# CSCI567 Machine Learning (Spring 2021)


Sirisha Rambhatla

University of Southern California


Jan 29, 2021



## Outline

1. Review of Last Lecture
2. Linear Classifier and Surrogate Losses
3. Perceptron





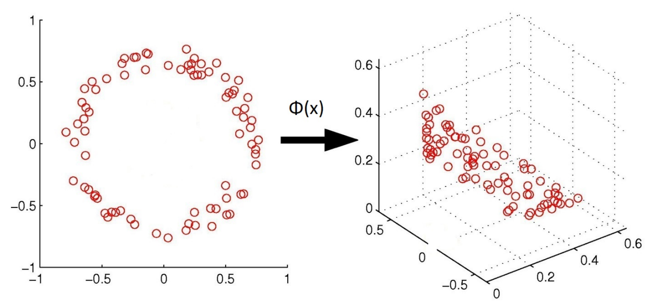
## Outline

1. Review of Last Lecture
2. Linear Classifier and Surrogate Losses
3. Perceptron





## Regression with nonlinear basis




**Model:** $f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})$ where $\boldsymbol{w} \in \mathbb{R}^M$

**Similar least square solution:** $\boldsymbol{w}^* = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi}\right)^{-1}\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{y}$

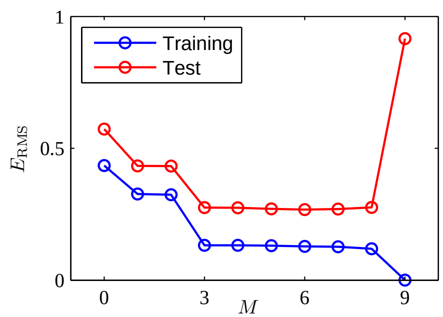
## Underfitting and Overfitting

$M \leq 2$ is *underfitting* the data

- large training error
- large test error

$M \geq 9$ is *overfitting* the data

- small training error
- **large test error**




How to prevent overfitting? more data + regularization

$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \left(\operatorname{RSS}(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_2^2\right) = \left(\boldsymbol{\Phi}^{\mathrm{T}}\boldsymbol{\Phi} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{y}$$





## General idea to derive ML algorithms

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

- e.g. $\mathcal{F} = \{f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{D}}\}$
- e.g. $\mathcal{F} = \{f(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{\Phi}(\boldsymbol{x}) \mid \boldsymbol{w} \in \mathbb{R}^{\mathsf{M}}\}$

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

Step 3. Find **empirical risk minimizer (ERM)**:

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

or **regularized empirical risk minimizer**:

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

*ML becomes optimization*





## 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 $f : \mathbb{R}^{\mathsf{D}} \to [\mathsf{C}]$

This lecture: **binary classification**

- Number of classes: $\mathsf{C} = 2$
- Labels: $\{-1, +1\}$ (cat or dog, fraud or not, price up or down...)

We have discussed **nearest neighbor classifier**:

- require carrying the training set
- more like a heuristic

## Deriving classification algorithms

Let's follow the recipe:

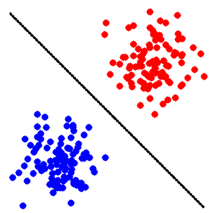
**Step 1**. Pick a set of models $\mathcal{F}$.

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:

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

(Sometimes use sgn for sign too.)







## The models

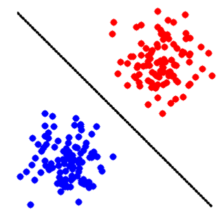
The set of **(separating) hyperplanes**:

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

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

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

for all $n \in [N]$.

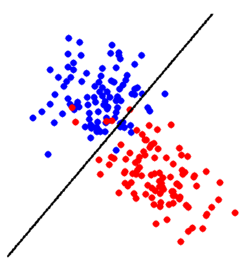






## The models

Still makes sense for "almost" linearly separable data

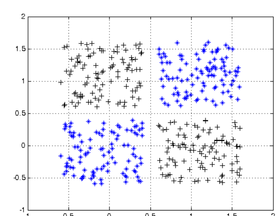


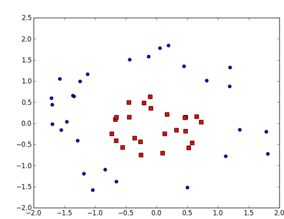




## The models

For clearly not linearly separable data,





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

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

More discussions in the next two lectures.

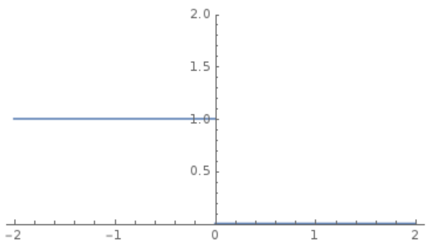
## 0-1 Loss

**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** $y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ (see ESL 4.5). That is, with

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



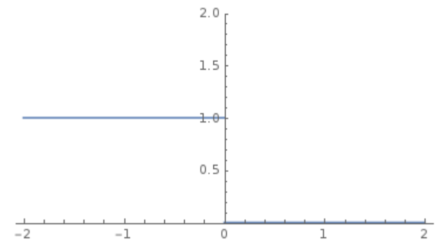

the loss for hyperplane $\boldsymbol{w}$ on example $(\boldsymbol{x}, y)$ is $\ell_{\text{0-1}}(y\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x})$





## Minimizing 0-1 loss is hard

However, 0-1 loss is *not convex*.



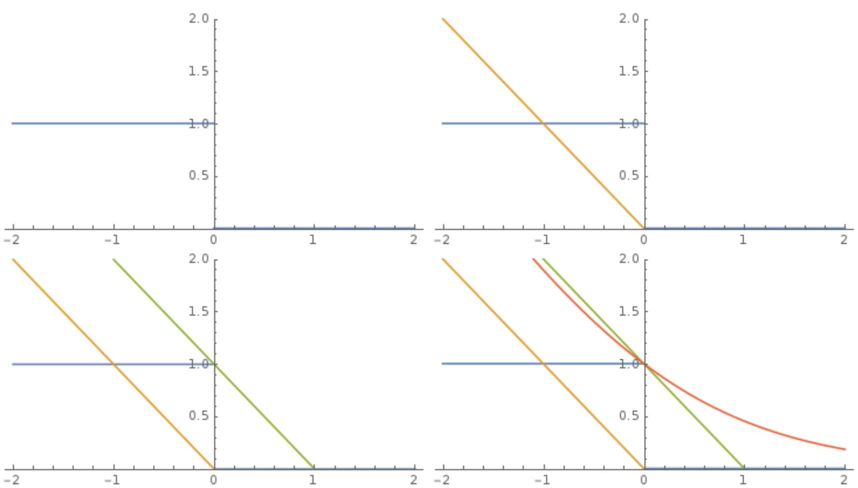

Even worse, minimizing 0-1 loss is *NP-hard in general*.





## Surrogate Losses

Solution: find a **convex surrogate loss**




- 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)





## ML becomes convex optimization

**Step 3**. Find ERM:

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

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

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

Note: minimizing perceptron loss *does not really make sense* (try $\boldsymbol{w} = \boldsymbol{0}$), but the algorithm derived from this perspective does.

## Outline







## The Perceptron Algorithm

In one sentence: Stochastic Gradient Descent applied to perceptron loss

i.e. find the minimizer of

$$F(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \ell_{\text{perceptron}}(y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n)$$
$$= \frac{1}{N}\sum_{n=1}^{N} \max\{0, -y_n \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}_n\}$$

using SGD





## A detour of numerical optimization methods

We describe two simple yet extremely popular methods

- **Gradient Descent (GD)**: simple and fundamental
- **Stochastic Gradient Descent (SGD)**: faster, effective for large-scale problems

Gradient is sometimes referred to as *first-order* information of a function. Therefore, these methods are called *first-order methods*.





## Gradient Descent (GD)

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

**Algorithm**: keep moving in the *negative gradient direction*

Start from some $\boldsymbol{w}^{(0)}$. For $t = 0, 1, 2, \ldots$

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

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

- in theory $\eta$ should be set in terms of some parameters of $F$
- in practice we just try several small values

## 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

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

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

$$t \leftarrow t + 1$$

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





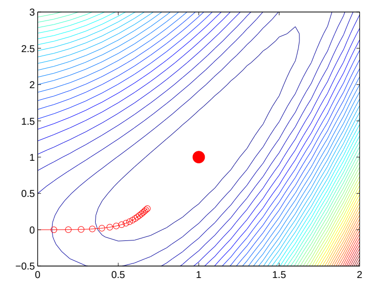
## Why GD?

Intuition: by first-order **Taylor approximation**

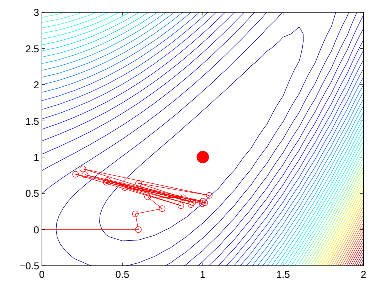
$$F(\boldsymbol{w}) \approx F(\boldsymbol{w}^{(t)}) + \nabla F(\boldsymbol{w}^{(t)})^{\mathrm{T}}(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

GD ensures

$$F(\boldsymbol{w}^{(t+1)}) \approx F(\boldsymbol{w}^{(t)}) - \eta \|\nabla F(\boldsymbol{w}^{(t)})\|_2^2 \leq F(\boldsymbol{w}^{(t)})$$



reasonable $\eta$ decreases function value



but large $\eta$ is unstable





## Stochastic Gradient Descent (SGD)

GD: keep moving in the negative gradient direction

SGD: keep moving in some *noisy* negative gradient direction

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

where $\tilde{\nabla} F(\boldsymbol{w}^{(t)})$ is a random variable (called **stochastic gradient**) s.t.

$$\mathbb{E}\left[\tilde{\nabla} F(\boldsymbol{w}^{(t)})\right] = \nabla F(\boldsymbol{w}^{(t)}) \qquad \text{(unbiasedness)}$$

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





## Convergence Guarantees

*Many* for both GD and SGD on convex objectives.

They tell you at most how many iterations you need to achieve

$$F(\boldsymbol{w}^{(t)}) - F(\boldsymbol{w}^*) \leq \epsilon$$

Even for *nonconvex objectives*, many recent works show effectiveness of GD/SGD.

## Applying GD to perceptron loss

**Objective**

$$F(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N}\max\{0, -y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n\}$$

Gradient (or really *sub-gradient*) is

$$\nabla F(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} -\mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \le 0]y_n\boldsymbol{x}_n$$

(only misclassified examples contribute to the gradient)

**GD update**

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\eta}{N}\sum_{n=1}^{N}\mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \le 0]y_n\boldsymbol{x}_n$$

*Slow: each update makes one pass of the entire training set!*





## Applying SGD to perceptron loss

How to construct a stochastic gradient?

**One common trick**: pick one example $n \in [N]$ uniformly at random, let

$$\tilde{\nabla} F(\boldsymbol{w}^{(t)}) = -\mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \le 0]y_n\boldsymbol{x}_n$$

clearly unbiased (convince yourself).

**SGD update**:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\mathbb{I}[y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n \le 0]y_n\boldsymbol{x}_n$$

*Fast: each update touches only one data point!*

Conveniently, objective of most ML tasks is a *finite sum* (over each training point) and the above trick applies!

**Exercise**: try SGD to minimize $\mathrm{RSS}$ for linear regression.





## The Perceptron Algorithm

Perceptron algorithm is SGD with $\eta = 1$ applied to perceptron loss:

Repeat:

- Pick a data point $\boldsymbol{x}_n$ uniformly at random
- If $\mathrm{sgn}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n) \neq y_n$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y_n\boldsymbol{x}_n$$

Note:

- $\boldsymbol{w}$ is always a *linear combination* of the training examples





## Why does it make sense?

If the current weight $\boldsymbol{w}$ makes a mistake

$$y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n < 0$$

then after the update $\boldsymbol{w}' = \boldsymbol{w} + y_n\boldsymbol{x}_n$ we have

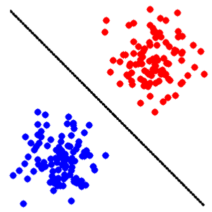
$$y_n\boldsymbol{w}'^{\mathrm{T}}\boldsymbol{x}_n = y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n + y_n^2\boldsymbol{x}_n^{\mathrm{T}}\boldsymbol{x}_n \ge y_n\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_n$$

Thus it is more likely to get it right after the update.

# Any theory?

(HW 1) If training set is linearly separable

- Perceptron *converges in a finite number of steps*
- training error is 0



There are also guarantees when the data are not linearly separable.