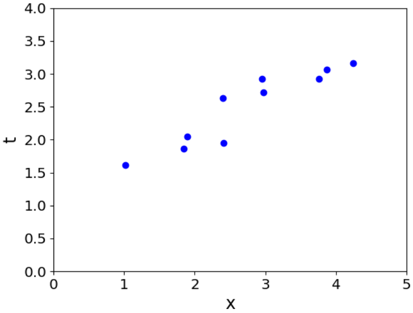
CSC421 Lecture 2: Linear Models

Roger Grosse and Jimmy Ba

Overview

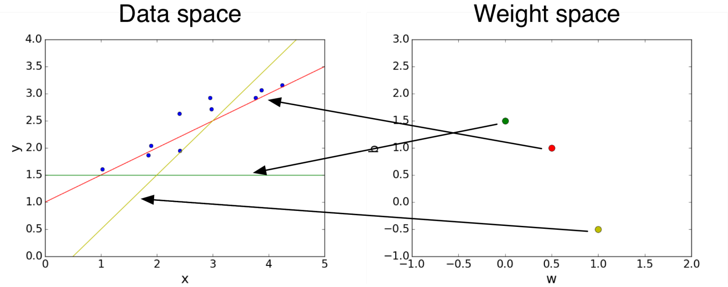
- Some canonical supervised learning problems:
  - Regression: predict a scalar-valued target (e.g. stock price)
  - Binary classification: predict a binary label (e.g. spam vs. non-spam email)
  - Multiway classification: predict a discrete label (e.g. object category, from a list)
- A simple approach is a linear model, where you decide based on a linear function of the input vector.
- This lecture reviews linear models, plus some other fundamental concepts (e.g. gradient descent, generalization)
- This lecture moves very quickly because it's all review. But there are detailed course readings if you need more of a refresher.

Problem Setup



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

Problem Setup



- **Model:** y is a linear function of x:

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

- y is the prediction
- $\mathbf{w}$ is the weight vector
- b is the bias
- $\mathbf{w}$ and b together are the parameters
- Settings of the parameters are called hypotheses

Problem Setup

- **Loss function:** squared error

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

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.

Problem Setup

- **Loss function:** squared error

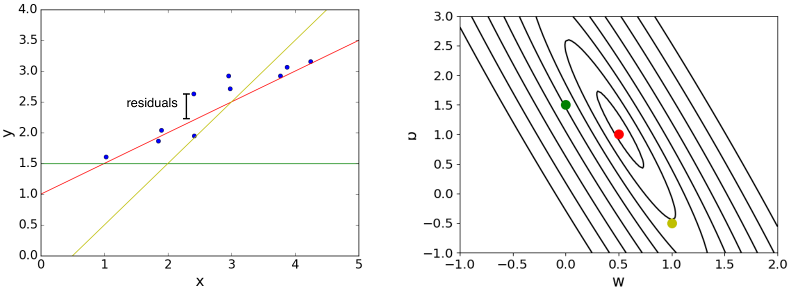
$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ 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

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

Problem Setup

Visualizing the contours of the cost function:



Vectorization

- We can organize all the training examples into a matrix **X** with one row per training example, and all the targets into a vector **t**.

one feature across all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training example (vector)

- Computing the predictions for the whole dataset:

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

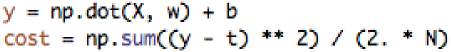
Vectorization

- 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:



```
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: the minimum of a smooth function (if it exists) occurs at a critical point, i.e. point where the partial derivatives are all 0.
- Two strategies for optimization:
  - Direct solution: derive a formula that sets the partial derivatives to 0. This works only in a handful of cases (e.g. linear regression).
  - Iterative methods (e.g. gradient descent): repeatedly apply an update rule which slightly improves the current solution. This is what we'll do throughout the course.

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:

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

- We will give a more precise statement of the Chain Rule next week. It's actually pretty complicated.
- Cost derivatives (average over data points):

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

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} - t^{(i)}$$

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.
- The gradient descent update decreases the cost function for 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 **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:

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

  - This is the direction of fastest increase in $\mathcal{J}$.

Gradient descent

- This gets its name from the gradient:

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

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

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

- 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 maps

- We can convert linear models into nonlinear models using feature maps.

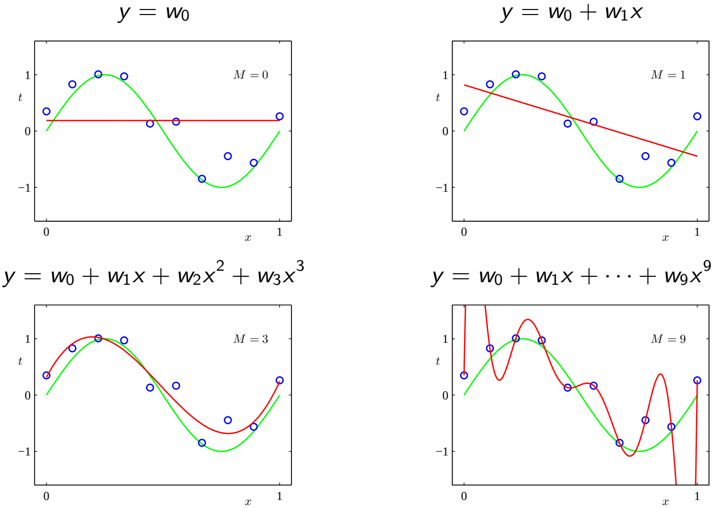
$$y = \mathbf{w}^\top \phi(\mathbf{x})$$

- E.g., if $\psi(x) = (1,\ x,\ \cdots, x^D)^\top$, then y is a polynomial in x. This model is known as polynomial regression:

$$y = w_0 + w_1 x + \cdots + w_D x^D$$

- This doesn't require changing the algorithm — just pretend $\psi(x)$ is the input vector.
- We don't need an expicit bias term, since it can be absorbed into ψ.
- Feature maps let us fit nonlinear models, but it can be hard to choose good features.
    - Before deep learning, most of the effort in building a practical machine learning system was feature engineering.

Feature maps



$y = w_0$

$y = w_0 + w_1 x$

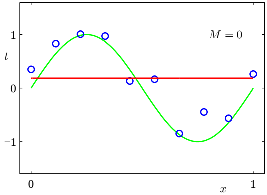
$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$

$y = w_0 + w_1 x + \cdots + w_9 x^9$

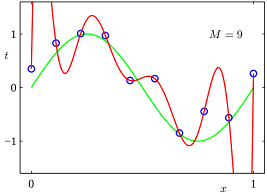
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

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

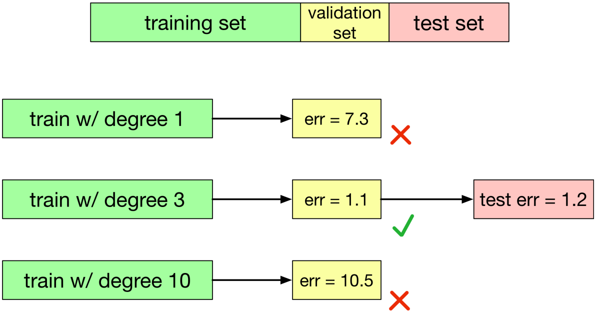


Overfitting : The model is too complex - fits perfectly, does not generalize.



Generalization

- We would like our models to generalize to data they haven't seen before
- The degree of the polynomial is an example of a hyperparameter, something we can't include in the training procedure itself
- We can tune hyperparameters using a validation set:



training set | validation set | test set

train w/ degree 1 → err = 7.3 ✗

train w/ degree 3 → err = 1.1 ✓ → test err = 1.2

train w/ degree 10 → err = 10.5 ✗

Classification

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 $\mathbf{x}$, thresholded at zero:

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

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

Logistic Regression

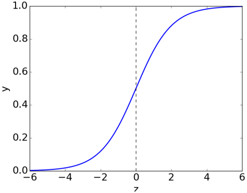
- We can't optimize classification accuracy directly with gradient descent because it's discontinuous.
- Instead, we typically define a continuous surrogate loss function which is easier to optimize. Logistic regression is a canonical example of this, in the context of classification.
- The model outputs a continuous value $y \in [0, 1]$, which you can think of as the probability of the example being positive.

Logistic Regression

- 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:

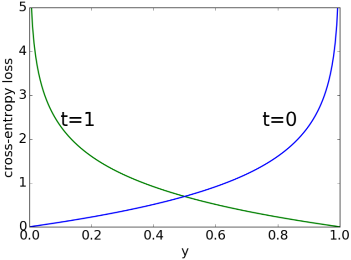
$$z = \mathbf{w}^\top \mathbf{x} + b$$
$$y = \sigma(z)$$

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

Logistic Regression

- Because $y \in [0, 1]$, we can interpret it as the estimated probability that $t = 1$.
- Being 99% confident of the wrong answer is much worse than being 90% confident of the wrong answer. Cross-entropy loss captures this intuition:

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

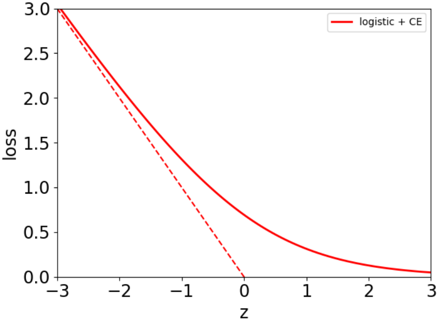


- Aside: why does it make sense to think of y as a probability? Because cross-entropy loss is a proper scoring rule, which means the optimal y is the true probability.

Logistic Regression

- Logistic regression combines the logistic activation function with cross-entropy loss.

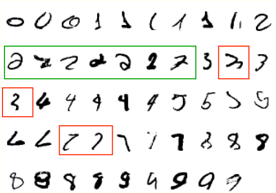
$$
\begin{aligned}
z &= \mathbf{w}^\top \mathbf{x} + b \\
y &= \sigma(z) \\
&= \frac{1}{1+e^{-z}} \\
\mathcal{L}_{\mathrm{CE}} &= -t \log y - (1-t)\log(1-y)
\end{aligned}
$$

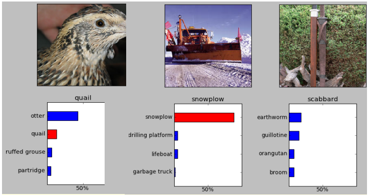


- Interestingly, the loss asymptotes to a linear function of the logit z.
- Full derivation in the readings.

Multiclass Classification

- What about classification tasks with more than two categories?





Multiclass Classification

- Targets form a discrete set $\{1, \ldots, K\}$.
- It's often more convenient to represent them as one-hot vectors, or a one-of-K encoding:

$$\mathbf{t} = \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{\text{entry } k \text{ is } 1}$$

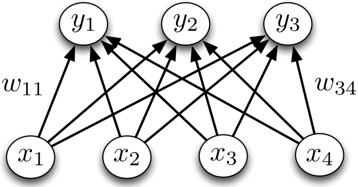
Multiclass Classification

- Now there are D input dimensions and K output dimensions, so we need $K \times D$ weights, which we arrange as a weight matrix $\mathbf{W}$.
- Also, we have a K-dimensional vector $\mathbf{b}$ of biases.
- Linear predictions:

$$z_k = \sum_j w_{kj} x_j + b_k$$

- Vectorized:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



Multiclass Classification

- A natural activation function to use is the softmax function, a multivariable generalization of the logistic function:

$$y_k = \mathrm{softmax}(z_1, \ldots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

- The inputs z_k are called the logits.
- Properties:
  - Outputs are positive and sum to 1 (so they can be interpreted as probabilities)
  - If one of the z_k's is much larger than the others, $\mathrm{softmax}(\mathbf{z})$ is approximately the argmax. (So really it's more like "soft-argmax".)
  - **Exercise:** how does the case of $K = 2$ relate to the logistic function?
- Note: sometimes $\sigma(\mathbf{z})$ is used to denote the softmax function; in this class, it will denote the logistic function applied elementwise.

Multiclass Classification

- If a model outputs a vector of class probabilities, we can use cross-entropy as the loss function:

$$\begin{aligned}\mathcal{L}_{\mathrm{CE}}(\mathbf{y}, \mathbf{t}) &= -\sum_{k=1}^{K} t_k \log y_k \\ &= -\mathbf{t}^\top (\log \mathbf{y}),\end{aligned}$$

  where the log is applied elementwise.

- Just like with logistic regression, we typically combine the softmax and cross-entropy into a softmax-cross-entropy function.

Multiclass Classification

- Softmax regression, also called multiclass logistic regression:

$$\begin{aligned} \mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} &= \mathrm{softmax}(\mathbf{z}) \\ \mathcal{L}_{\mathrm{CE}} &= -\mathbf{t}^{\top}(\log \mathbf{y}) \end{aligned}$$

- It's possible to show the gradient descent updates have a convenient form:

$$\frac{\partial \mathcal{L}_{\mathrm{CE}}}{\partial \mathbf{z}} = \mathbf{y} - \mathbf{t}$$