CSCI-567: Machine Learning

Prof. Victor Adamchik

U of Southern California

July 8, 2020

Your model is only as good as your data.

Outline

Outline

Regression

Predicting a continuous outcome variable using past observations

Key difference from classification

- continuous vs discrete
- measure *prediction errors* differently.
- lead to quite different learning algorithms.

Linear Regression: regression with *linear models:* $f(\boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{w}$

Linear Least Squares Regression

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} (\boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{w} - y_n)^2 = \operatorname*{argmin}_{\boldsymbol{w}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2$$

Three approaches to find the minimum:

- Closed Form (setting gradient to zero) $\boldsymbol{w}^* = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$
- Gradient Descent (GD)
- Stochastic Gradient Descent (SGD)

Gradient

The gradient vector ∇f points in the direction of greatest rate of increase of f at a given point.

The the rates of change of f in all directions is given by

$$\nabla f \cdot u = \|\nabla f\| \|u\| \cos\alpha$$

Hence, the direction of *greatest decrease* of f is the direction opposite to the gradient vector, when $\alpha = \pi$

We will minimize $RSS(w)$ using a gradient descent method.

Gradient Descent (GD)

Goal: minimize $f(w)$

Consider the definition

$$f'(w) = \lim_{\Delta x \to 0} \frac{f(w + \Delta x) - f(w)}{\Delta x}$$

Our gradient is an estimation of the derivative

$$\nabla f(w) = \frac{f(w + \Delta x) - f(w)}{\Delta x}$$

This gives the first-order approximation:

$$f(w + \Delta x) = f(w) + \Delta x \nabla f(w)$$

Note we need to move in its *opposite* direction to climb down the function.

Algorithm: Gradient Descent

Goal: minimize $F(\boldsymbol{w})$

Algorithm: move a bit in the *negative gradient direction*

initialize $\boldsymbol{w}^{(0)}$
while not converged do

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \lambda \nabla F(\boldsymbol{w}^{(t)})$$

where $\lambda > 0$ is called *step size or learning rate*

- *in theory λ should be set in terms of some parameters of F*
- *in practice we just try several small values*
- *there are many possible ways to reduce the learning rate λ during training (for example, exponential decay.)*

An example

Example: $F(\boldsymbol{w}) = 0.5(w_1^2 - w_2)^2 + 0.5(w_1 - 1)^2$.
Gradient is

$$\frac{\partial F}{\partial w_1} = 2(w_1^2 - w_2)w_1 + w_1 - 1 \qquad \frac{\partial F}{\partial w_2} = -(w_1^2 - w_2)$$

GD:

- Initialize $w_1^{(0)}$ and $w_2^{(0)}$ (to be 0 or *randomly*), $t = 0$
- do

$$w_1^{(t+1)} \leftarrow w_1^{(t)} - \lambda \left[2(w_1^{(t)2} - w_2^{(t)})w_1^{(t)} + w_1^{(t)} - 1 \right]$$
$$w_2^{(t+1)} \leftarrow w_2^{(t)} - \lambda \left[-(w_1^{(t)2} - w_2^{(t)}) \right]$$
$$t \leftarrow t + 1$$

- until $F(\boldsymbol{w}^{(t)})$ **does not change much**

Why does GD work ?

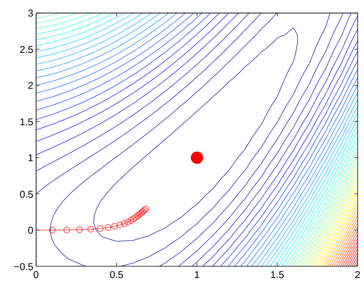
Using the first-order approximation

$$f(w + \Delta x) = f(w) + \Delta x \nabla f(w)$$

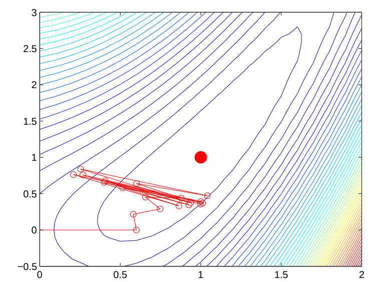
we move a bit in the negative gradient direction $\Delta x = -\lambda \nabla f(w)$

This ensures

$$f(w - \lambda \nabla f(w)) = f(w) - \lambda (\nabla f(w))^2 \leq f(w)$$



reasonable λ decreases function value



but large λ is unstable

Applying GD to Linear Regression

In the previous discussion, we have computed:

$$\frac{1}{2} \nabla RSS(\boldsymbol{w}) = \boldsymbol{X}^{\mathrm{T}} \boldsymbol{X} \boldsymbol{w} - \boldsymbol{X}^{\mathrm{T}} \boldsymbol{y} = \boldsymbol{X}^{\mathrm{T}} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$$

$$\frac{1}{2} \frac{\partial RSS}{w_j} = \sum_{n=1}^{N} x_{nj} \sum_{i=0}^{D} (x_{ni} w_i - y_n) = \sum_{n=1}^{N} x_{nj} \left(f(\boldsymbol{x}_n) - y_n \right)$$

GD update:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \lambda \boldsymbol{X}^{\mathrm{T}} \left(\boldsymbol{X} \boldsymbol{w}^{(t)} - \boldsymbol{y} \right)$$

For a single weight,

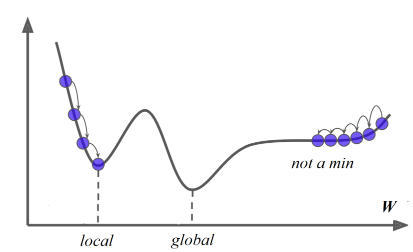
$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \sum_n x_{nj} \left(f(\boldsymbol{x}_n) - y_n \right)$$

The algorithm uses all training points on each iteration. The algorithm is called *batch gradient descent*.

GD challenges

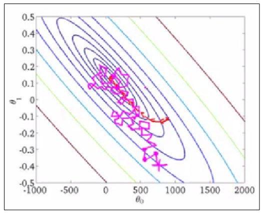
There two main challenges with GD:

- it may converge to a local minimum.
- it may not find a minimum at all. "vanishing gradient".



Stochastic Gradient Descent (SGD)

- GD: move a bit in the negative gradient direction.
- SGD: move a bit in a *noisy* negative gradient direction.
- In SGD, we use one training sample at each iteration.
- Need to randomly shuffle the training examples before calculating it.
- SGD is widely used for larger dataset and can be trained in parallel.



SGD for Linear Regression

Algorithm:

initialize $\boldsymbol{w}^{(0)}$
for each training sample n**:**
for each weight j**:**

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda\, x_{nj}\left(f(\boldsymbol{x}_n) - y_n\right)$$

The term "stochastic" comes from the fact that the gradient based on a single training sample.

SGD makes progress with each training example as it looks at.

Key point: it could be *much faster to obtain a stochastic gradient*!

GD versus SGD

In GD we calculate the gradient for all training points

In SGD we calculate the gradient for one sample (or a small batch, called a *mini-batch* SGD) of training data

In SGD you might not be taking the most optimal route to get to the solution.

SGD may work for non-convex functions

In SGD you need to go through the training set several times (this is called an *epoch*).

You must specify the batch size (a typical size is 256) and number of epochs (a hyperparameter) for a learning algorithm.

Example

Assume we have a set of 100 samples and you choose a batch size of 5 and 200 epochs.

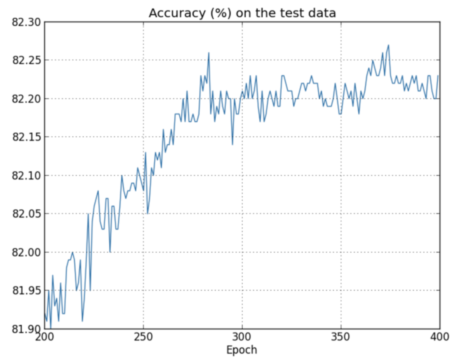
This means that the dataset will be divided into 20 batches. The model weights will be updated after each batch of five samples.

This also means that one epoch consists of 20 batches.

The model will pass through the whole training set 200 times, that is a total of 4000 batches.

Epoch and overfitting

This shows how test accuracy is changing due to the number of epochs:



GD pseudocode

In the algorithm we repeatedly make small steps downward on a surface defined by a loss function $f(params)$.

Gradient Descent

```
while True:
    loss = f(params)
    d_loss_wrt_params = ...                  (#compute the gradient)
    params = params - learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```


SGD pseudocode

SGD works GD but proceeds more quickly by estimating the gradient from just a few examples

Stochastic Gradient Descent

```
for (x_i, y_i) in training set:
    loss = f(params, x_i, y_i)
    d_loss_wrt_params = ...                  (#compute the gradient)
    params = params - learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```


Minibatch SGD pseudocode

Minibatch SGD works identically to SGD, except that we use more than one training example to make each estimate of the gradient.

Minibatch SGD

```
for (x_batch, y_batch) in training batch:
    loss = f(params, x_batch, y_batch)
    d_loss_wrt_params = ...                  (#compute the gradient)
    params = params - learning_rate * d_loss_wrt_params
    if <stopping condition is met>:
        return params
```

SGD optimization algorithms

A few challenges with gradient descent:

- choosing a proper learning rate
- the same learning rate applies to all parameter updates
- getting trapped in the numerous suboptimal local minima.

There are many variants of SGD: momentum, Nesterov accelerated gradient, Adagrad, Adadelta, Adam, Nadam, $\cdots$

SGD with momentum

The main idea behind it is to use the **moving average** of the gradient instead of using only the current real value of the gradient. This prevents from getting stuck in local minimum.

Initialize $\boldsymbol{w}_0$ and **velocity** $\boldsymbol{v} = \mathbf{0}$

For $t = 1, 2, \ldots$

- form a stochastic gradient $\boldsymbol{g}_t$.
- update velocity $\boldsymbol{v}_t \leftarrow \alpha \boldsymbol{v}_{t-1} - \lambda \boldsymbol{g}_t$ where $\alpha \in (0, 1)$.
- update weight $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \boldsymbol{v}_t$ by adding a fraction of the update vector of the past time step.

Updates for first few rounds:

- $\boldsymbol{w}_1 = \boldsymbol{w}_0 - \lambda \boldsymbol{g}_1$
- $\boldsymbol{w}_2 = \boldsymbol{w}_1 - \alpha\lambda \boldsymbol{g}_1 - \lambda \boldsymbol{g}_2$
- $\boldsymbol{w}_3 = \boldsymbol{w}_2 - \alpha^2\lambda \boldsymbol{g}_1 - \alpha\lambda \boldsymbol{g}_2 - \lambda \boldsymbol{g}_3$
- $\cdots$

Newton's method

Newton's method is an extension of steepest descent, where the second-order term in the Taylor series is used.

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

Let us minimize the right hand side:

$$f'(x_0) + f''(x_0)(x - x_0) = 0 \quad \text{or} \quad x = x_0 - \frac{f'(x_0)}{f''(x_0)}$$

We will literate this procedure

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Deriving Newton method

This could be generalized for functions f of several variables:

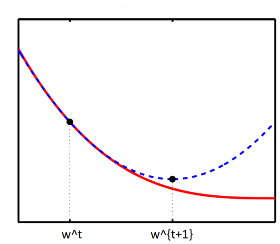
$$x_{n+1} = x_n - \boldsymbol{H}^{-1}(x_n)\, \nabla f(x_n)$$

where $\boldsymbol{H}$ is the Hessian

$$H_{ij} = \frac{\partial^2 F(\boldsymbol{x})}{\partial x_i \partial x_j}$$

Therefore, for convex F (so H_t is *positive semidefinite*) we obtain **Newton method**:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \boldsymbol{H}_t^{-1} \nabla F(\boldsymbol{w}^{(t)})$$



Comparing GD and Newton

$$w^{(t+1)} \leftarrow w^{(t)} - \lambda \nabla F(w^{(t)}) \qquad \text{(GD)}$$
$$w^{(t+1)} \leftarrow w^{(t)} - H_t^{-1} \nabla F(w^{(t)}) \qquad \text{(Newton)}$$

Both are iterative optimization procedures, but Newton method

- has no learning rate λ (*so no tuning needed!*)
- converges *super fast* in terms of #iterations needed
- requires **second-order** information

Outline

Predicting probability

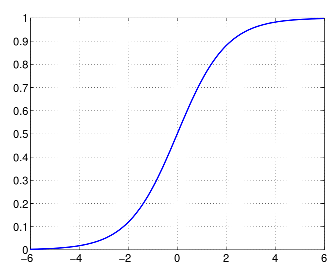
Instead of predicting a discrete label, can we *predict the probability of each label?* i.e. regress the probabilities

One way: **sigmoid function + linear model**

$$\mathbb{P}(y = +1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x})$$

where σ is the sigmoid function:

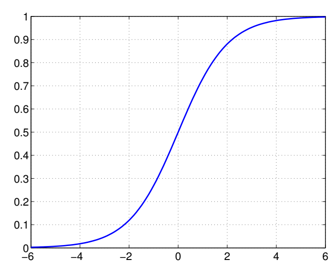
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Properties

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

- between 0 and 1 (good as probability)
- $\sigma(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) \geq 0.5 \Leftrightarrow \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x} \geq 0$, consistent with predicting the label with $\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x})$
- larger $\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x} \Rightarrow$ larger $\sigma(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) \Rightarrow$ higher *confidence* in label 1
- $\sigma(z) + \sigma(-z) = 1$ for all z



The probability of label -1 is naturally

$$1 - \mathbb{P}(y = +1 \mid \boldsymbol{x}; \boldsymbol{w}) = 1 - \sigma(\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) = \sigma(-\boldsymbol{w}^{\mathrm{T}} \boldsymbol{x})$$

and thus

$$\mathbb{P}(y \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}) = \frac{1}{1 + e^{-y \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}}}$$

How to regress with discrete labels?

What we observe are labels, not probabilities.

Take a **probabilistic view**

- assume data is generated in this way by some $\boldsymbol{w}$
- perform Maximum Likelihood Estimation (MLE)

Specifically, what is the probability of seeing label $y_1, \cdots, y_n$ given $x_1, \cdots, x_n$, as a function of some $\boldsymbol{w}$?

$$P(\boldsymbol{w}) = \prod_{n=1}^{N} \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w})$$

MLE: find $\boldsymbol{w}^*$ that **maximizes the probability** $P(\boldsymbol{w})$

The MLE solution

$$\begin{aligned}
\boldsymbol{w}^* &= \operatorname*{argmax}_{\boldsymbol{w}} P(\boldsymbol{w}) = \operatorname*{argmax}_{\boldsymbol{w}} \prod_{n=1}^{N} \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{w}} \sum_{n=1}^{N} \ln \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w}) = \operatorname*{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} - \ln \mathbb{P}(y_n \mid \boldsymbol{x_n}; \boldsymbol{w}) \\
&= \operatorname*{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} \ln(1 + e^{-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x_n}})
\end{aligned}$$

i.e. *minimizing logistic loss is exactly doing MLE for the sigmoid model!*

Let's apply SGD again

$$\begin{aligned}
\boldsymbol{w} &\leftarrow \boldsymbol{w} - \lambda \nabla F(\boldsymbol{w}) \\
&= \boldsymbol{w} - \lambda \nabla_{\boldsymbol{w}} \ln(1 + e^{-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x_n}}) \\
&= \boldsymbol{w} - \lambda \left(\frac{\partial \ln(1 + e^{-z})(z)}{\partial z} \Big|_{z = y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n} \right) y_n \boldsymbol{x}_n \\
&= \boldsymbol{w} - \lambda \left(\frac{-e^{-z}}{1 + e^{-z}} \Big|_{z = y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n} \right) y_n \boldsymbol{x}_n \\
&= \boldsymbol{w} + \lambda\, \sigma(-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) y_n \boldsymbol{x}_n \\
&= \boldsymbol{w} + \lambda\, \mathbb{P}(-y_n \mid \boldsymbol{x}_n; \boldsymbol{w}) y_n \boldsymbol{x}_n
\end{aligned}$$

Applying Newton to logistic loss

In the previous slide we have computed the gradient:

$$\nabla_{\boldsymbol{w}} \ln(1 + e^{-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x_n}}) = -\sigma(-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) y_n \boldsymbol{x}_n$$

Now we compute a second derivative:

$$\begin{aligned}
\nabla^2_{\boldsymbol{w}} \ln(1 + e^{-y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x_n}}) &= \left(\frac{\partial \sigma(z)}{\partial z} \Big|_{z = -y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n} \right) y_n^2 \boldsymbol{x}_n \boldsymbol{x}_n^{\mathrm{T}} \\
&= \left(\frac{e^{-z}}{(1 + e^{-z})^2} \Big|_{z = -y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n} \right) \boldsymbol{x}_n \boldsymbol{x}_n^{\mathrm{T}} \\
&= \sigma(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n) \left(1 - \sigma(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)\right) \boldsymbol{x}_n \boldsymbol{x}_n^{\mathrm{T}}
\end{aligned}$$

Outline

Classification

Recall the setup:

- input (feature vector): $\boldsymbol{x} \in \mathbb{R}^{\mathsf{D}}$
- output (label): $y \in [\mathsf{C}] = \{1, 2, \cdots, \mathsf{C}\}$
- goal: learn a mapping $f : \mathbb{R}^{\mathsf{D}} \to [\mathsf{C}]$

Examples:

- recognizing digits ($\mathsf{C} = 10$) or letters ($\mathsf{C} = 26$ or 52)
- predicting weather: sunny, cloudy, rainy, etc
- predicting image category: ImageNet dataset ($\mathsf{C} \approx 20K$)

Nearest Neighbor Classifier naturally works for arbitrary C.

Linear models: from binary to multiclass

What should a linear model look like for multiclass tasks?

Note: a linear model for binary tasks (switching from $\{-1, +1\}$ to $\{1, 2\}$)

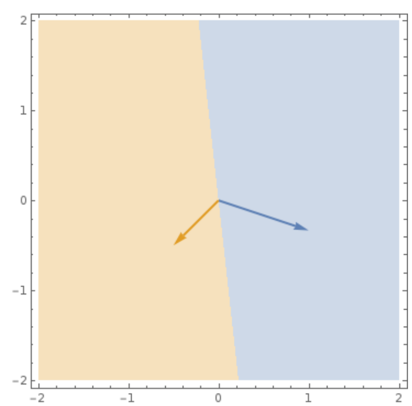
$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0 \\ 2 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0 \end{cases}$$

By setting $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$, it can be written as

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \geq \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} \\ 2 & \text{if } \boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x} > \boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x} \end{cases} = \underset{k \in \{1,2\}}{\operatorname{argmax}}\, \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$$

for any $\boldsymbol{w}_1, \boldsymbol{w}_2$.
Think of $\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}$ as **a score for class k**.

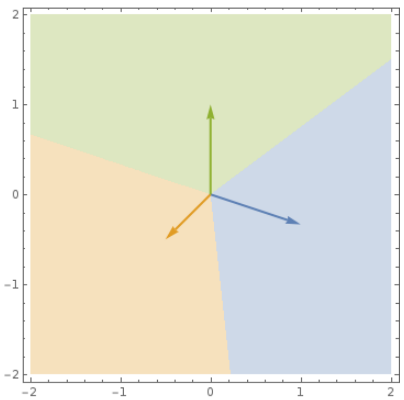
Linear models: from binary to multiclass



$\boldsymbol{w} = (\frac{3}{2}, \frac{1}{6}) = \boldsymbol{w}_1 - \boldsymbol{w}_2$
$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$

- Blue class:
  $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \geq 0\}$
  $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Orange class:
  $\{\boldsymbol{x} : \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} < 0\}$
  $\{\boldsymbol{x} : 2 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$

Linear models: from binary to multiclass



$\boldsymbol{w}_1 = (1, -\frac{1}{3})$
$\boldsymbol{w}_2 = (-\frac{1}{2}, -\frac{1}{2})$
$\boldsymbol{w}_3 = (0, 1)$

- Blue class: $\{\boldsymbol{x} : 1 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Orange class: $\{\boldsymbol{x} : 2 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$
- Green class: $\{\boldsymbol{x} : 3 = \operatorname{argmax}_k \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}\}$

Linear models for multiclass classification

$$\begin{aligned}\mathcal{F} &= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k\in[\mathsf{C}]} \boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{D}} \right\} \\ &= \left\{ f(\boldsymbol{x}) = \operatorname*{argmax}_{k\in[\mathsf{C}]} (\boldsymbol{W}\boldsymbol{x})_k \mid \boldsymbol{W} \in \mathbb{R}^{\mathsf{C}\times\mathsf{D}} \right\}\end{aligned}$$

Next, let us focus on the **logistic loss**.

Multinomial logistic regression: a probabilistic view

Observe: for binary logistic regression, with $\boldsymbol{w} = \boldsymbol{w}_1 - \boldsymbol{w}_2$:

$$\mathbb{P}(y = 1 \mid \boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}}} = \frac{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}}}{e^{\boldsymbol{w}_1^{\mathrm{T}}\boldsymbol{x}} + e^{\boldsymbol{w}_2^{\mathrm{T}}\boldsymbol{x}}}$$

Naturally, for class $y = y_n$

$$\mathbb{P}(y = y_n \mid \boldsymbol{x}; \boldsymbol{W}) = \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}}\boldsymbol{x}}}{\sum_{k\in[\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}}}$$

This is called the *softmax function*.

Applying MLE again

Maximize probability of seeing labels $y_1, \ldots, y_{\mathsf{N}}$ given $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\mathsf{N}}$

$$P(\boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) = \prod_{n=1}^{\mathsf{N}} \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}}\boldsymbol{x}_n}}{\sum_{k\in[\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}_n}}$$

By taking **negative log**, this is equivalent to minimizing

$$F(\boldsymbol{W}) = \sum_{n=1}^{\mathsf{N}} \ln\left(\frac{\sum_{k\in[\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}}\boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}}\boldsymbol{x}_n}}\right) = \sum_{n=1}^{\mathsf{N}} \ln\left(1 + \sum_{k\neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}}\boldsymbol{x}_n}\right)$$

This is the *multiclass logistic loss*, a.k.a *cross-entropy loss*.

When $\mathsf{C} = 2$, this is the same as binary logistic loss.

Optimization

Apply **SGD**: what is the gradient of

$$g(\boldsymbol{W}) = \ln\left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)?$$

This is a $\mathsf{C} \times \mathsf{D}$ matrix.

Take the derivative wrt $\boldsymbol{w}_j \neq \boldsymbol{w}_{y_n}$:

$$\begin{aligned}
\nabla_{\boldsymbol{w}_j} g(\boldsymbol{W}) &= \frac{e^{(\boldsymbol{w}_j - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}}{1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} \\
&= \frac{e^{\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{x}_n}}{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n} + \sum_{k \neq y_n} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} \\
&= \frac{e^{\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} = \mathbb{P}(j \mid \boldsymbol{x}_n; \boldsymbol{W}) \boldsymbol{x}_n^{\mathrm{T}}
\end{aligned}$$

Optimization

Apply **SGD**

$$g(\boldsymbol{W}) = \ln\left(1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}\right)$$

Take the derivative wrt $\boldsymbol{w}_{y_n}$.

$$\begin{aligned}
\nabla_{\boldsymbol{w}_{y_n}} g(\boldsymbol{W}) &= -\frac{\sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}}{1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}} \boldsymbol{x}_n^{\mathrm{T}} \\
&= \left(-1 + \frac{1}{1 + \sum_{k \neq y_n} e^{(\boldsymbol{w}_k - \boldsymbol{w}_{y_n})^{\mathrm{T}} \boldsymbol{x}_n}}\right) \boldsymbol{x}_n^{\mathrm{T}} \\
&= \left(-1 + \frac{e^{\boldsymbol{w}_{y_n}^{\mathrm{T}} \boldsymbol{x}_n}}{\sum_{k \in [\mathsf{C}]} e^{\boldsymbol{w}_k^{\mathrm{T}} \boldsymbol{x}_n}}\right) \boldsymbol{x}_n^{\mathrm{T}} = \left(-1 + \mathbb{P}(y_n \mid \boldsymbol{x}_n; \boldsymbol{W})\right) \boldsymbol{x}_n^{\mathrm{T}}
\end{aligned}$$

SGD for multinomial logistic regression

Initialize $\boldsymbol{W} = \boldsymbol{0}$ (or randomly). Repeat:

1. pick $n \in [\mathsf{N}]$ uniformly at random
2. update the parameters

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \lambda \begin{pmatrix} \mathbb{P}(y = 1 \mid \boldsymbol{x}_n; \boldsymbol{W}) \\ \vdots \\ \mathbb{P}(y = y_n \mid \boldsymbol{x}_n; \boldsymbol{W}) - 1 \\ \vdots \\ \mathbb{P}(y = \mathsf{C} \mid \boldsymbol{x}_n; \boldsymbol{W}) \end{pmatrix} \boldsymbol{x}_n^{\mathrm{T}}$$

Think about why the algorithm makes sense.

Consider $\mathbb{P}(y = y_n) \to 1$ and $\mathbb{P}(y = y_n) \to 0$...

Reduce multiclass to binary

Is there an *even more general and simpler approach* to derive multiclass classification algorithms?

Given a binary classification algorithm (*any one*, not just linear methods), can we turn it to a multiclass algorithm, *in a black-box manner*?

Yes, there are in fact many ways to do it.

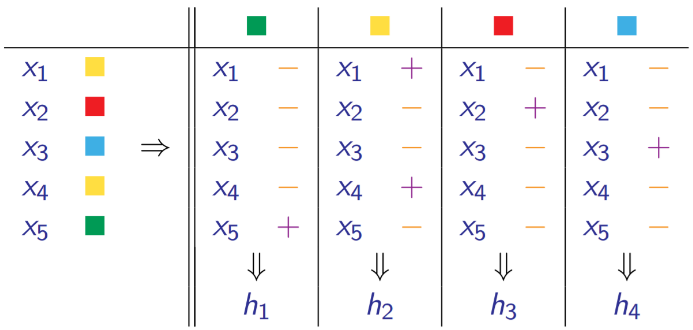
- **one-versus-all** (one-versus-rest, one-against-all, etc)
- **one-versus-one** (all-versus-all, etc)
- **Error-Correcting Output Codes** (ECOC)
- **tree-based reduction**

One-versus-all (OvA)

Idea: make C binary classifiers.

Training: for each class $k \in [\mathrm{C}]$,

- relabel each example with class k as $+1$, and all others as -1
- train a binary classifier h_k using this new dataset (what size?)



One-versus-all (OvA)

Prediction: for a new example $\boldsymbol{x}$

- ask each h_k: **does this belong to class k?** (i.e. $h_k(\boldsymbol{x})$)
- could be several h_k s.t. $h_k(\boldsymbol{x}) = +1$.
- randomly pick one

OvA becomes inefficient as the number of classes rises.

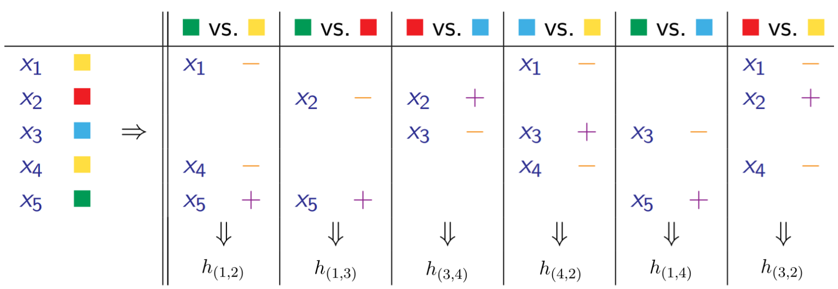
It's possible to create a significantly more efficient OvA model with a deep neural network.

One-versus-one (OvO)

Idea: make $\binom{\mathrm{C}}{2}$ binary classifiers.

Training: for each pair (k, k'),

- relabel each example with class k as $+1$ and with class k' as -1
- *discard all other examples*
- train a binary classifier $h_{(k,k')}$ using this new dataset (what size?)



One-versus-one (OvO)

Prediction: for a new example $\boldsymbol{x}$

- ask each classifier $h_{(k,k')}$ to **vote for either class k or k'**
- predict the class with the most votes (break tie in some way)

More robust than one-versus-all, but *slower* in prediction.

Error-correcting output codes (ECOC)

Idea: based on a code $\boldsymbol{M} \in \{-1, +1\}^{\mathsf{C}\times\mathsf{L}}$, train L binary classifiers to learn "**is bit b on or off**".

Training: for each bit $b \in [\mathsf{L}]$

- relabel example x_n as $M_{y_n,b}$
- train a binary classifier h_b on each column of $\boldsymbol{M}$.

M	1	2	3	4	5
🟩	+	−	+	−	+
🟨	−	−	+	+	+
🟥	+	+	−	−	−
🟦	+	+	+	+	−

			1		2		3		4		5	
x_1	🟨		x_1	−	x_1	−	x_1	+	x_1	+	x_1	+
x_2	🟥		x_2	+	x_2	+	x_2	−	x_2	−	x_2	−
x_3	🟦	$\Rightarrow$	x_3	+	x_3	+	x_3	+	x_3	+	x_3	−
x_4	🟨		x_4	−	x_4	−	x_4	+	x_4	+	x_4	+
x_5	🟩		x_5	+	x_5	−	x_5	+	x_5	−	x_5	+
			$\Downarrow$		$\Downarrow$		$\Downarrow$		$\Downarrow$		$\Downarrow$	
			h_1		h_2		h_3		h_4		h_5	

Error-correcting output codes (ECOC)

Prediction: for a new example $\boldsymbol{x}$

- compute the **predicted code** $\boldsymbol{c} = (h_1(\boldsymbol{x}), \ldots, h_{\mathsf{L}}(\boldsymbol{x}))^{\mathrm{T}}$
- predict the class with the **most similar code**: $k = \operatorname{argmax}_k(\boldsymbol{Mc})_k$

Suppose you have two classes

- 1: $\{$+,-,-,-,-$\}$
- 2: $\{$-,+,+,+,+$\}$

and the predicting code is $\{$+,+,+,-,-$\}$. Which class does it predict?

Class 1, since it makes only 2 mistakes.

Error-correcting output codes (ECOC)

How to pick the code $\boldsymbol{M}$?

- the more *dissimilar* the codes between different classes are, the better
- *random code* is a good choice, but might create *hard* training sets

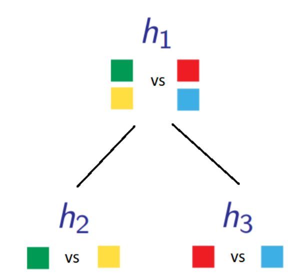
One-versus-all ($L =$ C) and One-versus-one ($L = \binom{\mathsf{C}}{2}$) are two examples of ECOC.

Tree based method

Idea: train $\approx$ C binary classifiers to learn "**belongs to which half?**".

Training: see pictures. In the tree each leaf is a single class.

			🟩🟨 vs 🟥🟦		🟩 vs 🟨		🟥 vs 🟦	
x_1	🟨		x_1	+	x_1	−		
x_2	🟥		x_2	−			x_2	+
x_3	🟦	$\Rightarrow$	x_3	−			x_3	−
x_4	🟨		x_4	+	x_4	−		
x_5	🟩		x_5	+	x_5	+		
			$\Downarrow$		$\Downarrow$		$\Downarrow$	
			h_1		h_2		h_3	



Prediction is also natural, *but is very fast!* (think ImageNet where $C \approx 20K$)

Comparisons

In big-O notation,

Reduction	**test time**	**#training points**	**remark**
OvA	C	CN	not robust
OvO	C^2	CN	can achieve very small training error
ECOC	L	LN	need diversity when designing code
Tree	$\log_2 C$	$(\log_2 C)N$	good for "extreme classification"

Outline

General idea to provide ML algorithms

1. Pick a set of **models** $\mathcal{F}$
   - e.g. $\mathcal{F} = \{f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}\}$
2. Define **error/loss** $L(y', y)$
3. Find **empirical risk minimizer (ERM)**:

$$\boldsymbol{f}^* = \operatorname*{argmin}_{f \in \mathcal{F}} \sum_{n=1}^{N} L(f(x_n), y_n)$$

Deriving classification algorithms

Let's follow the steps:

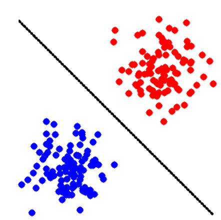
Step 1. Pick a set of models $\mathcal{F}$.

Again try linear models, but how to predict a label using $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$?

Sign of $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ predicts the label:

$$\text{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) = \begin{cases} +1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} > 0 \\ -1 & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \leq 0 \end{cases}$$

(Sometimes use sgn for sign too.)



The models

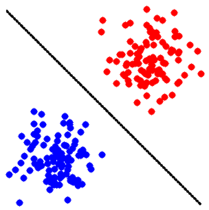
The set of **(separating) hyperplanes**:

$$\mathcal{F} = \{f(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}) \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}\}$$

Good choice for *linearly separable* data, i.e., $\exists \boldsymbol{w}$ s.t.

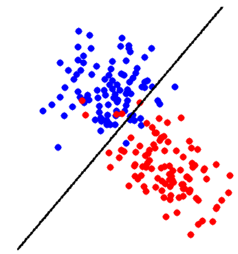
$$\text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x_n}) = y_n \quad \text{or} \quad y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x_n} > 0$$

for all $n \in [N]$.



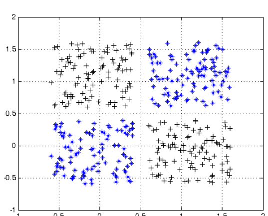
The models

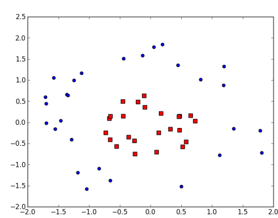
Still makes sense for "almost" linearly separable data



The models

For clearly not linearly separable data,





Again can apply a **nonlinear mapping** $\boldsymbol{\Phi}$:

$$\mathcal{F} = \{f(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\Phi}(\boldsymbol{x})) \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{M}}\}$$

More discussions in the lecture on kernels.

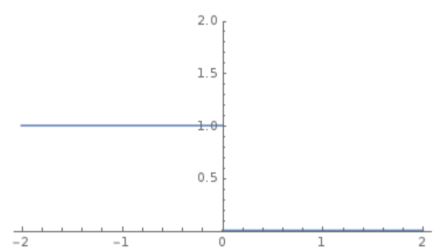
0-1 Loss

Step 2. Define error/loss $L(y', y)$.

Most natural one for classification: **0-1 loss** $L(y', y) = \mathbb{I}[y' \neq y]$

For classification, more convenient to look at the loss **as a function of** $y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$. That is, with

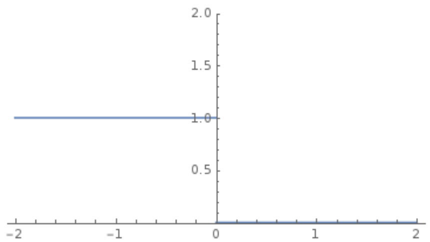
$$\ell_{\text{0-1}}(z) = \mathbb{I}[z \leq 0]$$



the loss for hyperplane $\boldsymbol{w}$ on example $(\boldsymbol{x}, y)$ is $\ell_{\text{0-1}}(y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$

Minimizing 0-1 loss is hard

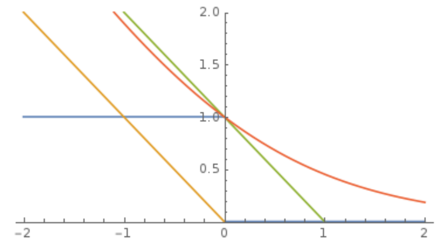
However, 0-1 loss is *not convex*, and even discontinuous.



Even worse, minimizing 0-1 loss is *NP-hard in general*.

The idea is to replace $\ell_{\text{0-1}}(z)$ which is computationally difficult by another loss function which has more advantageous properties.

Convex Surrogate Losses



- perceptron loss $\ell_{\text{perceptron}}(z) = \max\{0, -z\}$ (used in Perceptron)
- hinge loss $\ell_{\text{hinge}}(z) = \max\{0, 1 - z\}$(used in SVM and many others)
- logistic loss $\ell_{\text{logistic}}(z) = \log(1 + \exp(-z))$ (used in logistic regression)

ML becomes convex optimization

Step 3. Find empirical risk minimizer (ERM):

$$\boldsymbol{w}^* = \operatorname*{argmin}_{\boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}} \sum_{n=1}^{N} \ell(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)$$

where $\ell(\cdot)$ can be perceptron/hinge/logistic loss

- *no closed-form* in general (unlike linear regression)
- can apply general convex optimization methods

Outline

Problem 1

Why is the Hessian of logistic loss positive semidefinite?

Problem 2

For a fixed multiclass problem, which of the following multiclass-to-binary reductions has the smallest testing time complexity?

(A) One-versus-all

(B) One-versus-one

(C) Tree reduction

(D) Both (A) and (C)

Problem 3

Show that one-versus-all can be seen as a special case of error-correcting-output-code (ECOC). Specifically, write down the code matrix M for ECOC for a problem with C labels so that executing ECOC is the same as doing one-versus-all. (Note: the entry of M should be either −1 or +1.)

Problem 4

Assume we have a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$, the probability of seeing out come *y* is given by

$$P(\boldsymbol{y}|\boldsymbol{x}_n) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(\boldsymbol{y} - \boldsymbol{w}^T\boldsymbol{x}_n)^2}{2\,\sigma^2})\right)$$

Find the maximum likelihood estimations for $\boldsymbol{w}$ and σ

Solution