j)} - \ldots)^2
\]

\[
\frac{\partial J}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} \frac{1}{m} \sum_j (e_j(\theta))^2
\]

\[
= \frac{1}{m} \sum_j \frac{\partial}{\partial \theta_0} (e_j(\theta))^2
\]

\[
= \frac{1}{m} \sum_j 2e_j(\theta) \frac{\partial}{\partial \theta_0} e_j(\theta)
\]

\[
\frac{\partial e_j(\theta)}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} y^{(j)} - \frac{\partial}{\partial \theta_0} \theta_0 x_0^{(j)} - \frac{\partial}{\partial \theta_0} \theta_1 x_1^{(j)} - \ldots
\]

\[
= 0
\]

\[
= -x_0^{(j)}
\]
Gradient for the MSE

- **MSE**

\[ J(\theta) = \frac{1}{m} \sum_j (y(j) - \theta \cdot x(j)^T)^2 \]

- **\( \nabla J = ? \)**

\[ J(\theta) = \frac{1}{m} \sum_j (y(j) - \theta_0 x_0^{(j)} - \theta_1 x_1^{(j)} - \ldots)^2 \]

\[
\nabla J(\theta) = \begin{bmatrix}
\frac{\partial J}{\partial \theta_0} & \frac{\partial J}{\partial \theta_1} & \ldots \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
\frac{2}{m} \sum_j -e_j(\theta)x_0^{(j)} & \frac{2}{m} \sum_j -e_j(\theta)x_1^{(j)} & \ldots \\
\end{bmatrix}
\]
Gradient descent

- Initialization
- Step size $\alpha$
  - Can change over iterations
- Gradient direction
- Stopping condition

Initialize $\theta$

Do{
  $\theta \leftarrow \theta - \alpha \nabla_{\theta} J(\theta)$
}

while $(\alpha \| \nabla_{\theta} J \| > \varepsilon)$

$$J(\theta) = \frac{1}{m} \sum_{j} (y^{(j)} - \theta \cdot x^{(j)T})^2$$

$$\nabla J(\theta) = -\frac{2}{m} \sum_{j} (y^{(j)} - \theta \cdot x^{(j)T}) \cdot [x_0^{(j)} x_1^{(j)} \ldots]$$

Error magnitude & direction for datum $j$

Sensitivity to each param
Derivative of MSE

- Rewrite using matrix form

\[
\nabla J(\theta) = -\frac{2}{m} \sum_j (y^{(j)} - \theta \cdot x^{(j)T}) \cdot \begin{bmatrix} x_0^{(j)} & x_1^{(j)} & \ldots \end{bmatrix}
\]

\[
\theta = [\theta_0, \ldots, \theta_n]
\]

\[
y = \begin{bmatrix} y^{(1)} \ldots, y^{(m)} \end{bmatrix}^T
\]

\[
\nabla J(\theta) = -\frac{2}{m} (y^T - \theta X^T) \cdot X
\]

```python
# error residual
e = Y - X.dot(theta.T)
# compute the gradient
DJ = - e.dot(X) * 2.0/m
# take a step
theta -= alpha * DJ
```
Gradient descent on cost function
Comments on gradient descent

- Very general algorithm
  - We’ll see it many times
- Local minima
  - Sensitive to starting point
Comments on gradient descent

• Very general algorithm
  – We’ll see it many times

• Local minima
  – Sensitive to starting point

• Step size
  – Too large? Too small? Automatic ways to choose?
  – May want step size to decrease with iteration
  – Common choices:
    • Fixed
    • Linear: $C/(\text{iteration})$
    • Line search / backoff (Armijo, etc.)
    • Newton’s method
Newton’s method

- Want to find the roots of $f(x)$
  - “Root”: value of $x$ for which $f(x)=0$
- Initialize to some point $x$
- Compute the tangent at $x$ & compute where it crosses $x$-axis
  \[
  \nabla f(z) = \frac{0 - f(z)}{z' - z} \quad \Rightarrow \quad z' = z - \frac{f(z)}{\nabla f(z)}
  \]
- Optimization: find roots of $rJ(\mu)$
  \[
  \nabla^2 J(\theta) = \frac{0 - \nabla J(\theta)}{\theta' - \theta} \quad \Rightarrow \quad \theta' = \theta - \frac{\nabla J(\theta)}{\nabla^2 J(\theta)}
  \]
  - If converges, usually very fast
  - Works well for smooth, non-pathological functions, locally quadratic
  - For $n$ large, may be computationally hard: $O(n^2)$ storage, $O(n^3)$ time

(Multivariate:
  \[ r J(\mu) = \text{gradient vector} \]
  \[ r^2 J(\mu) = \text{matrix of 2^{nd} derivatives} \]
  \[ a/b = a^{-1}, \text{matrix inverse} \]
Stochastic / Online gradient descent

- **MSE**

\[
J(\theta) = \frac{1}{m} \sum_{j} J_j(\theta), \quad J_j(\theta) = (y^{(j)} - \theta \cdot x^{(j)T})^2
\]

- **Gradient**

\[
\nabla J(\theta) = \frac{1}{m} \sum_{j} \nabla J_j(\theta) \quad \nabla J_j(\theta) = (y^{(j)} - \theta \cdot x^{(j)T}) \cdot [x_0^{(j)} x_1^{(j)} \ldots]
\]

- **Stochastic (or “online”) gradient descent:**
  - Use updates based on individual datum j, chosen at random
  - At optima, \( \mathbb{E} [\nabla J_j(\theta)] = \nabla J(\theta) = 0 \) (average over the data)
Online gradient descent

- Update based on one datum, and its residual, at a time

Initialize $\theta$

Do {
  for $j=1:m$
    $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_j(\theta)$
} while (not done)
Online gradient descent

Initialize $\theta$
Do {
  for $j=1:m$
    $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_j(\theta)$
} while (not done)
Online gradient descent

Initialize $\theta$
Do {
    for $j=1:m$
        $\theta \leftarrow \theta - \alpha \nabla_{\theta}J_j(\theta)$
} while (not done)
Online gradient descent

Initialize $\theta$
Do {
  for $j=1:m$
    $\theta \leftarrow \theta - \alpha \nabla \theta_j(\theta)$
} while (not done)
Online gradient descent

Initialize $\theta$

Do {
  for $j=1:m$
    $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_j(\theta)$
} while (not done)
Online gradient descent

Initialize $\theta$
Do {
    for $j=1:m$
        $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_j(\theta)$
    } while (not done)
Online gradient descent

• Benefits
  – Lots of data = many more updates per pass
  – Computationally faster

• Disadvantages
  – No longer strictly “descent”
  – Stopping conditions may be harder to evaluate
  (Can use “running estimates” of J(.), etc. )

Initialize $\theta$

Do {
  for $j=1:m$
    $\theta \leftarrow \theta - \alpha \nabla_{\theta} J_j(\theta)$
} while (not done)

\[
J_j(\theta) = (y^{(j)} - \theta \cdot x^{(j)T})^2
\]

\[
\nabla J_j(\theta) = -2(y^{(j)} - \theta \cdot x^{(j)T}) \cdot [x_0^{(j)} x_1^{(j)} \ldots]
\]
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression
MSE Minimum

- Consider a simple problem
  - One feature, two data points
  - Two unknowns: $\theta_0$, $\theta_1$
  - Two equations:
    \[
    y^{(1)} = \theta_0 + \theta_1 x^{(1)} \\
    y^{(2)} = \theta_0 + \theta_1 x^{(2)}
    \]
- Can solve this system directly:
  \[
  \begin{align*}
  y^T &= \theta X^T \\
  \hat{\theta} &= y^T (X^T)^{-1}
  \end{align*}
  \]
- However, most of the time, $m > n$
  - There may be no linear function that hits all the data exactly
  - Instead, solve directly for minimum of MSE function
MSE Minimum

• Simplify with some algebra:

\[ \nabla J(\theta) = -\frac{2}{m} (y^T - \theta X^T) \cdot X = 0 \]

\[ y^T X - \theta X^T \cdot X = 0 \]
\[ y^T X = \theta X^T \cdot X \]
\[ \theta = y^T X (X^T X)^{-1} \]

• \( X (X^T X)^{-1} \) is called the “pseudo-inverse”

• If \( X^T \) is square and full rank, this is the inverse
• If \( m > n \): overdetermined; gives minimum MSE fit
Matlab MSE

- This is easy to solve in Matlab...

\[ \theta = y^T X (X^T X)^{-1} \]

\%
\%
\%
\%
\%
\%
\%
\%

% Solution 1: “manual”
\[ \text{th} = y' \times X \times \text{inv}(X' \times X); \]

% Solution 2: “mrdivide”
\[ \text{th} = y' / X' ; \% \text{th} \times X' = y \Rightarrow \text{th} = y / X' \]
Python MSE

• This is easy to solve in Python / NumPy…

\[ \theta = y^T X (X^T X)^{-1} \]

# y = np.matrix( [[y1], ... , [ym]] )
# X = np.matrix( [[x1_0 ... x1_n], [x2_0 ... x2_n], ...
# ...
#
# Solution 1: “manual”
# th = y.T * X * np.linalg.inv(X.T * X)

# Solution 2: “least squares solve”
# th = np.linalg.lstsq(X, Y)
Normal equations

\[ \nabla J(\theta) = 0 \implies (y^T - \theta X^T) \cdot X = 0 \]

- **Interpretation:**
  - \((y - \theta X) = (y - \hat{y})\) is the vector of errors in each example
  - \(X\) are the features we have to work with for each example
  - Dot product = 0: orthogonal

\[
\begin{align*}
\underline{y}^T &= [y^{(1)} \ldots y^{(m)}] \\
\underline{x}_i &= [x_i^{(1)} \ldots x_i^{(m)}]
\end{align*}
\]
Normal equations

\[ \nabla J(\theta) = 0 \implies (y^T - \theta X^T) \cdot X = 0 \]

• Interpretation:
  – \((y - \theta X) = (y - \hat{y})\) is the vector of errors in each example
  – \(X\) are the features we have to work with for each example
  – Dot product = 0: orthogonal

• Example:

\[
\begin{align*}
y &= [1 \ 3 \ 3]^T \\
x_0 &= [1 \ 1 \ 1]^T \\
x_1 &= [1 \ 2 \ 4]^T \\
\theta &= [1.00 \ 0.57] \\
\epsilon &= (y - \hat{y}) = [-0.57 \ 0.85 \ -0.28]^T
\end{align*}
\]
Effects of MSE choice

• Sensitivity to outliers

$16^2$ cost for this one datum

Heavy penalty for large errors
L1 error: Mean Absolute Error

\[ \ell_1(\theta) = \sum_j |y^{(j)} - \hat{y}(x^{(j)})| \]

\[ = \sum_j |y - \theta \cdot x^T| \]
Cost functions for regression

\[ \ell_2 : (y - \hat{y})^2 \quad \text{(MSE)} \]

\[ \ell_1 : |y - \hat{y}| \quad \text{(MAE)} \]

Something else entirely…

\[ c - \log(\exp(-(y - \hat{y})^2) + c) \quad \text{(???)} \]

Arbitrary functions cannot be solved in closed form - use gradient descent
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression
More dimensions?

\[ \hat{y}(x) = \theta \cdot x^T \]

\[ \theta = [\theta_0 \ \theta_1 \ \theta_2] \]

\[ x = [1 \ x_1 \ x_2] \]
Nonlinear functions

- What if our hypotheses are not lines?
  - Ex: higher-order polynomials
Nonlinear functions

• Single feature $x$, predict target $y$:

$$ D = \{(x^{(j)}, y^{(j)})\} $$

$$ \hat{y}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 $$

Add features:

$$ D = \{([x^{(j)}, (x^{(j)})^2, (x^{(j)})^3], y^{(j)})\} $$

$$ \hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 $$

Linear regression in new features

• Sometimes useful to think of “feature transform”

$$ \Phi(x) = [1, x, x^2, x^3, \ldots] $$

$$ \hat{y}(x) = \theta \cdot \Phi(x) $$
Higher-order polynomials

- Fit in the same way
- More “features”
Features

- In general, can use any features we think are useful

- Other information about the problem
  - Anything you can encode as fixed-length vectors of numbers

- Polynomial functions
  - Features $[1, x, x^2, x^3, \ldots]$

- Other functions
  - $1/x$, $\sqrt{x}$, $x_1 \times x_2$, …

- “Linear regression” = linear in the parameters
  - Features we can make as complex as we want!
Higher-order polynomials

- Are more features better?

- “Nested” hypotheses
  - $2^{\text{nd}}$ order more general than $1^{\text{st}}$,
  - $3^{\text{rd}}$ order more general than $2^{\text{nd}}$, …

- Fits the observed data better
Overfitting and complexity

- More complex models will always fit the training data better
- But they may “overfit” the training data, learning complex relationships that are not really present
Test data

- After training the model
- Go out and get more data from the world
  - New observations (x,y)
- How well does our model perform?
Training versus test error

- Plot MSE as a function of model complexity
  - Polynomial order
- Decreases
  - More complex function fits training data better
- What about new data?
  - 0\textsuperscript{th} to 1\textsuperscript{st} order
    - Error decreases
    - Underfitting
  - Higher order
    - Error increases
    - Overfitting

Training data

New, “test” data

Mean squared error

Polynomial order
Linear Regression via Least Squares

Gradient Descent Algorithms

Direct Minimization of Squared Error

Regression with Non-linear Features

Bias, Variance, & Validation

Regularized Linear Regression
Inductive bias

- The assumptions needed to predict examples we haven’t seen
- Makes us “prefer” one model over another
- Polynomial functions; smooth functions; etc

- Some bias is necessary for learning!
Bias & variance

Data we observe

\[ y(x) = \theta_0 + \theta_1 x + \nu \]

"The world"

\[ \hat{y}(x) = \hat{\theta}_0 + \hat{\theta}_1 x \]

Three different possible data sets:
Bias & variance

Three different possible data sets:

Each would give different predictors for any polynomial degree:
Detecting overfitting

- Overfitting effect
  - Do better on training data than on future data
  - Need to choose the “right” complexity

- One solution: “Hold-out” data
- Separate our data into two sets
  - Training
  - Test
- Learn only on training data
- Use test data to estimate generalization quality
  - Model selection

- All good competitions use this formulation
  - Often multiple splits: one by judges, then another by you
Model selection

• Which of these models fits the data best?
  – $p=0$ (constant); $p=1$ (linear); $p=3$ (cubic); …

• Or, should we use KNN? Other methods?

• Model selection problem
  – Can’t use training data to decide (esp. if models are nested!)

• Want to estimate
  \[
  \mathbb{E}_{(x,y)}[J(y, \hat{y}(x; D))]
  \]

  $J =$ loss function (MSE)
  $D =$ training data set
Hold-out method

• Validation data
  – “Hold out” some data for evaluation (e.g., 70/30 split)
  – Train only on the remainder

• Some problems, if we have few data:
  – Few data in hold-out: noisy estimate of the error
  – More hold-out data leaves less for training!
Cross-validation method

• K-fold cross-validation
  – Divide data into K disjoint sets
  – Hold out one set (= \( M / K \) data) for evaluation
  – Train on the others (= \( M^*(K-1) / K \) data)

<table>
<thead>
<tr>
<th>( x^{(i)} )</th>
<th>( y^{(i)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>79</td>
</tr>
<tr>
<td>32</td>
<td>-2</td>
</tr>
<tr>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>68</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>-16</td>
</tr>
<tr>
<td>20</td>
<td>43</td>
</tr>
<tr>
<td>53</td>
<td>77</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>87</td>
<td>94</td>
</tr>
</tbody>
</table>

- Split 1:
  MSE = 331.8

- Split 2:
  MSE = 361.2

- Split 3:
  MSE = 669.8

3-Fold X-Val MSE = 464.1
Cross-validation method

- **K-fold cross-validation**
  - Divide data into K disjoint sets
  - Hold out one set (= M / K data) for evaluation
  - Train on the others (= M*(K-1) / K data)

![Graph showing cross-validation results](image)

<table>
<thead>
<tr>
<th>x(i)</th>
<th>y(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>79</td>
</tr>
<tr>
<td>32</td>
<td>-2</td>
</tr>
<tr>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>68</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>-16</td>
</tr>
<tr>
<td>20</td>
<td>43</td>
</tr>
<tr>
<td>53</td>
<td>77</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
</tr>
<tr>
<td>87</td>
<td>94</td>
</tr>
</tbody>
</table>

Split 1: MSE = 280.5
Split 2: MSE = 3081.3
Split 3: MSE = 1640.1
3-Fold X-Val MSE = 1667.3
Cross-validation

• Advantages:
  – Lets us use more (M) validation data
    (= less noisy estimate of test performance)

• Disadvantages:
  – More work
    • Trains K models instead of just one
  – Doesn’t evaluate any *particular* predictor
    • Evaluates K different models & averages
    • Scores *hyperparameters / procedure*, not an actual, specific predictor!

• Also: still estimating error for M’ < M data…
Learning curves

• Plot performance as a function of training size
  – Assess impact of fewer data on performance
    Ex: MSE0 - MSE (regression)
    or 1-Err (classification)

• Few data
  – More data significantly improve performance

• “Enough” data
  – Performance saturates

• If slope is high, decreasing $m$ (for validation / cross-validation) might have a big impact…
Leave-one-out cross-validation

- When $K=M$ (# of data), we get
  - Train on all data except one
  - Evaluate on the left-out data
  - Repeat $M$ times (each data point held out once) and average

\[
\text{LOO X-Val MSE} = \ldots
\]

<table>
<thead>
<tr>
<th>(x^{(i)})</th>
<th>(y^{(i)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>79</td>
</tr>
<tr>
<td>32</td>
<td>-2</td>
</tr>
<tr>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>68</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>-16</td>
</tr>
<tr>
<td>20</td>
<td>43</td>
</tr>
<tr>
<td>53</td>
<td>77</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>87</td>
<td>94</td>
</tr>
</tbody>
</table>
Cross-validation Issues

• Need to balance:
  – Computational burden (multiple trainings)
  – Accuracy of estimated performance / error

• Single hold-out set:
  – Estimates performance with $M' < M$ data (important? learning curve?)
  – Need enough data to trust performance estimate
  – Estimates performance of a particular, trained learner

• K-fold cross-validation
  – K times as much work, computationally
  – Better estimates, still of performance with $M' < M$ data

• Leave-one-out cross-validation
  – M times as much work, computationally
  – $M' = M-1$, but overall error estimate may have high variance
Machine Learning

- Linear Regression via Least Squares
- Gradient Descent Algorithms
- Direct Minimization of Squared Error
- Regression with Non-linear Features
- Bias, Variance, & Validation
- Regularized Linear Regression
What to do about under/overfitting?

- **Ways to increase complexity?**
  - Add features, parameters
  - We’ll see more…

- **Ways to decrease complexity?**
  - Remove features ("feature selection")
  - “Fail to fully memorize data”
    - Partial training
    - Regularization

![Graph showing ideal range for model complexity](image)
Linear regression

• Linear model, two data

• Quadratic model, two data?
  – Infinitely many settings with zero error
  – How to choose among them?

• Higher order coefficients = 0?
  – Uses knowledge of where features came from…

• Could choose e.g. minimum magnitude:
  \[ \min \theta \theta^T \quad s.t. \quad J(\theta) = 0 \]

• A type of bias: tells us which models to prefer
0th Order Polynomial

\[ N = 0 \]
$1^{\text{st}}$ Order Polynomial

\[ N=1 \]
$3^{rd}$ Order Polynomial

![Graph showing a 3rd order polynomial with N=3]
$9^{\text{th}}$ Order Polynomial
## Estimated Polynomial Coefficients

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N=0$</th>
<th>$N=1$</th>
<th>$N=3$</th>
<th>$N=9$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.19</td>
<td>0.82</td>
<td>0.31</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>-1.27</td>
<td>7.99</td>
<td>232.37</td>
<td>640042.26</td>
</tr>
<tr>
<td></td>
<td>-25.43</td>
<td>-5321.83</td>
<td>48568.31</td>
<td>125201.43</td>
</tr>
<tr>
<td></td>
<td>17.37</td>
<td>-231639.30</td>
<td>-557682.99</td>
<td>1061800.52</td>
</tr>
</tbody>
</table>

The graph shows the estimated regression coefficients $\theta$ for different values of $N$. The inset graph plots $E_{\text{RMS}}$ versus $N$ for both training and test cases.
Regularization

• Can modify our cost function $J$ to add “preference” for certain parameter values

$$J(\theta) = \frac{1}{2}(y - \theta X^T) \cdot (y - \theta X^T)^T + \alpha \theta \theta^T$$

• New solution (derive the same way)

$$\theta = y X (X^T X + \alpha I)^{-1}$$

– Problem is now well-posed for any degree

• Notes:
  – “Shrinks” the parameters toward zero
  – Alpha large: we prefer small theta to small MSE
  – Regularization term is independent of the data: paying more attention reduces our model variance

$L_2$ penalty: “Ridge regression”

$$\theta \theta^T = \sum_{i} \theta_i^2$$
Regression: Zero Regularization
Regression: Moderate Regularization
Regression: Big Regularization
Impact of Regularization Parameter

![Graph showing the impact of regularization parameter $\ln(\alpha)$ on $E_{\text{RMS}}$. The graph displays two curves: one for training and another for test data, indicating the RMS error as a function of $\ln(\alpha)$.](Image)
# Estimated Polynomial Coefficients

<table>
<thead>
<tr>
<th>Estimated Regression Coefficients $\theta$</th>
<th>$\alpha$ zero</th>
<th>$\alpha$ medium</th>
<th>$\alpha$ big</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.35</td>
<td>0.35</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>232.37</td>
<td>4.74</td>
<td>-0.05</td>
<td></td>
</tr>
<tr>
<td>-5321.83</td>
<td>-0.77</td>
<td>-0.06</td>
<td></td>
</tr>
<tr>
<td>48568.31</td>
<td>-31.97</td>
<td>-0.05</td>
<td></td>
</tr>
<tr>
<td>-231639.30</td>
<td>-3.89</td>
<td>-0.03</td>
<td></td>
</tr>
<tr>
<td>640042.26</td>
<td>55.28</td>
<td>-0.02</td>
<td></td>
</tr>
<tr>
<td>-1061800.52</td>
<td>41.32</td>
<td>-0.01</td>
<td></td>
</tr>
<tr>
<td>1042400.18</td>
<td>-45.95</td>
<td>-0.00</td>
<td></td>
</tr>
<tr>
<td>-557682.99</td>
<td>-91.53</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>125201.43</td>
<td>72.68</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>
Regularization

- Compare between unreg. & reg. results

\[
\text{Alpha} = 0 \\
\text{(Unregularized)}
\]

\[
\text{Alpha} = 1
\]
Different regularization functions

- More generally, for the $L_p$ regularizer:

$$\left( \sum_i |\theta_i|^p \right)^{\frac{1}{p}}$$

Isosurfaces: $||\theta||_p = \text{constant}$

- $L_0 = \text{limit as } p \text{ goes to 0}: \text{“number of nonzero weights”, a natural notion of complexity}$
Different regularization functions

- More generally, for the $L_p$ regularizer:

$$
\left( \sum_i |\theta_i|^p \right)^{\frac{1}{p}}
$$

\begin{align*}
\text{\textcolor{red}{p=2}} & & \text{\textcolor{green}{p=1}} & & \text{\textcolor{blue}{p=0.3}} \\
\end{align*}
Regularization: $L_2$ vs $L_1$

- Estimate balances data term & regularization term

![Diagram showing $L_2$ and $L_1$ regularization with points minimizing data term and combination]
Regularization: $L_2$ vs $L_1$

- Estimate balances data term & regularization term
- Lasso tends to generate sparser solutions than a quadratic regularizer.
Gradient-Based Optimization

- \(L_2\) makes (all) coefficients smaller
- \(L_1\) makes (some) coefficients exactly zero: feature selection

**Objective Function:**

\[ f(\theta_i) = |\theta_i|^p \]

**Negative Gradient:**

\[ -f'(\theta_i) \]

*(Informal intuition: Gradient of \(L_1\) objective not defined at zero)*