Non-linear regression

High temperature / peak demand observations for all days in 2008-2011
Central idea of non-linear regression: same as linear regression, just with non-linear features

\[
\phi(x_i) = \begin{bmatrix}
    x_i^2 \\
    x_i \\
    1
\end{bmatrix}
\]

Two ways to construct non-linear features: explicitly (construct actual feature vector), or implicitly (using kernels)
A degree 2 polynomial is fitted to observed data, showing a curvilinear relationship between high temperature and peak hourly demand.
Observed Data
d = 3

Degree 3 polynomial
Degree 4 polynomial
Constructing explicit feature vectors

- Polynomial features (max degree $d$)

  Special case, $n=1$: $\phi(z) = \begin{bmatrix} z^d \\ z^{d-1} \\ \vdots \\ z \\ 1 \end{bmatrix} \in \mathbb{R}^{d+1}$

  General case: $\phi(z) = \left\{ \prod_{i=1}^{n} z_i^{b_i} : \sum_{i=1}^{n} b_i \leq d \right\} \in \mathbb{R}^{\binom{n+d}{d}}$
Plot of polynomial bases
- Radial basis function (RBF) features
  - Defined by bandwidth $\sigma$ and $k$ RBF centers $\mu_j \in \mathbb{R}^n$, $j = 1, \ldots, k$

$$
\phi_j(z) = \exp \left\{ -\frac{\|z - \mu_j\|^2}{2\sigma^2} \right\}
$$
Difficulties with non-linear features

- Problem #1: Computational difficulties
  - Polynomial features,
    \[ k = \binom{n + d}{d} = O(d^n) \]
  - RBF features; suppose we want centers in uniform grid over input space (w/ \(d\) centers along each dimension)
    \[ k = d^n \]
  - In both cases, exponential in the size of the input dimension; quickly intractable to even store in memory
Problem #2: Representational difficulties

- With many features, our prediction function becomes very expressive

- Can lead to overfitting (low error on input data points, but high error nearby)

- Let’s see an intuitive example
Least-squares fits for polynomial features of different degrees
Least-squares fits for different numbers of RBFs
A few ways to deal with representational problem:

- Choose less expressive function (e.g., lower degree polynomial, fewer RBF centers, larger RBF bandwidth)

- Regularization: penalize large parameters $\theta$

\[
\minimize_{\theta} \sum_{i=1}^{m} \ell(\hat{y}_i, y_i) + \lambda \|\theta\|_2^2
\]

$\lambda$: regularization parameter, trades off between low loss and small values of $\theta$ (often, don’t regularize constant term)
Pareto optimal surface for 20 RBF functions
RBF fits varying regularization parameter (not regularizing constant term)
- **Regularization:** penalize large parameters $\theta$

$$\min_{\theta} \sum_{i=1}^{m} \ell(\hat{y}_i, y_i) + \lambda \|\theta\|_2^2$$

$\lambda$: regularization parameter, trades off between low loss and small values of $\theta$ (often, don’t regularize constant term)

- Solve with normal equations like before

$$\min_{\theta} \|\Phi\theta - y\|_2^2 + \lambda \theta^T \theta$$

$$\min_{\theta} \theta^T \Phi^T \Phi \theta - 2y^T \Phi \theta + y^T y + \lambda \theta^T \theta$$

$$\min_{\theta} \theta^T (\Phi^T \Phi + \lambda I) \theta - 2y^T \Phi \theta + y^T y$$

- Setting gradient to zero

$$\theta^* = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T y$$
Evaluating algorithms

- How do we determine when an algorithm achieves “good” performance?
- How should we tune the parameters of the learning algorithms (regularization parameter, choice of features, etc?)
- How do we report the performance of learning algorithms?
- One possibility: just look at the loss function

\[ J(\theta) = \sum_{i=1}^{m} \ell(\theta^T \phi(x_i), y_i) \]

- The problem: adding more features will \textit{always} decrease the loss

- Example example: random outputs, random features, we can get zero loss for enough features

```matlab
m = 500;
y = randn(m,1);
Phi = randn(m,m);
theta = (Phi\' * Phi) \ (Phi\' * y);
norm(Phi*theta - y)^2

ans =
  2.3722e-22
```
A better criterion: \textit{training} and \textit{testing} loss

- \textbf{Training set:} $x_i \in \mathbb{R}^n, y_i \in \mathbb{R}, \ i = 1, \ldots, m$

- \textbf{Testing set:} $x'_i \in \mathbb{R}^n, y'_i \in \mathbb{R}, \ i = 1, \ldots, m'$

Find parameters by minimizing loss on the training set, but evaluate on the testing set

\[
\text{Training: } \theta^* = \arg \min_{\theta} \sum_{i=1}^{m} \ell(\theta^T \phi(x_i), y_i)
\]

\[
\text{Evaluation: Average Loss } = \frac{1}{m'} \ell((\theta^*)^T \phi(x'_i), y'_i)
\]

Performance on test set called \textit{generalization} performance.
• Sometimes, there is a natural breakdown between training and testing data (e.g., train system on one year, test on the next)

• More common, simply divide the data: for example, use 70% for training, 30% for testing

```matlab
% Phi, y, m are all the data
m_train = ceil(0.7*m);
m_test = m - m_train;
p = randperm(m);

Phi_train = Phi(p(1:m_train),:);
y_train = y(p(1:m_train));
Phi_test = Phi(p(m_train+1:end),:);
y_test = y(p(m_train+1:end));
```
High temperature / peak demand observations
Testing loss versus degree of degree of polynomial

Average squared loss

Polynomial degree, $d$

Training set

Testing set
Testing loss (log-scale) versus degree of polynomial
Testing loss versus number of RBF bases
Testing loss (log-scale) versus number of RBF bases
Testing loss (log-scale) versus regularization parameter (log-scale), for 70 RBF bases.
A common mistake: split the data into training/testing sets, use testing set to find best performing features, regularization parameter, kernel parameters, etc (*hyperparameters*), then report the testing error for these best features.
• This is not a valid method for evaluating error: the problem is that we effectively used the testing set to “train” the system

• What we need to do instead: break the training set itself into two sets (training and cross-validation) sets
• **Cross-validation Procedure:**

1. Break all data into training/testing sets (e.g., 70%/30%)

2. Break training set into training/cross-validation set (e.g., 70%/30% again)

3. Choose hyperparameters using cross-validation set

4. (Optional) Once we have selected hyperparameters, retrain using all the training set

5. Evaluate performance on the testing set
$k$-fold cross-validation: Split training set into $k$ different “folds” (equally sized random subsets)
  - For each fold $i$, train on $k - 1$ only folks, evaluate on held out fold $i$

The extreme case, leave one out cross validation: folds are individual examples
Non-linear regression in higher dimensions
If we want to report performance of an algorithm, how do we do this?

Reporting just test error doesn’t give a sense of our “confidence” in the prediction.
- If we have a testing set of size 1000, doesn’t this imply more confidence in result than a testing set of size 10?

What about variance in predictions? Are we getting some almost completely right and others very wrong?
Setting: in our test set, we have a number of actual labels $y_i'$, and predictions $\hat{y}_i'$ of our algorithm.

There are really two things we may care about:

1. What is the distribution of our errors $y_i' - \hat{y}_i'$?

2. If we want to report some average loss

   $$\text{Average loss} = \frac{1}{m'} \sum_{i=1}^{m'} \ell(\hat{y}_i', \hat{y}_i)$$

   how confident are we in this value?
Some basic probability notation

- We’ll use $Z$ to denote a *random variable* (with distribution $\mathcal{D}$), and use $p(z)$ to denote its probability density.

- Expected value, or mean:

  $$\mu = \mathbb{E}[Z] = \int z p(z) \, dz$$

- Variance

  $$\sigma^2 = \mathbb{E}[(Z - \mu)^2]$$

- If you haven’t seen any of this notation before, there are a number of good reviews available.
Suppose we have \( m \) samples drawn from the probability distribution \( \mathcal{D} \), written as \( z_1, \ldots, z_m \sim \mathcal{D} \).

Then we can form empirical estimates of the mean and variance of the distribution

\[
\hat{\mu} = \frac{1}{m} \sum_{i=1}^{m} z_i
\]

\[
\hat{\sigma}^2 = \frac{1}{m} \sum_{i=1}^{m} (z_i - \mu)^2 \approx \frac{1}{m} \sum_{i=1}^{m} (z_i - \hat{\mu})^2
\]

[You may have seen variance estimates with a \( \frac{1}{m-1} \) term instead; this is needed to make the estimator unbiased, but we’ll typically deal with large \( m \), so there isn’t much difference]
As mentioned before, we might want to know about the distribution over our prediction errors $\hat{y}_i' - y_i'$.

Histogram of errors $\hat{y}_i' - y_i'$
• Treat $\hat{y}_i' - y_i'$ as samples from a distribution

• Might want to know about the mean (also called bias), or variance of this distribution

• If we assume prediction errors are zero-mean (but this is not always the case), then

$$\hat{\sigma}^2 = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i' - y_i')^2$$

which is the mean squared error
If we want to report some average loss, then we can treat $\ell(\hat{y}_i', y_i')$ (for any loss) as the random samples (the average loss is just the mean of these samples)

Histogram of losses $\ell(\hat{y}_i', y_i')$ for absolute loss
How confident are we in our estimate of the mean (i.e., the average loss)?

Here we’ll exploit the central limit theorem: If \( z_1, \ldots, z_m \) are (independent, identically distributed) samples from any distribution with mean \( \mu \) and variance \( \sigma^2 \), then

\[
\frac{1}{m} \sum_{i=1}^{m} z_i \to \mathcal{N}(\mu, \sigma^2/m)
\]

- I.e., the mean of any set of random variables is normally distributed

For a normal distribution, 95\% of the data falls within 1.96 standard deviations \( \sigma \).
This suggests a method for computing “confidence intervals” of our estimate of the average loss

1. Form estimate of the mean:

\[ \hat{\mu} = \frac{1}{m'} \sum_{i=1}^{m'} \ell(\hat{y}_i', y_i) \]

2. Form estimate of the variance:

\[ \hat{\sigma}^2 = \frac{1}{m'} \sum_{i=1}^{m'} (\ell(\hat{y}_i', y_i') - \hat{\mu})^2 \]

3. With 95% confidence, the “true” mean lies within

\[ \hat{\mu} \pm 1.96 \hat{\sigma} / \sqrt{m'} \]

This procedure is technically wrong (we should be using the a different estimate of the variance, and a Student-t distribution instead of Gaussian), but it is close enough when \( m' \) is reasonably large, which is usually our setting.
• Should report errors relative to some baseline (i.e., degree zero polynomial)

<table>
<thead>
<tr>
<th>Degree</th>
<th>Test Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2414 ± 0.0039</td>
</tr>
<tr>
<td>1</td>
<td>0.2407 ± 0.0027</td>
</tr>
<tr>
<td>2</td>
<td>0.1505 ± 0.0013</td>
</tr>
<tr>
<td>3</td>
<td>0.1255 ± 0.0009</td>
</tr>
<tr>
<td>4</td>
<td>0.1257 ± 0.0009</td>
</tr>
<tr>
<td>5</td>
<td>0.1267 ± 0.0009</td>
</tr>
</tbody>
</table>

• A better way of determining how algorithms compare: pairwise hypothesis testing
Alternative loss functions

- Nothing special about least-squares loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$.

- Some alternatives:
  
  **Absolute loss:** $\ell(\hat{y}, y) = |\hat{y} - y|$
  
  **Deadband loss:** $\ell(\hat{y}, y) = \max\{0, |\hat{y} - y| - \epsilon\}, \quad \epsilon \in \mathbb{R}_+$
How do we find parameters that minimize absolute loss?

\[
\begin{align*}
\min_{\theta} \sum_{i=1}^{m} |\theta^T \phi(x_i) - y_i|
\end{align*}
\]

- Non-differentiable, can’t take gradient

Solution: frame as a constrained optimization problem
- Introduce new variables \( \nu \in \mathbb{R}^m, (\nu_i \geq |\theta^T \phi(x_i) - y_i|) \)

\[
\begin{align*}
\min_{\theta, \nu} \sum_{i=1}^{m} \nu_i \\
\text{subject to } -\nu_i \leq \theta^T \phi(x_i) - y_i \leq \nu_i
\end{align*}
\]

- Linear program (LP): linear object and linear constraints
Aside: general optimization problems

In this class we'll consider general optimization problems

\[
\begin{align*}
\text{minimize} & \quad J(\theta) \\
\text{subject to} & \quad g_i(\theta) \leq 0, \quad i = 1, \ldots, N_i \\
& \quad h_i(\theta) = 0, \quad i = 1, \ldots, N_e
\end{align*}
\]

A constrained optimization problem; \( g_i \) terms are the inequality constraints; \( h_i \) terms are the equality constraints.

Many different classifications of optimization problems (linear programming, quadratic programming, semidefinite programming, integer programming), depending on the form of \( J \), the \( g_i \)'s and the \( h_i \)'s.
Important distinctions in optimization is between *convex* (where $J$, $g_i$ are convex and $h_i$ linear) and *non-convex* problems

$$ f \text{ convex } \iff f(a\theta + (1 - a)\theta') \leq af(\theta) + (1 - a)f(\theta') $$

for $0 \leq a \leq 1$

Informally speaking, we can usually find *global* solutions of convex problems efficiently, while for non-convex problems we must settle for *local* solutions or time-consuming optimization.
Solving optimization problems

- Many generic optimization libraries

- We will be using YALMIP (Yet Another Linear Matrix Inequality Parser): http://users.isy.liu.se/johanl/yalmip/

- YALMIP code for least squares optimization:

  ```
  theta = sdpvar(n,1);
  solvesdp([], sum((Phi*theta - y).^2));
  double(theta)
  
  ans  =
      0.0466
     -1.4600
  ```
To solve LPs, typically need to put them in standard form:

\[
\begin{array}{c}
\text{minimize} \quad c^T z \\
\text{subject to} \quad Az \leq b
\end{array}
\]

- \( z \in \mathbb{R}^n, \quad A \in \mathbb{R}^{N_i \times n}, \quad b \in \mathbb{R}^{N_i} \)

For absolute loss LP

\[
\begin{align*}
    z &= \begin{bmatrix} \theta \\ \nu \end{bmatrix}, \\
    c &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \\
    A &= \begin{bmatrix} \Phi & -I \\ -\Phi & -I \end{bmatrix}, \\
    b &= \begin{bmatrix} y \\ -y \end{bmatrix}
\end{align*}
\]
MATLAB code

c = [zeros(n,1); ones(m,1)];
A = [Phi -eye(m); -Phi -eye(m)];
b = [y; -y];
z = linprog(c,A,b);
theta = z(1:n)

theta =
  0.0477
 -1.5978

The same solution in YALMIP:

theta = sdpvar(n,1);
solvesdp([], sum(abs((Phi*theta - y))));
double(theta)

theta =
  0.0477
 -1.5978
Which loss function should we use?