CSC 411 Lecture 3: Linear Models I

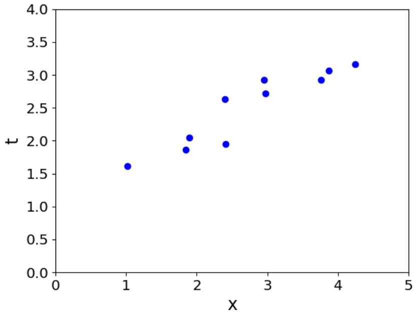
Roger Grosse

University of Toronto

Overview

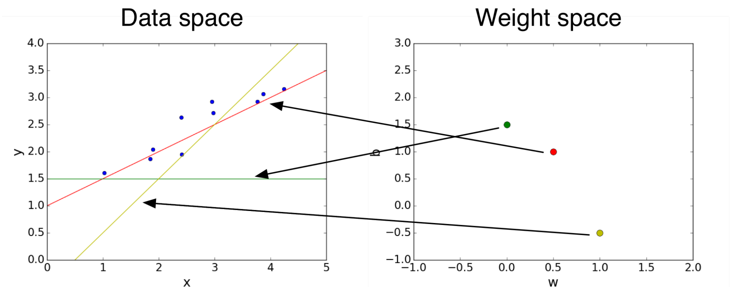
- So far, we've talked about *procedures* for learning.
  - KNN, decision trees, bagging
- For the remainder of this course, we'll take a more modular approach:
  - choose a model describing the relationships between variables of interest
  - define a loss function quantifying how bad is the fit to the 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 called inputs, and the $t^{(i)}$ are called 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 called hypotheses

Problem Setup

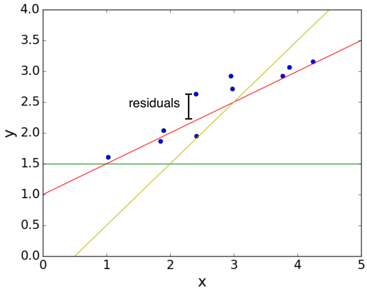
- Loss function: squared error (says how bad the fit is)

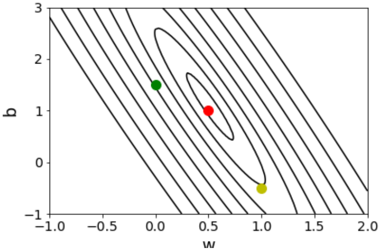
$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- Cost function: loss function averaged over all training examples

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

Problem Setup





Problem setup

- Suppose we have multiple inputs $x_1, \ldots, x_D$. This is referred to as multivariable regression.
- This is no different than the single input case, just harder to visualize.
- Linear model:

$$y = \sum_j w_j x_j + b$$

Vectorization

- Computing the prediction using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- For-loops in Python are slow, so we vectorize algorithms by expressing them in terms of vectors and matrices.

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

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

- This is simpler and 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
  - Matrix multiplication is very fast on a Graphics Processing Unit (GPU)

Vectorization

- We can take this a step further. Organize all the training examples into the design matrix $\mathbf{X}$ with one row per training example, and all the targets into the target vector $\mathbf{t}$.

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{array}{ccccc} & \text{one feature across all training examples} & & & \\ 8 & 0 & 3 & 0 & \\ 6 & -1 & 5 & 3 & \text{one training example (vector)} \\ 2 & 5 & -2 & 8 & \end{array}$$

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^\top \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

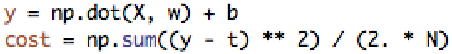
Vectorization

- Computing the squared error cost across the whole dataset:

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

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

- In Python:



```python
y = np.dot(X, w) + b
cost = np.sum((y - t) ** 2) / (2. * N)
```

Solving the optimization problem

- We defined a cost function. This is what we'd like to minimize.
- Recall from calculus class: minimum of a smooth function (if it exists) occurs at a critical point, i.e. point where the derivative is zero.
- Multivariate generalization: partial derivatives must be zero.
  - Finding a minimum by analytically setting the partial derivatives to zero is called direct solution.

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:

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

- Cost derivatives (average over data points):

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

Direct solution

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

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

- If $\partial \mathcal{J} / \partial w_j \neq 0$, you could reduce the cost by changing w_j.
- This turns out to give a system of linear equations, which we can solve efficiently. **Full derivation in the readings.**
- 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.
- Gradient descent is an iterative algorithm, which means we apply an update repeatedly until some criterion is met.
- We initialize the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the direction of steepest descent.

Gradient descent

- Observe:
  - if $\partial \mathcal{J} / \partial w_j > 0$, then slightly increasing w_j increases $\mathcal{J}$.
  - if $\partial \mathcal{J} / \partial w_j < 0$, then slightly increasing w_j decreases $\mathcal{J}$.
- The following update decreases the cost function, assuming small enough α:

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

- α is a learning rate. 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

Gradient descent

- This gets its name from the gradient:

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

  - This is the direction of fastest increase in $\mathcal{J}$.
- Update rule in vector form:

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

- Hence, gradient descent updates the weights in the direction of fastest *decrease*.

Gradient descent

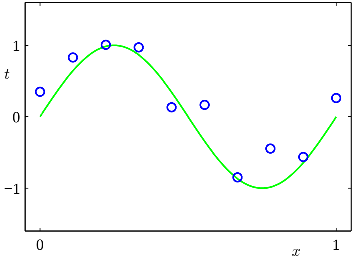
Visualization:
`http://www.cs.toronto.edu/~guerzhoy/321/lec/W01/linear_regression.pdf#page=21`

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

$$y = w_3x^3 + w_2x^2 + w_1x + w_0$$

- Do we need to derive a whole new algorithm?

Feature mappings

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

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

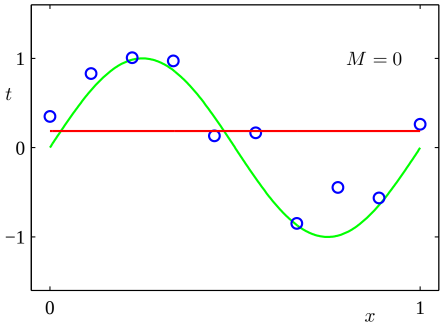
- Polynomial regression model:

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

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

Fitting polynomials

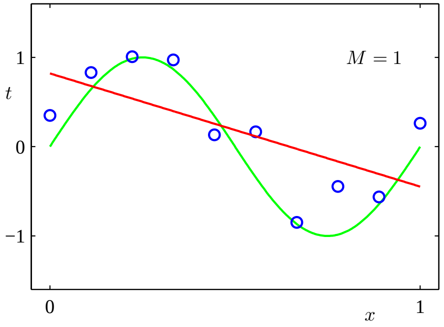
$$y = w_0$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

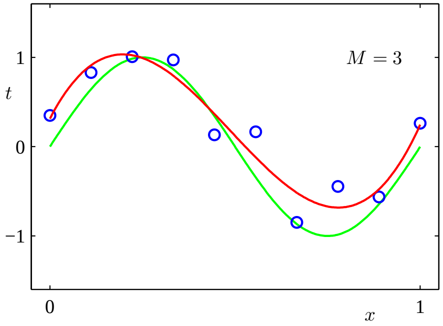
$$y = w_0 + w_1 x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

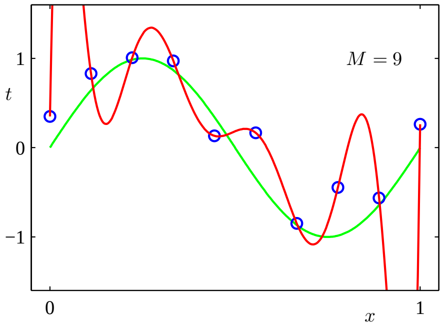
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

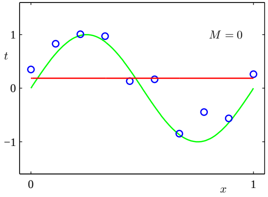
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$



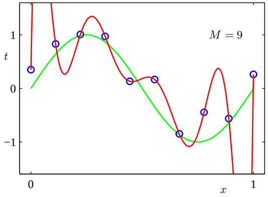
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

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

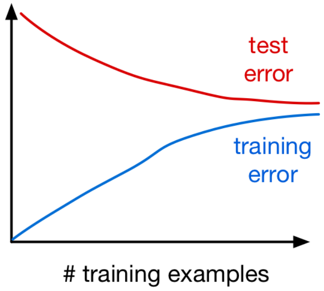


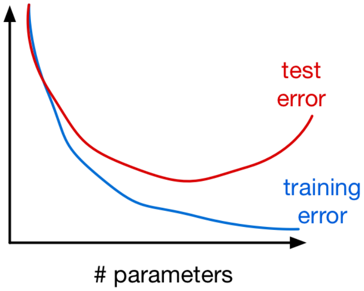
Overfitting : model is too complex — fits perfectly, does not generalize.



Generalization

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



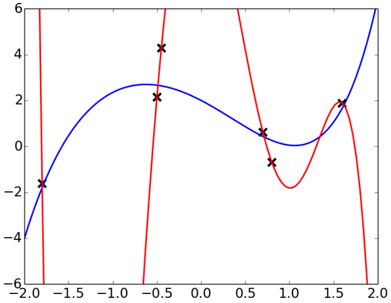


Regularization

- 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 Regularization

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

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

  - Note: to be pedantic, the L^2 norm is Euclidean distance, so we're really regularizing the *squared* L^2 norm.

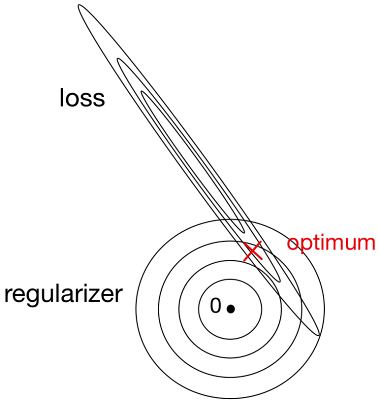
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

$$\mathcal{J}_{\mathrm{reg}} = \mathcal{J} + \lambda\mathcal{R} = \mathcal{J} + \frac{\lambda}{2}\sum_j w_j^2$$

- Here, λ is a hyperparameter that we can tune using a validation set.

L^2 Regularization

- The geometric picture:



L^2 Regularization

- Recall the gradient descent update:

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

- The gradient descent update of the regularized cost has an interesting interpretation as weight decay:

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

Conclusion

Linear regression exemplifies recurring themes of this course:

- choose a model and a loss function
- formulate an optimization problem
- solve the optimization problem using one of two strategies
  - direct solution (set derivatives to zero)
  - gradient descent
- vectorize the algorithm, i.e. represent in terms of linear algebra
- make a linear model more powerful using features
- improve the generalization by adding a regularizer

Linear Classification

Overview

- Classification: predicting a discrete-valued target
  - Binary classification: predicting a binary-valued target
- Examples
  - predict whether a patient has a disease, given the presence or absence of various symptoms
  - classify e-mails as spam or non-spam
  - predict whether a financial transaction is fraudulent

Overview

Binary linear classification

- **classification:** predict a discrete-valued target
- **binary:** predict a binary target $t \in \{0, 1\}$
    - Training examples with $t = 1$ are called positive examples, and training examples with $t = 0$ are called negative examples. Sorry.
- **linear:** model is a linear function of $\mathbf{x}$, followed by a threshold:

$$z = \mathbf{w}^T \mathbf{x} + b$$

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

Some simplifications

Eliminating the threshold

- We can assume WLOG that the threshold $r = 0$:

$$\mathbf{w}^T\mathbf{x} + b \geq r \quad \Longleftrightarrow \quad \mathbf{w}^T\mathbf{x} + \underbrace{b - r}_{\triangleq b'} \geq 0.$$

Eliminating the bias

- Add a dummy feature x_0 which always takes the value 1. The weight w_0 is equivalent to a bias.

Simplified model

$$z = \mathbf{w}^T\mathbf{x}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Examples

NOT

x_0	x_1	t
1	0	1
1	1	0

$$b > 0$$
$$b + w < 0$$

$$b = 1, \; w = -2$$

Examples

AND

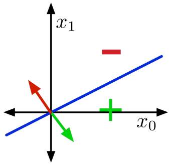
x_0	x_1	x_2	t
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$$\begin{aligned} b &< 0 \\ b + w_2 &< 0 \\ b + w_1 &< 0 \\ b + w_1 + w_2 &> 0 \end{aligned}$$

$$b = -1.5,\ w_1 = 1,\ w_2 = 1$$

The Geometric Picture

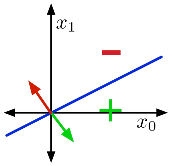
Input Space, or Data Space

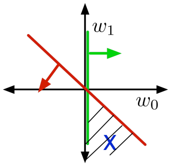


- Here we're visualizing the **NOT** example
- Training examples are points
- Hypotheses are half-spaces whose boundaries pass through the origin
- The boundary is the decision boundary
    - In 2-D, it's a line, but think of it as a hyperplane
- If the training examples can be separated by a linear decision rule, they are linearly separable.

The Geometric Picture

Weight Space





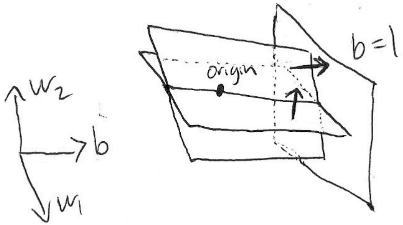
$w_0 > 0$

$w_0 + w_1 < 0$

- Hypotheses are points
- Training examples are half-spaces whose boundaries pass through the origin
- The region satisfying all the constraints is the feasible region; if this region is nonempty, the problem is feasible

The Geometric Picture

- The **AND** example requires three dimensions, including the dummy one.
- To visualize data space and weight space for a 3-D example, we can look at a 2-D slice:

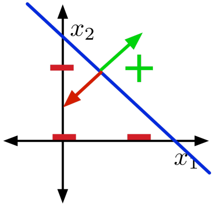


- The visualizations are similar, except that the decision boundaries and the constraints need not pass through the origin.

The Geometric Picture

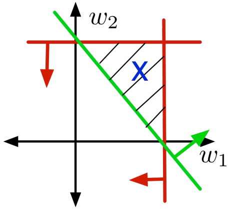
Visualizations of the **AND** example

Data Space



Slice for $x_0 = 1$

Weight Space

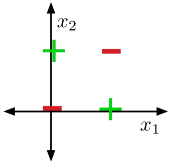


Slice for $w_0 = -1$

What happened to the fourth constraint?

The Geometric Picture

Some datasets are not linearly separable, e.g. **XOR**



Proof coming in a later lecture...

Overview

- **Recall: binary linear classifiers.** Targets $t \in \{0, 1\}$

$$z = \mathbf{w}^T \mathbf{x} + b$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

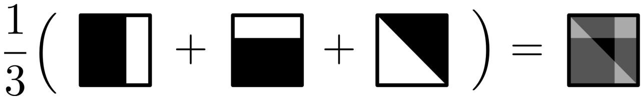
- What if we can't classify all the training examples correctly?
- Seemingly obvious loss function: 0-1 loss

$$\begin{aligned} \mathcal{L}_{0-1}(y, t) &= \begin{cases} 0 & \text{if } y = t \\ 1 & \text{if } y \neq t \end{cases} \\ &= \mathbb{1}_{y \neq t}. \end{aligned}$$

Attempt 1: 0-1 loss

- As always, the cost $\mathcal{J}$ is the average loss over training examples; for 0-1 loss, this is the error rate:

$$\mathcal{J} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{y^{(i)} \neq t^{(i)}}$$



Attempt 1: 0-1 loss

- Problem: how to optimize?
- Chain rule:

$$\frac{\partial \mathcal{L}_{0-1}}{\partial w_j} = \frac{\partial \mathcal{L}_{0-1}}{\partial z} \frac{\partial z}{\partial w_j}$$

- But $\partial \mathcal{L}_{0-1} / \partial z$ is zero everywhere it's defined!
  - $\partial \mathcal{L}_{0-1} / \partial w_j = 0$ means that changing the weights by a very small amount probably has no effect on the loss.
  - The gradient descent update is a no-op.

Attempt 2: Linear Regression

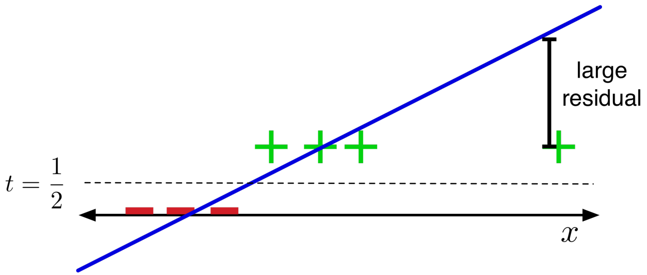
- Sometimes we can replace the loss function we care about with one which is easier to optimize. This is known as a surrogate loss function.
- We already know how to fit a linear regression model. Can we use this instead?

$$y = \mathbf{w}^\top \mathbf{x} + b$$
$$\mathcal{L}_{\mathrm{SE}}(y, t) = \frac{1}{2}(y - t)^2$$

- Doesn't matter that the targets are actually binary.
- Threshold predictions at $y = 1/2$.

Attempt 2: Linear Regression

The problem:



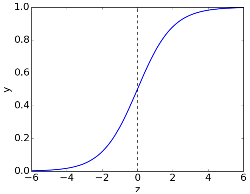
- The loss function hates when you make correct predictions with high confidence!
- If $t = 1$, it's more unhappy about $y = 10$ than $y = 0$.

Attempt 3: Logistic Activation Function

- There's obviously no reason to predict values outside [0, 1]. Let's squash y into this interval.

- The logistic function is a kind of sigmoidal, or S-shaped, function:

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



- A linear model with a logistic nonlinearity is known as log-linear:

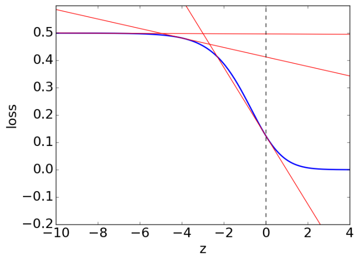
$$z = \mathbf{w}^\top \mathbf{x} + b$$
$$y = \sigma(z)$$
$$\mathcal{L}_{\mathrm{SE}}(y, t) = \frac{1}{2}(y - t)^2.$$

- Used in this way, σ is called an activation function, and z is called the logit.

Attempt 3: Logistic Activation Function

The problem:

(plot of $\mathcal{L}_{\mathrm{SE}}$ as a function of z)



$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w_j}$$

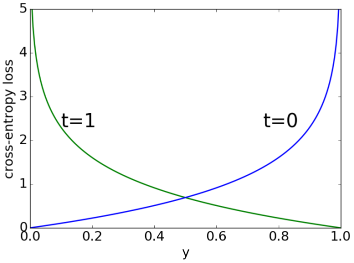
$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{L}}{\partial w_j}$$

- In gradient descent, a small gradient (in magnitude) implies a small step.
- If the prediction is really wrong, shouldn't you take a large step?
- This happens because the loss function saturates.

Logistic Regression

- Because $y \in [0, 1]$, we can interpret it as the estimated probability that $t = 1$.
- The pundits who were 99% confident Clinton would win were much more wrong than the ones who were only 90% confident.
- Cross-entropy loss captures this intuition:

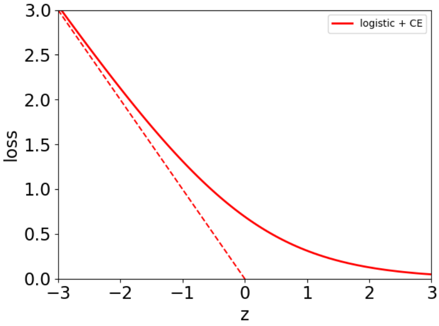
$$\begin{aligned}\mathcal{L}_{\mathrm{CE}}(y, t) &= \begin{cases} -\log y & \text{if } t = 1 \\ -\log(1 - y) & \text{if } t = 0 \end{cases} \\ &= -t \log y - (1 - t) \log(1 - y)\end{aligned}$$



Logistic Regression

Logistic Regression:

$$\begin{aligned} z &= \mathbf{w}^\top \mathbf{x} + b \\ y &= \sigma(z) \\ &= \frac{1}{1+e^{-z}} \\ \mathcal{L}_{\mathrm{CE}} &= -t \log y - (1-t)\log(1-y) \end{aligned}$$



[[gradient derivation in the notes]]

Logistic Regression

- Problem: what if $t = 1$ but you're really confident it's a negative example ($z \ll 0$)?
- If y is small enough, it may be numerically zero. This can cause very subtle and hard-to-find bugs.

$$\begin{aligned} y &= \sigma(z) & &\Rightarrow y \approx 0 \\ \mathcal{L}_{\mathrm{CE}} &= -t \log y - (1-t)\log(1-y) & &\Rightarrow \text{ computes } \log 0 \end{aligned}$$

- Instead, we combine the activation function and the loss into a single logistic-cross-entropy function.

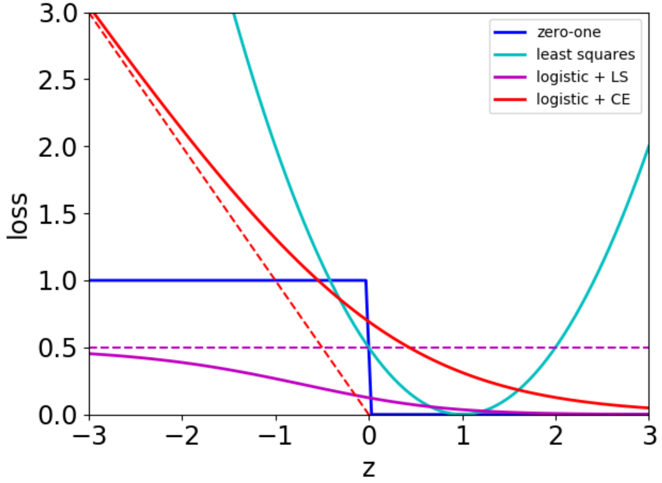
$$\mathcal{L}_{\mathrm{LCE}}(z, t) = \mathcal{L}_{\mathrm{CE}}(\sigma(z), t) = t \log(1 + e^{-z}) + (1-t)\log(1 + e^{z})$$

- Numerically stable computation:

```
E = t * np.logaddexp(0, -z) + (1-t) * np.logaddexp(0, z)
```

Logistic Regression

Comparison of loss functions:



Logistic Regression

Comparison of gradient descent updates:

- Linear regression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, \mathbf{x}^{(i)}$$

- Logistic regression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, \mathbf{x}^{(i)}$$

- Not a coincidence! These are both examples of generalized linear models, but that's beyond the scope of this course.