Each section is a new lecture.

1 Applications

- slides on applications of machine learning.
- calculus/programming quiz.

2 Training data in supervised learning

Training data: $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, given/known.
Inputs $x_i \in \mathcal{X}$, outputs $y_i \in \dagger$.
For each example, what is $\mathcal{X}, \mathcal{Y}$?

- Image classification
- Image segmntation
- Translation
- Spam filtering

Usually $\mathcal{X} = \mathbb{R}^p$ (e.g. word counts in emails).
Notation.

- Inputs $x \in \mathbb{R}^p$.
- p = dimension of each input.
- n = number of training examples.
- Outputs $y \in \mathbb{R}$ for regression, $y \in \{0, 1\}$ for binary classification.
- $f : \mathcal{X} \to \mathcal{Y}$ is the prediction function we want to learn.
- Regression function $f : \mathbb{R}^p \to \mathbb{R}$
- Binary classification function $f : \mathbb{R}^p \to \{0, 1\}$.

Geometric interpretation of regression, height son/father (linear pattern), species versus temperature (non-linear pattern).
Draw residual r_i on graphs as vertical line segments.

Test data. $D' = \{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$, unknown. Goal is generalization to new/test data: algo $\textsc{Learn}(D) = f$, with $f(x'_i) \approx y'_i$ for all test data, i.e. minimize

$$\text{Err}_{D'}(f) = \sum_{(x',y') \in D'} \ell[f(x'), y'] \tag{1}$$

where ℓ is a loss function:

- In binary classification we typically use the mis-classification-rate/0-1-loss $\ell[f(x), y] = I[c_f(x)! = y]$, where the binary classifier $c_f(x) = 0$ if $f(x) < 0.5$ and 1 otherwise.
- In regression we use the squared error $\ell[f(x), y] = [f(x) - y]^2$.

Since D' is unknown, we need to assume that $D, D' \sim \mathcal{P}$.

Nearest neighbors algorithm. What is near? $\text{DIST} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+$ is a non-negative distance function between two data points.

e.g. for $\mathcal{X} = \mathbb{R}^p$ we use

- L1 distance $\text{DIST}(x, x') = ||x - x'||_1 = \sum_{j=1}^{p} |x_j - x'_j|$
- L2 distance $\text{DIST}(x, x') = ||x - x'||_2 = \sqrt{\sum_{j=1}^{p} (x_j - x'_j)^2}$

Draw geometric interpretation of L1 and L2 distances.

Compute distances $d_1, \ldots, d_n \in \mathbb{R}_+$ where $d_i = \text{DIST}(x_i, x)$.

Example graph, draw horizontal line segments d_i between x, x_i.

Define neighbors function $N_{D,K}(x') = \{t_1, t_2, \ldots, t_K\}$, where $t_1, \ldots, t_n \in \{1, \ldots, n\}$ are indices such that distances $d_{t_1} \leq \cdots \leq d_{t_n}$ are sorted ascending.

Predict the mean output value of the K nearest neighbors,

$$f_{D,K}(x') = \frac{1}{K} \sum_{i \in N_{D,K}(x')} y_i$$

- For regression this is the mean.
- For binary classification the mean is intepreted as a probability, predict 1 if greater than 0.5, and predict 0 otherwise.

3 Nearest neighbors model selection and code

- Review of supervised learning.
- Definition of nearest neighbors prediction function $f_{D,K}(x)$.
- How to choose K? Discussion.
- Formal definition of total error.
- Visualizations. 1d regression and binary classification.
- Draw train/validation matrices. Validation error.
- Pseudocode for computing $f_{D,K}(x)$.

$$\mathcal{L}_{D,D'}(k) = \sum_{(x',y') \in D'} \ell[f_{D,k}(x'), y'] \tag{2}$$

Function PredKNearestNeighbors
Inputs: train inputs/features $x_1, \ldots, x_n$, outputs/labels $y_1, \ldots, y_n$,
  test input/feature x', max number of neighbors $K_{\max}$:
for $i = 1$ to n **do**:
  $d_i \leftarrow \text{Distance}(x', x_i)$
end for
$t_1, \ldots, t_n \leftarrow \text{SortedIndices}(d_1, \ldots, d_n)$
totalY $\leftarrow 0.0$
for $k = 1$ to $K_{\max}$ **do**:
  $i \leftarrow t_k$
  totalY `+=` y_i
end for
Output: prediction totalY$/K_{\max}$

Total complexity: $O(np + n \log n)$ where the Distance sub-routine is $O(p)$.

4 Cross-validation

Interactive data viz.

Remember we assume that training data and test data are randomly drawn from a common probability distribution $\mathcal{P}$. To simulate that process, we randomly assign a fold ID for each observation $z_1, \ldots, z_n \in \{1, \ldots, S_{\max}\}$. We then define NumFolds train/validation splits

$$D_{z=s} = \{(x_i, y_i) | z_i = s\} \text{ — validation set } sD_{z \neq s} = \{(x_i, y_i) | z_i \neq s\} \text{ — train set } s \tag{3}$$

Draw train/validation/test matrices for $s = 1$, $s = 2$, etc.

Cross-validation data visualizations.

Mean validation error over all train/validation splits (folds):

$$\text{MeanVerr}(k) = \mathcal{L}_{D_{z \neq s}, D_{z=s}}(k) \tag{4}$$

Overall we choose the parameter $\hat{k} = \arg\min_k \text{MeanVerr}(k)$.

And the learning algorithm is $\textsc{LearnKNN}(D) = f_{D,\hat{k}}$.

4.1 Pseudocode

We assume there are sub-routines

- $\textsc{Pred1toKmaxNN}(D, x', K_{\max}) = [f_{D,1}(x') ... f_{D,K_{\max}}(x')]$ – all predictions from $k = 1$ to $K_{\max}$ neighbors for a single test point x' based on the training data set D.
- $\textsc{PredError}(D, D', K_{\max})$ trains on D and computes $\mathcal{L}_{D,D'}(k)$, the validation error wrt D' for $k = 1$ to $K_{\max}$ neighbors.

Then the pseudo code for the learning algo is:

Function LearnKNN
Inputs: train data D (n observations in p dimensions), number of folds $S_{\max} \in \{1, \ldots, n\}$, max number of neighbors $K_{\max} \in \{1, \ldots, n\}$.
Randomly assign fold IDs $z \in \{1, \ldots, S_{\max}\}^n$
for each fold $s = 1$ to $S_{\max}$ **do**:
    $E_s \leftarrow \textsc{PredError}(D_{z \neq s}, D_{z=s}, K_{\max}) \in \mathbb{R}^{K_{\max}}$
end for Let $E \leftarrow [E_1 ... E_{S_{\max}}] \in \mathbb{R}^{K_{\max} \times S_{\max}}$ be the matrix of error/loss values for all folds/columns and neighbors/rows.
MeanVerr $\leftarrow \textsc{colMeans}(E) \in \mathbb{R}^{K_{\max}}$
The optimal number of neighbors is $\hat{k} = \arg\min_k \text{MeanVerr}_k$
return $f_{D,\hat{k}}$

5 R package development

Choose groups.

6 Coding nearest neighbors prediction

For one test point, demo code in C++.

7 Linear regression / gradient descent

Want to minimize squared error on test data, but we don't have them, so instead we do

$$\min_f \sum_{i=1}^{n} [f(x_i) - y_i]^2 \tag{5}$$

We parameterize a linear function using a weight vector $w \in \mathbb{R}^p$, $f_w(x_i) = w^T x_i$.

The input feature matrix is

$$X = \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix} \in \mathbb{R}^{n \times p} \tag{6}$$

So for a partcular weight vector w the prediction vector is

$$Xw = \begin{bmatrix} x_1^T w \\ \vdots \\ x_n^T w \end{bmatrix} \in \mathbb{R}^n \tag{7}$$

So the least squares regression problem is

$$\min_{w \in \mathbb{R}^p} ||Xw - y||_2^2 \tag{8}$$

Notation about matrix/vector dimensions.

What is a norm $|| \cdot ||$ = size of a vector.

Squared L2 norm $||v||_2^2 = v^T v$.

L1 norm $||v||_1 = \sum_{j=1}^{p} |v_j|$.

Univariate linear regression: $p = 2$, $x_i = [1 \; x_{i,2}]$.

$$X = \begin{bmatrix} 1 & x_{1,2} \\ \vdots & \vdots \\ 1 & x_{n,2} \end{bmatrix} \in \mathbb{R}^{n \times 2} \tag{9}$$

and $f(x_i) = w^T x_i = \underbrace{w_1}_{\text{intercept}} + \underbrace{w_2}_{\text{slope}} x_{i,2}$. Draw figure on board then show data viz.

Gradient is defined for any w as the direction of steepest ascent:

$$\nabla \mathcal{L}(w) = \begin{bmatrix} \frac{d}{dw_1} \mathcal{L}(w) \\ \vdots \\ \frac{d}{dw_p} \mathcal{L}(w) \end{bmatrix} \in \mathbb{R}^p \tag{10}$$

For $p = 1$ we have

$$\nabla \mathcal{L}(w) = \mathcal{L}'(w) = \frac{d}{dw} \mathcal{L}(w) = \sum_{i=1}^{n} 2x_i(wx_i - y_i). \tag{11}$$

In general we have

$$\nabla \mathcal{L}(w) = \nabla_w ||Xw - y||_2^2 = 2X^T(Xw - y). \tag{12}$$

Algorithm: $w^{(0)} = 0$ be the initial guess. Then for any stepsize $\alpha > 0$ define update rules for any iteration $t > 0$:

$$w^{(t)} = w^{(t-1)} - \alpha \nabla \mathcal{L}(w^{(t-1)}). \tag{13}$$

Example: gradient descent on $\mathcal{L}(w) = (x + 5)^2$. Start at the origin. Compute gradient, first step.

8 Probabilistic interpretation of linear regression

Definitions of argmin, min.

$$\arg\min_{w \in \mathbb{R}^p} \mathcal{L}(w) = \{w^* \in \mathbb{R}^p | \mathcal{L}(w^*) \leq L(w_0) \forall w_0 \in \mathbb{R}^p\}.$$

Draw quadratic function x^2 and $x^2 + 1$ which have the same argmin but different min.

Definition of argmax: draw upside down quadratic function $-x^2$ to show relationship between argmin and argmax.

Why do we need to do this derivation?

1. For binary classification can't minimize zero-one loss via gradient descent.
2. Using $w^T x_i \in \mathbb{R}$ directly for binary classification does not make sense.

Begin by re-deriving the least squares problem, assuming labels are normally distributed.

Let $y_i \sim N(w^T x_i, s^2)$, draw probability density function.

$$\Pr(y_i, w^T x_i, s^2) = (2\pi s^2)^{-1/2} \exp\left[\frac{-1}{2s^2}(y_i - w^T x_i)^2\right]$$

Let the likelihood of w be

$$\text{Lik}(w) = \prod_{i=1}^{n} \Pr(y_i, w^T x_i, s^2)$$

Then the log-likelihood is

$$\log \text{Lik}(w) = \sum_{i=1}^{n} \log \Pr(y_i, w^T x_i, s^2)$$

Then we have

$$\begin{aligned}
\arg\max_w \log \text{Lik}(w) &= \arg\max_w \sum_{i=1}^{n} \underbrace{\frac{-1}{2}\log(2\pi s^2)}_{\text{constant wrt } w} - \frac{1}{2s^2}(y_i - w^T x_i)^2 \\
&= \arg\max_w -\sum_{i=1}^{n} \underbrace{\frac{1}{2s^2}}_{\text{cst}} (y_i - w^T x_i)^2 \\
&= \arg\min_w \sum_{i=1}^{n} (y_i - w^T x_i)^2
\end{aligned}$$

For binary labels assume Bernoulli distribution. Exercise: Derive logistic loss, assuming $\tilde{y}_i \in \{-1, 1\}$.

9 Logistic regression

Assume $y_i \sim \text{Bernoulli}(p_i)$ with $p_i = \sigma(w^T x_i) \in (0, 1)$ and

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

The probability mass function is

$$\Pr(y_i, p_i) = \begin{cases} p_i & \text{if } y_i = 1 \\ 1 - p_i & \text{if } y_i = 0. \end{cases}$$

The log-likelihood is then

$$\begin{aligned}
\log \mathrm{Lik}(w) &= \sum_{i=1}^{n} \log \Pr(y_i, \sigma(w^T x_i)) \\
&= \sum_{i=1}^{n} \log[\sigma(w^T x_i)] I(y_i = 1) + \log[1 - \sigma(w^T x_i)] I(y_i = 0) \\
&= \sum_{i=1}^{n} \log[\frac{1}{1+\exp(-w^T x_i)}] I(y_i = 1) + \log[\underbrace{1 - \frac{1}{1+\exp(-w^T x_i)}}_{\frac{\exp(-w^T x_i)}{1+\exp(-w^T x_i)}}] I(y_i = 0) \\
&= \sum_{i=1}^{n} \log[\frac{1}{1+\exp(-w^T x_i)}] I(\tilde{y}_i = 1) + \log[\frac{1}{1+\exp(w^T x_i)}] I(\tilde{y}_i = -1) \\
&= \sum_{i=1}^{n} \log[\frac{1}{1+\exp(-\tilde{y}_i w^T x_i)}] \\
&= \sum_{i=1}^{n} -\log[1+\exp(-\tilde{y}_i w^T x_i)]
\end{aligned}$$

where the alternate labels are

$$\tilde{y}_i = \begin{cases} -1 & \text{if } y_i = 0 \\ 1 & \text{if } y_i = 1. \end{cases}$$

The logistic loss function that we want to minimize is thus

$$\mathcal{L}(w) = -\log \mathrm{Lik}(w) = \sum_{i=1}^{n} \underbrace{\log[1+\exp(-\tilde{y}_i w^T x_i)]}_{\ell[w^T x_i, \tilde{y}_i]}.$$

The gradient is

$$\nabla_w \ell[w^T x_i, \tilde{y}_i] = \frac{-\tilde{y}_i x_i \exp(-\tilde{y}_i w^T x_i)}{1+\exp(-\tilde{y}_i w^T x_i)} = \frac{-\tilde{y}_i x_i}{1+\exp(\tilde{y}_i w^T x_i)}$$

Draw the logistic loss for $y_i = 1$ as a function of $w^T x_i$.

$$\begin{aligned}
\ell[w^T x_i, \tilde{y}_i] &= \log[1+\exp(-w^T x_i)] \\
\nabla_{w^T x_i} \ell[w^T x_i, \tilde{y}_i] &= \frac{-1}{1+\exp(w^T x_i)}
\end{aligned}$$

- As $w^T x_i \to \infty$ we have $\exp(-w^T x_i) \to 0$ and $\log[1+\exp(-w^T x_i)] \to 0$.
- As $w^T x_i \to -\infty$ we have $\exp(-w^T x_i) \to \infty$, $\ell[w^T x_i, \tilde{y}_i] = \log[1+\exp(-w^T x_i)] \to \infty$, $\exp(w^T x_i) \to 0$, and $\nabla_{w^T x_i} \ell[w^T x_i, \tilde{y}_i] \to -1$.

The gradient $\nabla \mathcal{L} : \mathbb{R}^p \to \mathbb{R}^p$ is

$$\nabla \mathcal{L}(w) = \sum_{i=1}^{n} \nabla_w \ell[w^T x_i, \tilde{y}_i] = \sum_{i=1}^{n} \frac{-\tilde{y}_i x_i}{1+\exp(\tilde{y}_i w^T x_i)} = \sum_{i=1}^{n} -\tilde{y}_i x_i \sigma(-\tilde{y}_i w^T x_i) = -X^T \tilde{Y} S(-\tilde{Y} X w),$$

where $\tilde{Y} = \mathrm{Diag}(\tilde{y})$ and $S : \mathbb{R}^n \to \mathbb{R}^n$ is the componentwise application of the logistic link function, $S(v) = \begin{bmatrix} \sigma(v_1) & \cdots & \sigma(v_n) \end{bmatrix}^T$.

Discuss gradient descent algorithm, same as least squares regression.

Discuss learning algorithm which chooses the number of steps (early stopping) that minimizes the mean validation loss from CV.

10 L2 regularization

Define the cost function

$$C_\lambda(w) = \mathcal{L}(w) + \lambda||w||_2^2. \tag{14}$$

- The loss $\mathcal{L}(w)$ encourages fitting the train data.
- The squared L2 norm $||w||_2^2$ encourages a regularized model.
- $\lambda \geq 0$ is a penalty constant (larger for more regularization).

For a given penalty λ the L2 regularized linear model is defined as $f_w(x) = w^T x$, using f_{w^λ}, where w^λ is the weight vector with minimal cost:

$$w^\lambda = \arg\min_{w \in \mathbb{R}^p} C_\lambda(w). \tag{15}$$

The learning algorithm then looks like:

- For each train/validation split do:
- For $\lambda \in \{\lambda_1, \ldots, \lambda_m\}$ compute the optimal weight vector w^λ, where m is the number of penalty values/models considered. Combine them into a matrix $W \in \mathbb{R}^{p \times m}$.
- Compute prediction matrix $\hat{Y} = XW \in \mathbb{R}^{n \times m}$.
- Compute average loss on this validation set, $\mathcal{L}(\hat{Y}, Y)$.
- Let $\hat{\lambda} = \arg\min_\lambda \text{MeanValidationLoss}(\lambda)$.
- Use $f_{\hat{\lambda}}$ for the final predictions on test data.

To compute w^λ we need the gradient:

$$\nabla C(w) = \nabla \mathcal{L}(w) + 2\lambda w. \tag{16}$$

The gradent of the loss $\nabla \mathcal{L}$ is the same as we computed earlier for the early stopping model.

In the statistics literature we usually parameterize the linear function $f_{\beta,w}(x) = \beta + x^T w$ with an additional intercept/bias term $\beta \in \mathbb{R}$, which is un-penalized:

$$C_\lambda(\beta, w) = \mathcal{L}(\beta, w) + \lambda||w||_2^2. \tag{17}$$

Then we need to compute the gradient with respect to both variables, $\nabla_\beta C(\beta, w), \nabla_w C(\beta, w)$.

11 Scaling

The gradient descent will only work if the inputs are scaled. Ex input matrix with height in mm, weight in kg, shoe size.

We start with an unscaled input matrix $X \in \mathbb{R}^{n \times p}$ and we want to do gradient descent on a scaled input matrix $\tilde{X} \in \mathbb{R}^{n \times \tilde{p}}$, where $\tilde{p} \leq p$ is fewer features than we started out with, because some with zero variance may be filtered before doing gradient descent.

To define the scaled features we need to compute the mean m_j and standard deviation s_j of each column $j \in \{1, \ldots, p\}$ of X:

$$\begin{aligned} m_j &= \frac{1}{n}\sum_{i=1}^{n} x_{ij} \\ s_j &= \sqrt{\frac{1}{n}\sum_{i=1}(x_{ij} - m_j)^2} \end{aligned}$$

We define the set of features with non-zero variance in the train set as $\mathcal{J} = \{j | s_j > 0\}$. The size of this set is $\tilde{p} = |\mathcal{J}|$, the number of features/columns in the scaled data matrix. Let $x_{\mathcal{J}} \in \mathbb{R}^{\tilde{p}}$ be the unscaled feature vector, omitting the entries/features which have zero variance in the train set.

Then let $s = \begin{bmatrix} s_1 & \cdots & s_p \end{bmatrix}^T \in \mathbb{R}^p$ be the vector of standard deviations for all features, and let $\tilde{s} = \begin{bmatrix} \tilde{s}_1 & \cdots & \tilde{s}_{\tilde{p}} \end{bmatrix}^T = s_{\mathcal{J}} \in \mathbb{R}^{\tilde{p}}$ be the standard deviations for all features with non-zero variance in the training set. The diagonal scaling matrix is

$$\tilde{S}^{-1} = \text{Diag}(\tilde{s})^{-1} = \begin{bmatrix} 1/\tilde{s}_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & 1/\tilde{s}_{\tilde{p}} \end{bmatrix} \in \mathbb{R}^{\tilde{p} \times \tilde{p}} \tag{18}$$

Then for any unscaled feature vector $x \in \mathbb{R}^p$ we define the corresponding scaled feature vector as

$$\tilde{x} = \tilde{S}^{-1}(x_{\mathcal{J}} - m_{\mathcal{J}}). \tag{19}$$

So the linear model on a scaled feature vector $\tilde{x} \in \mathbb{R}^{\tilde{p}}$ is

$$\tilde{f}_{\tilde{w}}(\tilde{x}) = \tilde{w}^T \tilde{x}, \tag{20}$$

where $\tilde{w} \in \mathbb{R}^{\tilde{p}}$ is the scaled weight vector. The cost we want to minimize in gradient descent is then

$$\tilde{C}_\lambda(\tilde{w}) = \tilde{\mathcal{L}}(\tilde{w}) + \lambda ||\tilde{w}||_2^2. \tag{21}$$

Gradient descent gives us an optimal scaled weight vector $\tilde{w}^\lambda = \arg\min_{\tilde{w} \in \mathbb{R}^{\tilde{p}}} \tilde{C}_\lambda(\tilde{w})$. To make a prediction using a new unscaled test data point $x \in \mathbb{R}^p$ we

- scale it via (**??**), which gives us $\tilde{x} \in \mathbb{R}^{\tilde{p}}$.
- compute the inner product with the learned weight vector (**??**).

It is equivalent (and more convenient for the user) to return the weight vector w in the original (unscaled) space:

$$\tilde{f}_{\tilde{w}}(\tilde{x}) = \tilde{x}^T \tilde{w} = (x_{\mathcal{J}} - m_{\mathcal{J}})^T \tilde{S}^{-1} \tilde{w} = x_{\mathcal{J}} \underbrace{\tilde{S}^{-1} \tilde{w}}_{w} \underbrace{- m_{\mathcal{J}} \tilde{S}^{-1} \tilde{w}}_{\beta}. \tag{22}$$

which implies that the prediction function $f_{\beta,w}(x)$ for an unscaled input/feature vector $x \in \mathbb{R}^p$ is defined by the bias/intercept $\beta \in \mathbb{R}$ and weight vector w as above. Actually there is a slight abuse of notation: the equation above only defines the weights for the features $j \in \mathcal{J}$ with non-zero variance in the train data. The other features $j \notin \mathcal{J}$ with zero variance in the train data should have zero weight in the final weight vector w.

12 Exact line search

Gradient descent is for finding a weight vector $w \in \mathbb{R}^p$ which minimizes a smooth cost function $C(w)$.

We start the algorithm at the origin, $w^{(0)} = 0$.

For each iteration t we compute the descent direction

$$d^{(t)} = -\nabla C(w^{(t)}) \tag{23}$$

Then we define the next iteration by taking a step in that direction,

$$w^{(t+1)} = w^{(t)} + \alpha^{(t)} d^{(t)}, \tag{24}$$

where $\alpha^{(t)} > 0$ is an iteration-specific step size parameter.

For line search the main idea is that we want to choose a step size that minimizes the cost:

$$\mathcal{C}_t(\alpha) = C(w^{(t)} + \alpha d^{(t)}). \tag{25}$$

For exact line search we take the best possible step,

$$\alpha^{(t)} = \arg\min_{\alpha} \mathcal{C}_t(\alpha). \tag{26}$$

Example: a function with a minimum at (-1, 2).

$$C(w) = 0.5(w_1 + 1)^2 + 0.5(w_2 - 2) = 0.5||w + \begin{bmatrix} 1 & -2 \end{bmatrix}^T ||_2^2. \tag{27}$$

Derive the exact line search step.

The exact line search step can be derived for the L2-regularized linear model for regression.

$$\begin{aligned} \mathcal{C}_t(\alpha) &= 0.5||X(w^{(t)} + \alpha d^{(t)}) - y||_2^2 + 0.5\lambda||w^{(t)} + \alpha d^{(t)}||_2^2 & (28)\\ \mathcal{C}_t'(\alpha) &= d^{(t)T}X^T[X(w^{(t)} + \alpha d^{(t)}) - y] + \lambda d^{(t)T}[w^{(t)} + \alpha d^{(t)}]. & (29) \end{aligned}$$

Setting the derivative equal to zero and solving for α yields

$$\alpha^{(t)} = \arg\min_{\alpha} \mathcal{C}_t(\alpha) = \frac{d^{(t)T}\overbrace{[-X^T(Xw^{(t)} - y)}^{d^{(t)}} - \lambda w^{(t)}]}{d^{(t)T}(X^TX + \lambda I_p)d^{(t)}} = \frac{||d^{(t)}||_2^2}{||Xd^{(t)}||_2^2 + \lambda||d^{(t)}||_2^2} \tag{30}$$

Is it any slower than the constant gradient step? We are already computing the gradient / descent direction,

$$d^{(t)} = -\nabla C(w^{(t)}) = -X^T(Xw^{(t)} - y). \tag{31}$$

- Computing each element of $Xw^{(t)} \in \mathbb{R}^n$ is an $O(p)$ operation, so overall time is $O(np)$.
- The next operation, subtracting y is $O(n)$.
- For the next operation, left-multiply by $-X^T$, computing each of the p elements is $O(n)$, so overall time is $O(np)$.
- Overall computation of $d^{(t)}$ is thus $O(np)$.

Computing the largest component of $\alpha^{(t)}$, $||Xd^{(t)}||_2^2$, is also $O(np)$ so there is no asymptotic difference in time complexity between constant and exact line search (for this problem).

13 Backtracking line search

Backtracking line search is more general than exact line search, in the sense that it can be used on more cost functions. It performs approximate/numerical solution of the optimal step size (rather than the analytic solution used by exact line search).

The main idea is to start by trying a big $\alpha = 1$. If the cost increases then the step is too big, so decrease the step by a reduction factor $\rho \in (0, 1)$. Keep decreasing the step size until the cost decreases (it is guaranteed to when d is a descent direction). In practice the range of ρ values is from 0.1 (crude search) to 0.8 (more exhaustive search).

In practice the simple rule above is not used, because it may result in a sequence of steps with very small decreases in cost. Instead we introduce a parameter $\gamma \in (0, 0.5)$ which controls how much progress/decrease is significant/acceptable. We want α such that

$$\mathcal{C}_t(\alpha) = C(w^{(t)} + \alpha d^{(t)}) < C(w^{(t)} + \gamma\alpha\nabla C(w^{(t)})^T d^{(t)}) = \mathcal{A}_t^{\gamma}(\alpha) \tag{32}$$

When $\gamma = 0$ then a small decrease in cost is acceptable; when $\gamma = 0.5$ then only a very large decrease is acceptable. In practice γ is usually between 0.01 and 0.3 (**?**, page 466).

The approximate cost $\mathcal{A}_t^\gamma(\alpha)$ for $\gamma = 1$ is a linear approxmation of the cost around $w^{(t)}$, which underestimates the cost since it is a convex function. By using a $\gamma < 0.5$ we get a linear approximation between the min cost and min decrease (draw plot).

14 Neural networks

We follow the presentation/notation of **?**, section 16.5.

Draw neural network diagrams for logistic regression and e.g. (p, 5, 10, 1) network has two hidden layers and one output.

15 Backpropagation algorithm for single layer regression network

$$x \to^V a \to^\sigma z \to^w b$$

Overall prediction function is

$$f(x_i) = b_i = w^T z_i = w^T \sigma(a_i) = w^T S(V^T x_i)$$

Feature matrix is $X \in \mathbb{R}^{n \times p}$. Feature vector for one observation is $x_i \in \mathbb{R}^p$. One feature of that observation is $x_{ij} \in \mathbb{R}$.

Labels are $y \in \mathbb{R}^n$ for regression. One label is $y_i \in \mathbb{R}$.

Hidden layer vector is $z_i = S(a_i) = S(V^T x_i) \in \mathbb{R}^u$. One hidden unit is $z_{ik} = \sigma(a_{ik}) \in \mathbb{R}$.

Indices are $i \in \{1, \ldots, n\}$ for observations/examples, $j \in \{1, \ldots, p\}$ for features, and $k \in \{1, \ldots, u\}$ for hidden units.

Weight matrix for first layer is $V \in \mathbb{R}^{p \times u}$. Weight vector for predicting hidden unit k is $v_k \in \mathbb{R}^p$.

Weight vector for predicting output is $w \in \mathbb{R}^u$.

Overall loss function that we want to minimize is

$$\mathcal{L}(w, V) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}[f(x_i) - y_i]^2 = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}[w^T S(V^T x_i) - y_i]^2 = \frac{1}{2n} ||S(XV)w - y||_2^2$$

The learning algorithm is as before with linear models: start at $w, V = 0$ (or some random values close) and then take steps in the opposite direction of the gradient. We therefore need to compute the gradients of $\mathcal{L}$ with respect to the parameters w, V. We consider the gradient with respect to a single observation i:

$$\nabla_w \frac{1}{2}[f(x_i) - y_i]^2 = \underbrace{\frac{\partial}{\partial b_i} \frac{1}{2}[b_i - y_i]^2}_{b_i - y_i = \delta_i^w} \nabla_w \underbrace{b_i}_{w^T z_i} = \delta_i^w z_i$$
$$\underbrace{\qquad\qquad}_{z_i}$$

And we consider the gradient of the weights $v_k \in \mathbb{R}^p$ used to predict hidden unit k:

$$\begin{aligned} \nabla_{v_k} \frac{1}{2}[f(x_i) - y_i]^2 &= \underbrace{\frac{\partial}{\partial a_{ik}} \frac{1}{2}[w^T S(a_i) - y_i]^2}_{\delta_{ik}^v} \nabla_{v_k} \underbrace{a_{ik}}_{v_k^T x_i} = \delta_{ik}^v x_i \\ &= \underbrace{[\frac{\partial}{\partial b_i} \frac{1}{2}(b_i - y_i)^2]}_{\delta_i^w} \underbrace{[\frac{\partial}{\partial a_{ik}} b_i]}_{w_k \sigma'(a_{ik})} x_i \end{aligned}$$

The first layer errors $\delta_i^v = \left[\begin{array}{ccc} \delta_{i1}^v & \cdots & \delta_{iu}^v \end{array} \right] \in \mathbb{R}^u$ can thus be written in terms of the second layer errors $\delta_i^w \in \mathbb{R}$.

Therefore the gradient of the full first layer weight matrix V with respect to one observation is

$$\nabla_V \frac{1}{2}[f(x_i) - y_i]^2 = x_i \delta_i^{vT}$$

The algorithm is to compute, in order:

quantity	one observation	batch
hidden before sigmoid	$a_i = V^T x_i \in \mathbb{R}^u$	$A = XV \in \mathbb{R}^{n \times u}$
hidden after sigmoid	$z_i = S(a_i) \in \mathbb{R}^u$	$Z = S(A) \in \mathbb{R}^{n \times u}$
predictions	$b_i = w^T z_i \in \mathbb{R}$	$b = Zw \in \mathbb{R}^n$
second level errors	$\delta_i^w = b_i - y_i \in \mathbb{R}$	$\delta^w = b - y \in \mathbb{R}^n$
first level errors	$\delta_i^v = \delta_i^w \operatorname{Diag}(w) S'(a_i) \in \mathbb{R}^u$	$\delta^v = \operatorname{Diag}(\delta^w) S'(A) \operatorname{Diag}(w) \in \mathbb{R}^{n \times u}$
second level gradient	$\nabla_w \frac{1}{2}[f(x_i) - y_i]^2 = \delta_i^w z_i \in \mathbb{R}^u$	$\nabla_w \mathcal{L}(w, V) = \frac{1}{n} \sum_{i=1}^n \delta_i^w z_i = Z^T \delta^w / n \in \mathbb{R}^u$
first level gradient	$\nabla_V \frac{1}{2}[f(x_i) - y_i]^2 = x_i \delta_i^{vT} \in \mathbb{R}^{p \times u}$	$\nabla_V \mathcal{L}(w, V) = \frac{1}{n} \sum_{i=1}^n x_i \delta_i^{vT} = X^T \delta^v / n \in \mathbb{R}^{p \times u}$

Note that since $\sigma'(t) = \sigma(t)[1 - \sigma(t)]$ the derivative of the sigmoid can be computed as

$$S'(a_i) = \operatorname{Diag}(z_i)(\mathbf{1}_u - z_i) \tag{33}$$

16 Backpropagation for single layer binary classification network

For binary classification we have outputs $\tilde{y}_i \in \{-1, 1\}$ and we use the logistic loss, which yields the mean loss

$$\frac{1}{n} \sum_{i=1}^n \log[1 + e^{-\tilde{y}_i f(x_i)}] = \frac{1}{n} \sum_{i=1}^n \log[1 + e^{-\tilde{y}_i w^T S[V^T x_i]}] \tag{34}$$

Recall that the predicted values are $f(x_i) = b_i \in \mathbb{R}$, which means the gradient of one observation i is

$$\nabla_w \log[1 + e^{-\tilde{y}_i b_i}] = \underbrace{\frac{\partial}{\partial b_i} \log[1 + e^{-\tilde{y}_i b_i}]}_{\delta_i^w} \underbrace{\nabla_w b_i}_{z_i}. \tag{35}$$

So the algorithm described in the last section is still valid, as long as the new second level errors are computed via

$$\delta_i^w = \frac{-\tilde{y}_i}{1 + \exp(\tilde{y}_i b_i)} = -\tilde{y}_i \sigma(-\tilde{y}_i b_i) \in \mathbb{R} \tag{36}$$

or in vector notation for n data points,

$$\delta^w = -\tilde{Y} S[-\tilde{Y} b] \in \mathbb{R}^n. \tag{37}$$

17 Scaling for neural networks

As in linear models, the train input matrix X must be scaled before gradient descent, for numerical stability. If $x_i \in \mathbb{R}^p$ is an unscaled input/feature vector, let $\tilde{x}_i = D[x_i - m]$ be the corresponding scaled input/feature vector. Then the hidden units before applying the sigmoid are

$$a_i = \tilde{V}^T \tilde{x}_i = \tilde{V}^T D(x_i - m) = \tilde{V}^T D x_i - \tilde{V}^T D m = V^T x_i + \beta = \left[\begin{array}{cc} \beta & V^T \end{array} \right] \left[\begin{array}{c} 1 \\ x_i \end{array} \right] \in \mathbb{R}^u, \tag{38}$$

where the weight matrix/intercept on the original scale are

$$\begin{aligned} V &= D\tilde{V} \in \mathbb{R}^{p \times u} \\ \beta &= -\tilde{V}^T D m \in \mathbb{R}^u. \end{aligned}$$

18 L1-regularized linear models

The goal is to learn a linear function $f(x_i) = w^T x_i$ by minimizing the L1-regularized cost function

$$C_\lambda(w) = \mathcal{L}(w) + \lambda ||w||_1, \tag{39}$$

where $\mathcal{L}$ is a function that measures how well the model with weight vector w fits the train data (typically the square loss for regression or the logistic loss for binary classification).

The learning algorithm is to compute the optimal weight vector

$$\hat{w}(\lambda) = \arg\min_w C_\lambda(w) \tag{40}$$

for a range of penalty λ parameters, and then select the model which minimizes the error with respect to a held-out validation set,

$$\hat{\lambda} = \arg\min_\lambda \text{MeanValidationError}[\hat{w}(\lambda)]. \tag{41}$$

The final prediction function that we use is $f(x_i) = \hat{w}(\hat{\lambda})^T x_i$.

19 Proximal gradient algorithm

We adapt the proximal gradient algorithm described by **?**, Section 13.4.3. It is characterized by the initialization $w^{(0)} = 0$ and updates for every $t \geq 0$

$$u^{(t)} = w^{(t)} + \alpha^{(t)} d^{(t)} \tag{42}$$

$$w^{(t+1)} = \text{Prox}_{\alpha^{(t)} R}(u^{(t)}) = \arg\min_z \alpha^{(t)} R(z) + \frac{1}{2}||z - u^{(t)}||_2^2 \tag{43}$$

where the descent direction is the negative gradient of the loss,

$$d^{(t)} = -\nabla \mathcal{L}(w^{(t)}), \tag{44}$$

and the proximal operator for the L1 regularizer $R(w) = \lambda ||w||_1$ is the soft-thresholding function:

$$\text{Prox}_R(w) = \text{soft}(w, \lambda) = \text{sign}(w)(|w| - \lambda)_+ \tag{45}$$

So after simplifying the update rule can be written as

$$w_j^{(t+1)} = \begin{cases} w_j^{(t)} + \alpha^{(t)} d_j^{(t)} & \text{for the un-regularized intercept } j = 0 \\ \text{soft}(w_j^{(t)} + \alpha^{(t)} d_j^{(t)}, \lambda \alpha^{(t)}) & \text{for the L1-regularized weights } j > 0 \end{cases} \tag{46}$$

20 Optimality criterion for L1-regularized linear models

The sub-differential can be used to characterize the minimum of the convex, non-smooth cost function:

$$w^* = \arg\min_w C_\lambda(w) \iff 0 \in \partial C_\lambda(w^*), \tag{47}$$

which implies

$$-\nabla \mathcal{L}(w) \in \lambda \partial ||w||_1, \tag{48}$$

or

$$\begin{cases} \frac{\partial}{\partial w_j} - \mathcal{L}(w) = \lambda & \text{if } w_j > 0 \\ \frac{\partial}{\partial w_j} - \mathcal{L}(w) = -\lambda & \text{if } w_j < 0 \\ \frac{\partial}{\partial w_j} - \mathcal{L}(w) \in [-\lambda, \lambda] & \text{if } w_j = 0 \end{cases} \tag{49}$$

We can therefore define the sub-optimality vector

$$c_j^{(t)} = \begin{cases} |d_j^{(t)} - \text{sign}(w_j^{(t)})\lambda| & \text{if } w_j \neq 0 \\ (|d_j^{(t)}| - \lambda)_+ & \text{if } w_j = 0. \end{cases} \tag{50}$$

When $c^{(t)}$ is big then $w^{(t)}$ is highly sub-optimal. In practice we stop the proximal gradient algorithm when $c_j^{(t)} < \epsilon \approx 10^{-3}$ for all weights/features j.

21 Bias term for L1-regularized linear models

Typically an un-regularized intercept/bias term is included in the prediction function, $f(x_i) = \beta + w^T x_i$, so the cost becomes a function of that parameter as well:

$$C_\lambda(\beta, w) = \mathcal{L}(\beta, w) + \lambda||w||_1 \tag{51}$$

Exercise in class: derive the update rule and optimality criterion for this new model.

22 Max regularization/penalty parameter

$\lambda_{\max}$ is the smallest regularization/penalty parameter such that all optimal linear model weights are 0. To derive it we need to consider the optimality conditions of the weight vector:

$$0 \in \partial C_{\lambda_{\max}(0) = \nabla\mathcal{L}(0) + \lambda_{\max} \partial||0||_1} \tag{52}$$

which implies for all features j,

$$-\nabla_{w_j}\mathcal{L}(0) \in [-\lambda_{\max}, \lambda_{\max}]_{\Rightarrow |\nabla_{w_j}\mathcal{L}(0)| \leq \lambda_{\max}}. \tag{53}$$

The equations above imply that

$$\lambda_{\max = \max_j |\nabla_{w_j}\mathcal{L}(0)| = ||\nabla\mathcal{L}(0)||_\infty} \tag{54}$$