

DS 503: Advanced Data Analytics

Lecture 10: Machine Learning I

Based on lecture slides by
Dr. Piyush Rai at IITK

Instructor: Dr. Gagan Raj

Today's Class

○ Linear Regression problems

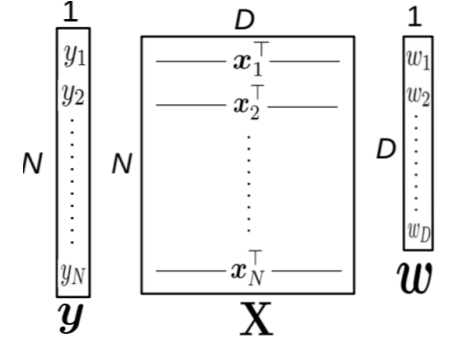
- Loss Functions for Regression
- Stability Issues in matrix operations
- Regularization to avoid over-fitting
- Different regularization objectives
- SGD to solve Linear Regression

○ Classification Problems

- Loss functions for Classification
- Perceptron algorithm as SGD
- Difficulties with perceptron
- Margin of data in a classification problem
- Online Learning classification
- SVMs to learn maximum margin separators

Linear Regression

- Given: Training data with N input-output pairs $\{(\mathbf{x}_n, y_n)\}_{n=1}^N$, $\mathbf{x}_n \in \mathbb{R}^D$, $y_n \in \mathbb{R}$
- Goal: Learn a model to predict the output for new test inputs



- Assume the function that approximates the I/O relationship to be a linear model

$$y_n \approx f(\mathbf{x}_n) = \mathbf{w}^\top \mathbf{x}_n \quad (n = 1, 2, \dots, N)$$

Can also write all of them compactly using matrix-vector notation as $\mathbf{y} \approx \mathbf{X}\mathbf{w}$

- Let's write the total error or "loss" of this model over the training data as

Goal of learning is to find the \mathbf{w} that minimizes this loss + does well on test data

$$L(\mathbf{w}) = \sum_{n=1}^N \ell(y_n, \mathbf{w}^\top \mathbf{x}_n)$$

Unlike models like KNN and DT, here we have an explicit problem-specific objective (loss function) that we wish to optimize for

$\ell(y_n, \mathbf{w}^\top \mathbf{x}_n)$ measures the prediction error or "loss" or "deviation" of the model on a single training input (\mathbf{x}_n, y_n)

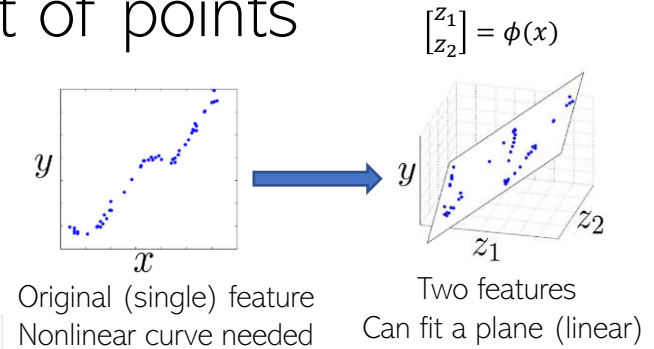
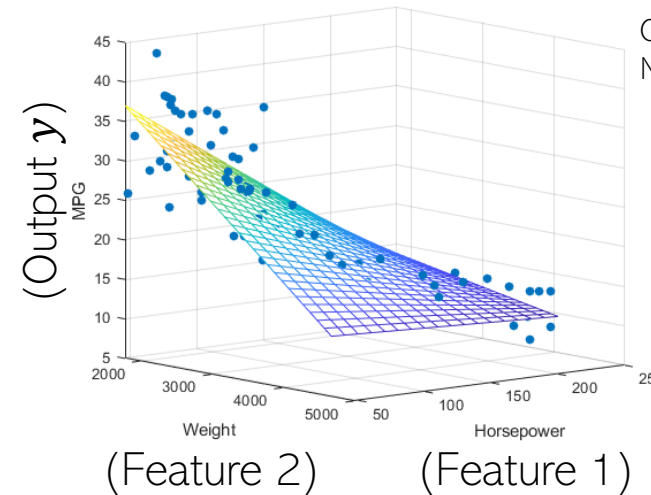
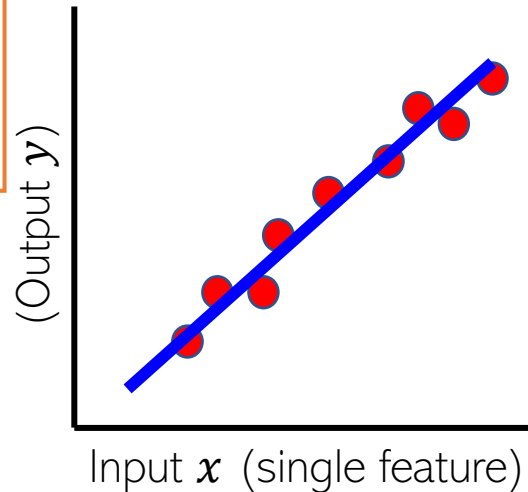


Linear Regression: Pictorially

- Linear regression is like fitting a line or (hyper)plane to a set of points

What if a line/plane doesn't model the input-output relationship very well, e.g., if their relationship is better modeled by a nonlinear curve or curved surface?

Do linear models become useless in such cases?



No. We can even fit a curve using a linear model after suitably transforming the inputs

$$y \approx \mathbf{w}^T \phi(x)$$

The transformation $\phi(\cdot)$ can be predefined or learned (e.g., using [kernel methods](#) or a deep neural network based feature extractor). More on this later

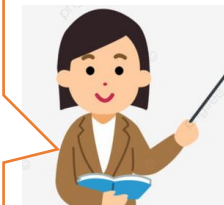
- The line/plane must also predict outputs the unseen (test) inputs well

Loss Functions for Regression

- Many possible loss functions for regression problems

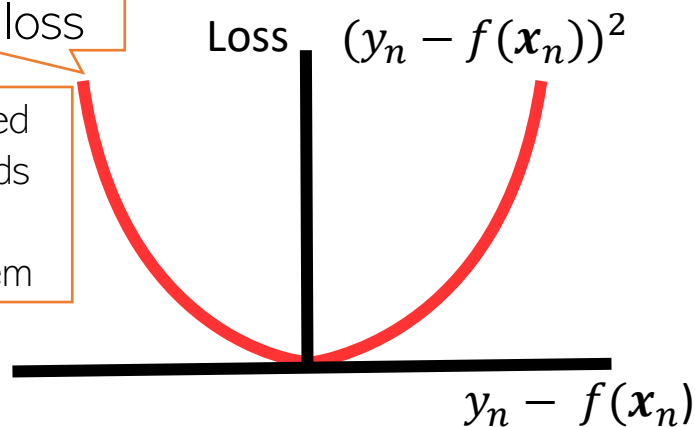
Choice of loss function usually depends on the nature of the data. Also, some loss functions result in easier optimization problem than others

5



Squared loss

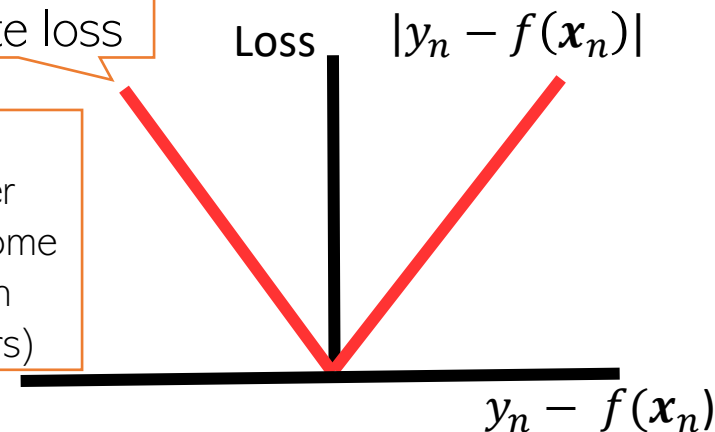
Very commonly used for regression. Leads to an easy-to-solve optimization problem



Absolute loss

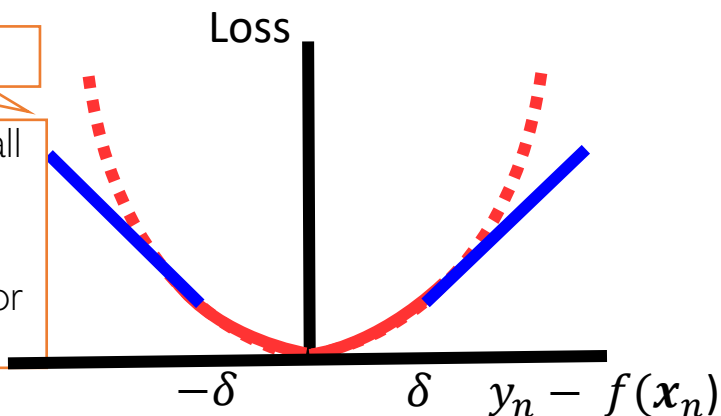
Grows more slowly than squared loss. Thus better suited when data has some outliers (inputs on which model makes large errors)

Loss $|y_n - f(x_n)|$



Huber loss

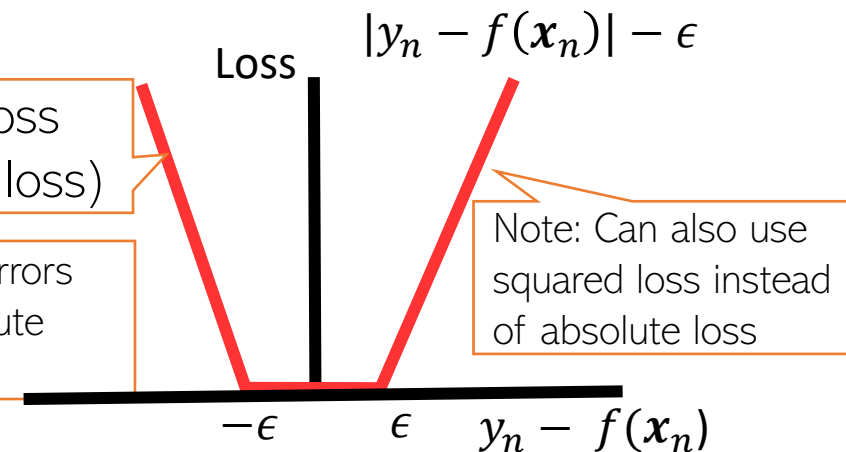
Squared loss for small errors (say up to δ); absolute loss for larger errors. Good for data with outliers



ϵ -insensitive loss (a.k.a. Vapnik loss)

Zero loss for small errors (say up to ϵ); absolute loss for larger errors

Loss $|y_n - f(x_n)| - \epsilon$



Linear Regression with Squared Loss

- In this case, the loss func will be

In matrix-vector notation, can write it compactly as $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$

$$L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

- Let us find the \mathbf{w} that optimizes (minimizes) the above squared loss
- We need calculus and optimization to do this!

The “least squares” (LS) problem
Gauss-Legendre, 18th century)

- The LS problem can be solved easily and has a closed form solution

$$\mathbf{w}_{LS} = \arg \min_{\mathbf{w}} L(\mathbf{w}) = \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$$

Closed form solutions to ML problems are rare.

$$\mathbf{w}_{LS} = \left(\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top \right)^{-1} \left(\sum_{n=1}^N y_n \mathbf{x}_n \right) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

$D \times D$ matrix inversion – can be expensive.
Ways to handle this. Will see later



Proof: A bit of calculus/optim. (more on this later) ⁷

- We wanted to find the minima of $L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$
- Let us apply basic rule of calculus: Take first derivative of $L(\mathbf{w})$ and set to zero

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 = \sum_{n=1}^N 2(y_n - \mathbf{w}^\top \mathbf{x}_n) \frac{\partial}{\partial \mathbf{w}} (y_n - \mathbf{w}^\top \mathbf{x}_n) = 0$$

Chain rule of calculus

Partial derivative of dot product w.r.t each element of \mathbf{w}

Result of this derivative is \mathbf{x}_n - same size as \mathbf{w}

- Using the fact $\frac{\partial}{\partial \mathbf{w}} \mathbf{w}^\top \mathbf{x}_n = \mathbf{x}_n$, we get $\sum_{n=1}^N 2(y_n - \mathbf{w}^\top \mathbf{x}_n) \mathbf{x}_n = 0$
- To separate \mathbf{w} to get a solution, we write the above as

$$\sum_{n=1}^N 2\mathbf{x}_n(y_n - \mathbf{x}_n^\top \mathbf{w}) = 0 \quad \longrightarrow \quad \sum_{n=1}^N y_n \mathbf{x}_n - \mathbf{x}_n \mathbf{x}_n^\top \mathbf{w} = 0$$

$$\mathbf{w}_{LS} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

Problem(s) with the Solution!

- We minimized the objective $L(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$ w.r.t. \mathbf{w} and got

$$\mathbf{w}_{LS} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

- Problem: The matrix $\mathbf{X}^\top \mathbf{X}$ may not be invertible
 - This may lead to non-unique solutions for \mathbf{w}_{opt}
- Problem: Overfitting since we only minimized loss defined on training data
 - Weights $\mathbf{w} = [w_1, w_2, \dots, w_D]$ may become arbitrarily large to fit training data perfectly
 - Such weights may perform poorly on the test data however
- One Solution: Minimize a **regularized objective** $L(\mathbf{w}) + \lambda R(\mathbf{w})$
 - The reg. will prevent the elements of \mathbf{w} from becoming too large
 - Reason: Now we are minimizing **training error** + **magnitude of vector \mathbf{w}**

$R(\mathbf{w})$ is called the **Regularizer** and measures the “magnitude” of \mathbf{w}

$\lambda \geq 0$ is the **reg. hyperparam.**
Controls how much we wish to regularize (needs to be tuned via cross-validation)

Regularized Least Squares (a.k.a. Ridge Regression)⁹

- Recall that the regularized objective is of the form $L_{reg}(\mathbf{w}) = L(\mathbf{w}) + \lambda R(\mathbf{w})$
- One possible/popular regularizer: the squared Euclidean (ℓ_2 squared) norm of \mathbf{w}

$$R(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \mathbf{w}^\top \mathbf{w}$$

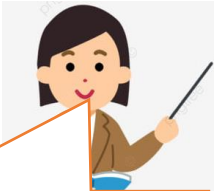
- With this regularizer, we have the regularized least squares problem as

$$\mathbf{w}_{ridge} = \arg \min_{\mathbf{w}} L(\mathbf{w}) + \lambda R(\mathbf{w})$$

$$= \arg \min_{\mathbf{w}} \sum_{n=1}^N (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \mathbf{w}^\top \mathbf{w}$$



Why is the method called "ridge" regression



Look at the form of the solution. We are adding a small value λ to the diagonals of the $D \times D$ matrix $\mathbf{X}^\top \mathbf{X}$ (like adding a ridge/mountain to some land)

- Proceeding just like the LS case, we can find the optimal \mathbf{w} which is given by

$$\mathbf{w}_{ridge} = (\sum_{n=1}^N \mathbf{x}_n \mathbf{x}_n^\top + \lambda \mathbf{I}_D)^{-1} (\sum_{n=1}^N y_n \mathbf{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{y}$$

A closer look at ℓ_2 regularization

- The regularized objective we minimized is

$$L_{reg}(\mathbf{w}) = \sum_{n=1}^N (y_n - \mathbf{w}^T \mathbf{x}_n)^2 + \lambda \mathbf{w}^T \mathbf{w}$$

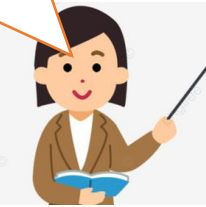
- Minimizing $L_{reg}(\mathbf{w})$ w.r.t. \mathbf{w} gives a solution for \mathbf{w} that

- Keeps the training error small
- Has a small ℓ_2 squared norm $\mathbf{w}^T \mathbf{w} = \sum_{d=1}^D w_d^2$

Good because, consequently, the individual entries of the weight vector \mathbf{w} are also prevented from becoming too large

- Small entries in \mathbf{w} are good since they lead to “smooth” models

Remember – in general, weights with large magnitude are bad since they can cause overfitting on training data and may not work well on test data



Not a “smooth” model since its test data predictions may change drastically even with small changes in some feature’s value

$\mathbf{x}_n =$	1.2	0.5	2.4	0.3	0.8	0.1	0.9	2.1
$\mathbf{x}_m =$	1.2	0.5	2.4	0.3	0.8 + ϵ	0.1	0.9	2.1

Exact same feature vectors only differing in just one feature by a small amount

$$y_n = 0.8$$

$$y_m = 100$$

Very different outputs though (maybe one of these two training ex. is an outlier)

A typical \mathbf{w} learned without ℓ_2 reg.

3.2	1.8	1.3	2.1	10000	2.5	3.1	0.1
-----	-----	-----	-----	-------	-----	-----	-----

Just to fit the training data where one of the inputs was possibly an outlier, this weight became too big. Such a weight vector will possibly do poorly on normal test inputs

Other Ways to Control Overfitting


- Use a regularizer $R(\mathbf{w})$ defined by other norms, e.g.,

ℓ_1 norm regularizer

$$\|\mathbf{w}\|_1 = \sum_{d=1}^D |w_d|$$

$$\|\mathbf{w}\|_0 = \#\text{nnz}(\mathbf{w})$$

ℓ_0 norm regularizer (counts number of nonzeros in \mathbf{w})



When should I use these regularizers instead of the ℓ_2 regularizer?

Automatic feature selection? Wow, cool!!! But how exactly?

Use them if you have a very large number of features but many irrelevant features. These regularizers can help in [automatic feature selection](#)



Note that optimizing loss functions with such regularizers is usually harder than ridge reg. but several advanced techniques exist (we will see some of those later)

Using such regularizers gives a [sparse](#) weight vector \mathbf{w} as solution

sparse means many entries in \mathbf{w} will be zero or near zero. Thus those features will be considered irrelevant by the model and will not influence prediction

- Use non-regularization based approaches

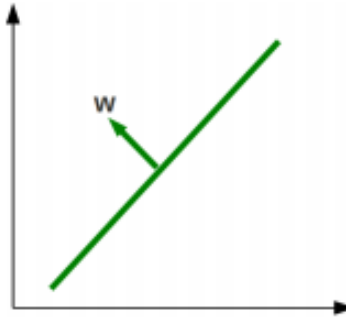
- Early-stopping (stopping training just when we have a decent val. set accuracy)
- Dropout (in each iteration, don't update some of the weights)
- Injecting noise in the inputs

All of these are very popular ways to control overfitting in deep learning models. More on these later when we talk about deep learning

Classification Techniques

Hyperplane

- Separates a D -dimensional space into two **half-spaces** (positive and negative)
- Defined by a normal vector $\mathbf{w} \in \mathbb{R}^D$ (pointing towards positive half-space)



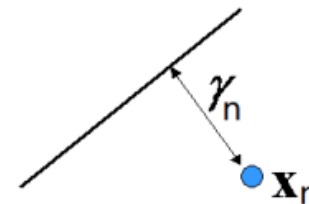
$b > 0$ means moving $\mathbf{w}^T \mathbf{x} = 0$ along the direction of \mathbf{w} ; $b < 0$ means in opp. dir.

$$\mathbf{w}^T \mathbf{x} + b = 0$$

- Equation of the hyperplane: $\mathbf{w}^T \mathbf{x} = 0$
- Assumption: The hyperplane passes through origin. If not, add a bias term b
- Distance of a point \mathbf{x}_n from a hyperplane $\mathbf{w}^T \mathbf{x} + b = 0$

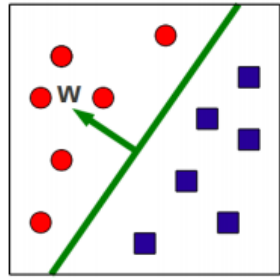
Can be positive or negative

$$\gamma_n = \frac{\mathbf{w}^T \mathbf{x}_n + b}{\|\mathbf{w}\|}$$



Hyperplane based (binary) classification

- Basic idea: Learn to separate two classes by a hyperplane $\mathbf{w}^\top \mathbf{x} + b = 0$



Prediction Rule

$$y_* = \text{sign}(\mathbf{w}^\top \mathbf{x}_* + b)$$

For multi-class classification with hyperplanes, there will be multiple hyperplanes (e.g., one for each pair of classes); more on this later

- The hyperplane may be “implied” by the model, or **learned directly**
 - Implied: Prototype-based classification, nearest neighbors, generative classification, etc
 - Directly learned: Logistic regression, **Perceptron**, **Support Vector Machine (SVM)**, etc
- The “direct” approach defines a model with params \mathbf{w} (and optionally a bias param b)
 - The parameters are learned by optimizing a **classification loss function** (will soon see examples)
 - These are also **discriminative** approaches – \mathbf{x} is not modeled but treated as fixed (given)
- The hyperplane need not be linear (e.g., can be made nonlinear using **kernels**; later)

Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \mathbf{w}^\top \mathbf{x}$), some common loss fn

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

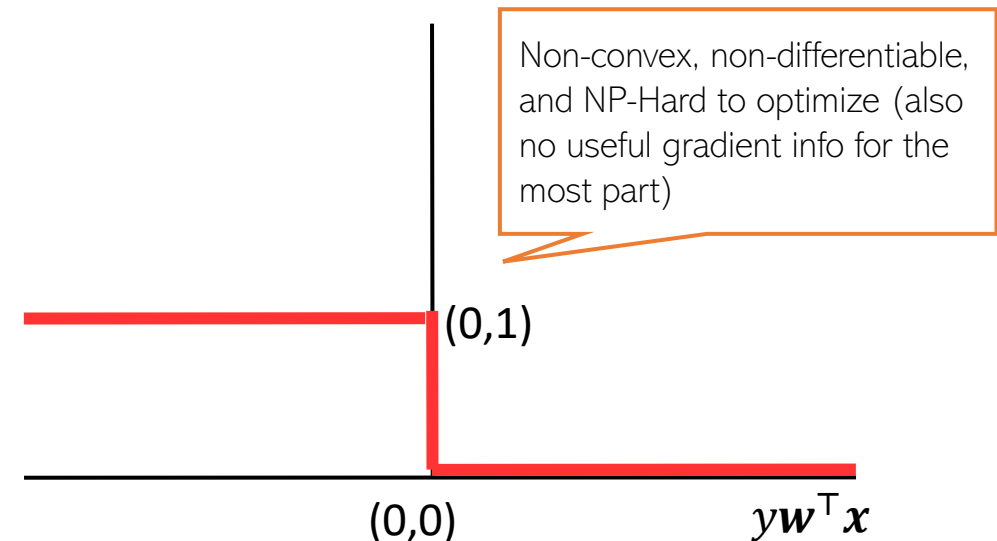
$$\ell(y, \hat{y}) = |y - \hat{y}|$$

- These measure the difference between the true output and model's prediction
- What about loss functions for classification where $\hat{y} = \text{sign}(\mathbf{w}^\top \mathbf{x})$?
- Perhaps the most natural classification loss function would be a “0-1 Loss”
 - Loss = 1 if $\hat{y} \neq y$ and Loss = 0 if $\hat{y} = y$.
 - Assuming labels as +1/-1, it means

$$\ell(y, \hat{y}) = \begin{cases} 1 & \text{if } y\mathbf{w}^\top \mathbf{x} < 0 \\ 0 & \text{if } y\mathbf{w}^\top \mathbf{x} \geq 0 \end{cases}$$

Same as $\mathbb{I}[y\mathbf{w}^\top \mathbf{x} < 0]$ or $\mathbb{I}[\text{sign}(\mathbf{w}^\top \mathbf{x}) \neq y]$

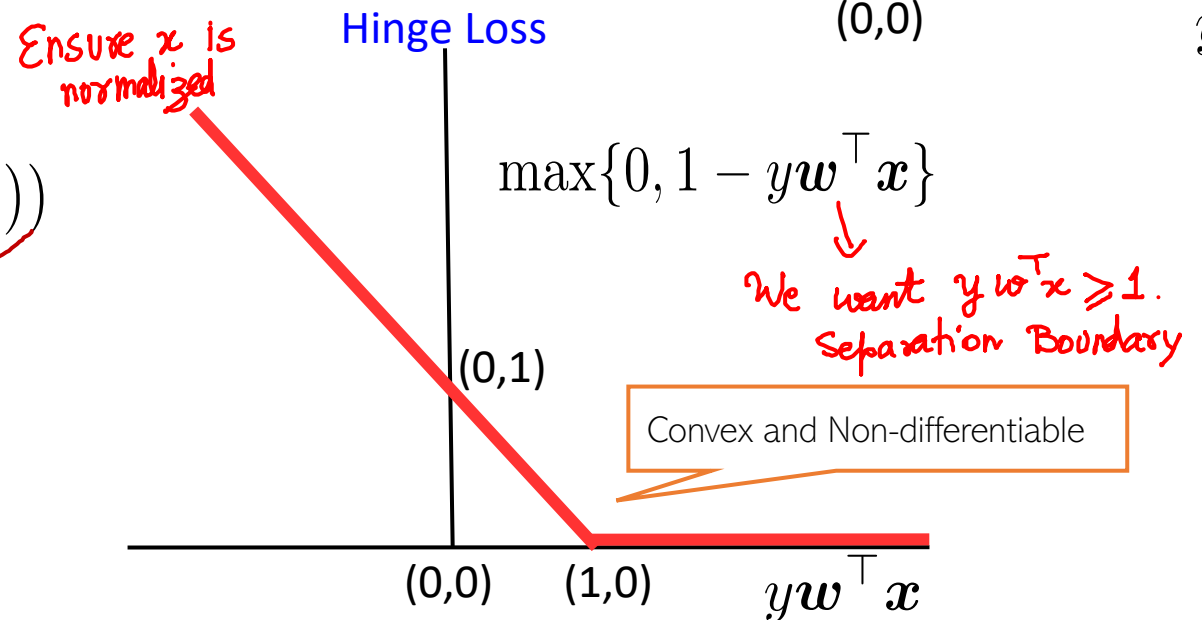
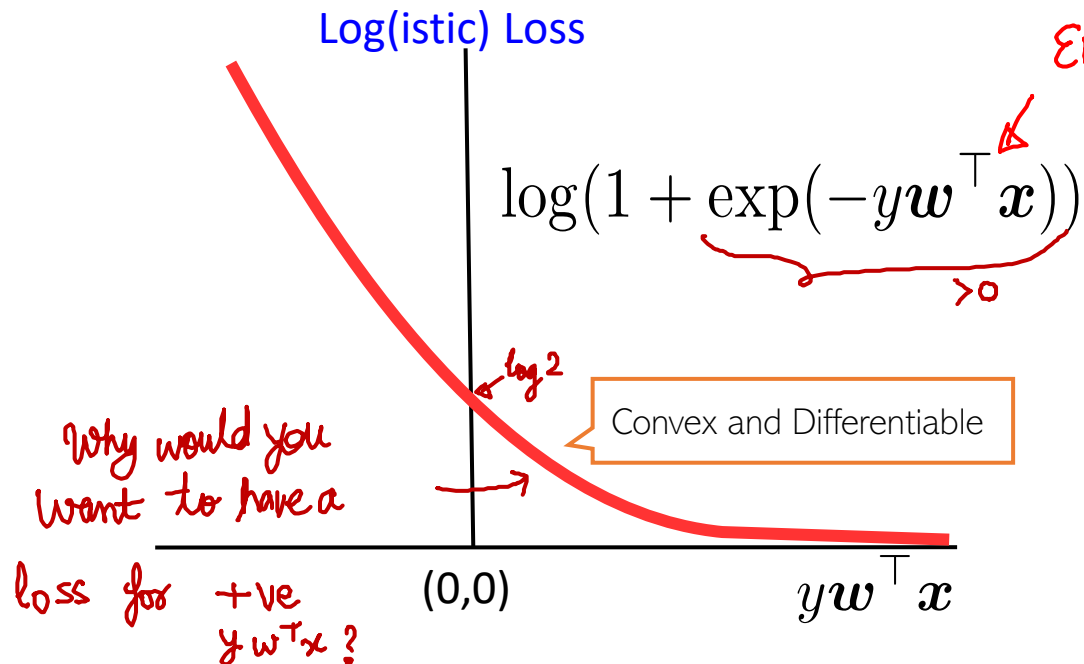
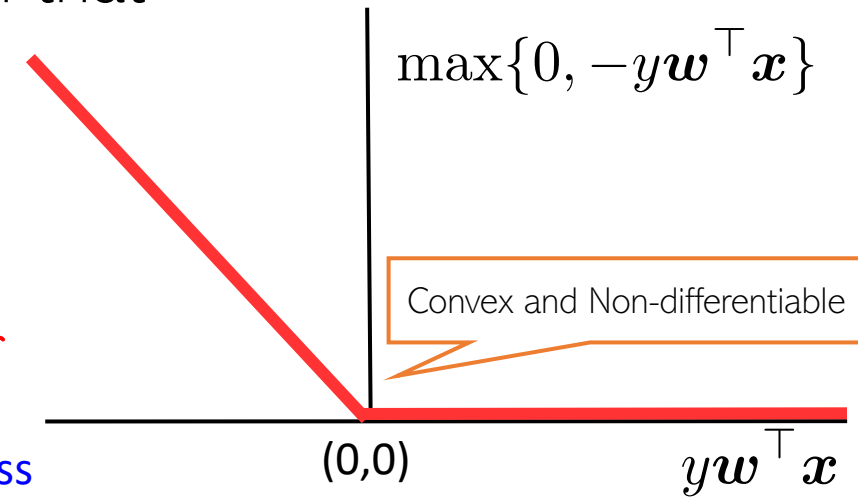
0-1 Loss



Interlude: Loss Functions for Classification

■ An ideal loss function for classification should be such that “Perceptron” Loss

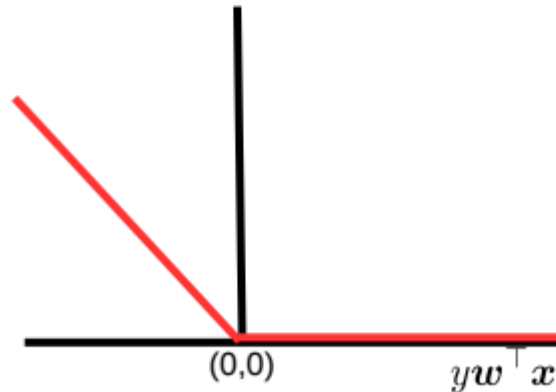
- Loss is small/zero if y and $\text{sign}(\mathbf{w}^\top \mathbf{x})$ match
- Loss is large/non-zero if y and $\text{sign}(\mathbf{w}^\top \mathbf{x})$ do not match
- Large positive $y\mathbf{w}^\top \mathbf{x} \Rightarrow$ small/zero loss
- Large negative $y\mathbf{w}^\top \mathbf{x} \Rightarrow$ large/non-zero loss



Learning by Optimizing Perceptron Loss

- Let's ignore the bias term b for now. So the hyperplane is simply $\mathbf{w}^\top \mathbf{x} = 0$
- The Perceptron loss function: $L(\mathbf{w}) = \sum_{n=1}^N \max\{0, -y_n \mathbf{w}^\top \mathbf{x}_n\}$. Let's do SGD

"Perceptron" Loss: $\max\{0, -y\mathbf{w}^\top \mathbf{x}\}$



Subgradients w.r.t. \mathbf{w}

$$\mathbf{g}_n = \begin{cases} 0, & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n > 0 \\ -y_n \mathbf{x}_n & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n < 0 \\ k y_n \mathbf{x}_n & \text{for } y_n \mathbf{w}^\top \mathbf{x}_n = 0 \quad (\text{where } k \in [-1, 0]) \end{cases}$$

One randomly chosen example in each iteration

- If we use $k = 0$ then $\mathbf{g}_n = 0$ for $y_n \mathbf{w}^\top \mathbf{x}_n \geq 0$, and $\mathbf{g}_n = -y_n \mathbf{x}_n$ for $y_n \mathbf{w}^\top \mathbf{x}_n < 0$
- Non-zero gradients only when the model makes a mistake on current example (\mathbf{x}_n, y_n)
- Thus SGD will update \mathbf{w} only when there is a mistake (mistake-driven learning)

The Perceptron Algorithm

- Stochastic Sub-grad desc on Perceptron loss is also known as the Perceptron algorithm

Stochastic SubGD

- 1 Initialize $\mathbf{w} = \mathbf{w}^{(0)}$, $t = 0$, set $\eta_t = 1, \forall t$
- 2 Pick some (\mathbf{x}_n, y_n) randomly.
- 3 If current \mathbf{w} makes a **mistake** on (\mathbf{x}_n, y_n) , i.e., $y_n \mathbf{w}^{(t)\top} \mathbf{x}_n < 0$

$$\begin{aligned}\mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + y_n \mathbf{x}_n \\ t &= t + 1\end{aligned}$$
- 4 If not converged, go to step 2.

Note: An example may get chosen several times during the entire run

Mistake condition

Updates are “corrective”: If $y_n = +1$ and $\mathbf{w}^\top \mathbf{x}_n < 0$, after the update $\mathbf{w}^\top \mathbf{x}_n$ will be less negative. Likewise, if $y_n = -1$ and $\mathbf{w}^\top \mathbf{x}_n > 0$, after the update $\mathbf{w}^\top \mathbf{x}_n$ will be less positive

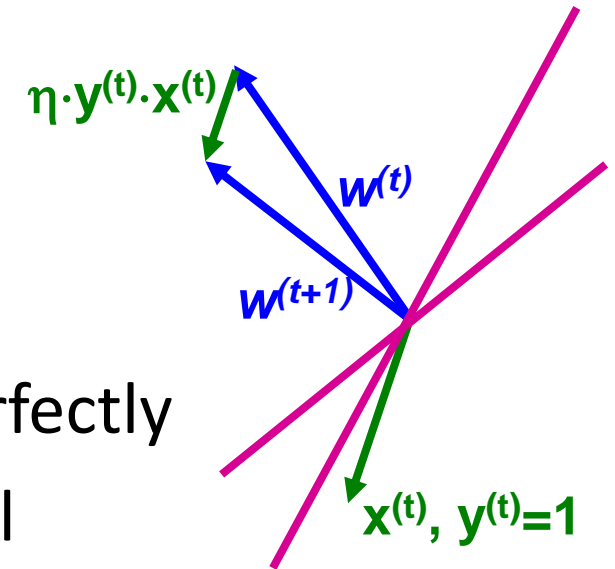
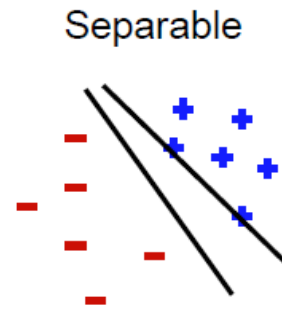


If training data is linearly separable, the Perceptron algo will converge in a finite number of iterations (Block & Novikoff theorem)

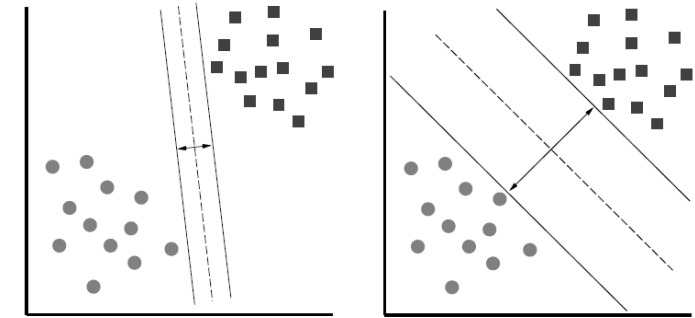
- An example of an **online learning** algorithm (processes one training ex. at a time)
- Assuming $\mathbf{w}^{(0)} = \mathbf{0}$, easy to see that the final \mathbf{w} has the form $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$
 - α_n is total number of mistakes made by the algorithm on example (\mathbf{x}_n, y_n)
 - As we'll see, many other models also have weights \mathbf{w} in the form $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$

Meaning of α_n may be different

Perceptron in Action



- **Separability:** Some parameters separate training set perfectly
- **Convergence:** If training set is separable, perceptron will converge. How fast will it converge? Depends on the margin
- Margin w.r.t. HP: $\text{margin}(D, w) = \min_{(x,y) \in D} y(w^T x)$
- $\text{Margin}(D) = \gamma = \sup_w \text{margin}(D, w), \|w\|_2 = 1$
 - Minimum distance of any example to the optimal plane
- **(Training) Mistake bound:** Number of mistakes $< \frac{1}{\gamma^2}$



Multiclass perceptron

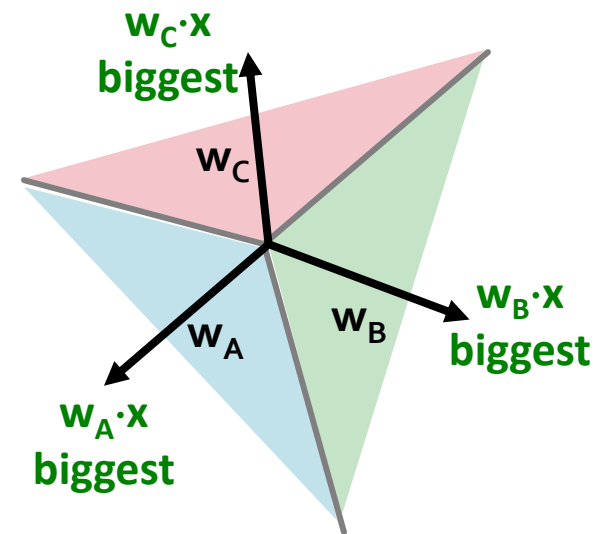
- What if more than 2 classes?
- Weight vector w_c for each class c
 - Train one class vs. the rest:
 - Example: 3-way classification $y = \{A, B, C\}$
 - Train 3 classifiers: w_A : A vs. B,C; w_B : B vs. A,C; w_C : C vs. A,B

- Calculate activation for each class

$$f(x, c) = w_c^T x$$

- Highest activation wins

$$c = \arg \max_c f(x, c)$$



XOR in the real world

- Perceptron being a linear separator cannot generate non-linear decision boundaries
- You may wonder, do XOR like problems exist in the real world?
- Example from NLP (sentiment analysis):
 - Reviews have keywords which we use to score the review
 - Assume only 3 features (whether these words are present or absent)
 - Excellent: +ve , Terrible: -ve , Not: Flips the categories
- Solution: Multi-Layer Perceptron, Neural networks, Kernels
- BQ 10.1: Explain geometry of the above problem using Hypercube

Online Learning

- **New setting: Online Learning**
 - Allows for modeling problems where we have a continuous stream of data
 - We want an algorithm to learn from it and slowly adapt to the changes in data
- **Idea: Do slow updates to the model**
 - Perceptron algorithm makes updates if it misclassifies an example
 - **So:** First train the classifier on training data. Then for every example from the stream, if we misclassify, update the model (using small learning rate, η)
- Update Rule: $w^{t+1} = w^t + \eta y^{(t)} x^{(t)}$

Example: Dynamic Pricing of Services

- **Airline Ticket:**

- User comes and tell us origin, destination, date of travel
- We offer to sell the ticket for some money (\$100 - \$500)
- Based on the price we offer, sometimes the user buys the ticket ($y = 1$), sometimes they don't ($y = -1$)

- **Task:** Build an algorithm to optimize what price we offer to the users

- **Features x capture:**

- Information about user
- Origin and destination, how far are the dates of travel
- Seasonal effects: vacation/holiday season, festivals
- Current demand: many users looking vs. low demand, many seats left

- **Problem: Will user accept the price?**

Dynamic Pricing

- **Model whether user will accept our price: $y = f(x; w)$**
 - **Accept: $y = 1$, Not accept: $y = -1$**
 - Build this model with say Perceptron
- **The website that runs continuously**
- **Online learning algorithm would do something like**
 - User makes the search along with the necessary information (login as well)
 - He is represented as an **(x, y)** pair where
 - **x** : Feature vector including price we offer, origin, destination
 - **y** : If they chose to use our service or not
 - The algorithm updates **w** using just the **(x, y)** pair
 - Basically, we update the **w** parameters every time we get some new data

Data set vs. Data stream

- We discard this idea of a data “set”
- Instead we have a continuous stream of data
- **Further comments:**
 - For a major website where you have a massive stream of data then this kind of algorithm is pretty reasonable
 - Don't need to deal with all the training data
 - If you had a small number of users you could save their data and then run a normal algorithm on the full dataset
 - Doing multiple passes over the data

Adaptive Algorithm

- An online algorithm can adapt to changing user preferences
- For example, over time users may become more price sensitive
- **The algorithm adapts and learns this**
- So the system is dynamic

Potential Issues

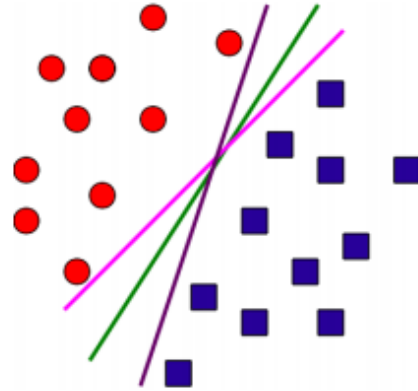
- Weights may oscillate a lot!
- You may lose a good classifier
- Average Perceptron:
 - Keep a weighted (# of correct examples) average of the weight vectors
 - Good separators are not lost

Algorithm 7 AVERAGEPERCEPTRONTRAIN(\mathbf{D} , $MaxIter$)

```
1:  $w \leftarrow \langle 0, 0, \dots, 0 \rangle$  ,  $b \leftarrow 0$  // initialize weights and bias
2:  $u \leftarrow \langle 0, 0, \dots, 0 \rangle$  ,  $\beta \leftarrow 0$  // initialize cached weights and bias
3:  $c \leftarrow 1$  // initialize example counter to one
4: for  $iter = 1 \dots MaxIter$  do
5:   for all  $(x, y) \in \mathbf{D}$  do
6:     if  $y(w \cdot x + b) \leq 0$  then
7:        $w \leftarrow w + y x$  // update weights
8:        $b \leftarrow b + y$  // update bias
9:        $u \leftarrow u + y c x$  // update cached weights
10:       $\beta \leftarrow \beta + y c$  // update cached bias
11:     end if
12:      $c \leftarrow c + 1$  // increment counter regardless of update
13:   end for
14: end for
15: return  $w - \frac{1}{c} u$ ,  $b - \frac{1}{c} \beta$  // return averaged weights and bias
```

Perceptron and (lack of) Margins

- Perceptron would learn a hyperplane (of many possible) that separates the classes



Basically, it will learn the hyperplane which corresponds to the \mathbf{w} that minimizes the Perceptron loss

Kind of an “unsafe” situation to have – ideally would like it to be reasonably away from closest training examples from either class

- Doesn't guarantee any “margin” around the hyperplane
 - The hyperplane can get arbitrarily close to some training example(s) on either side
 - This may not be good for generalization performance
- Can artificially introduce margin by changing the mistake condition to $\mathbf{y}_n \mathbf{w}^\top \mathbf{x}_n < \gamma$
- Support Vector Machine (SVM) does it directly by learning the **max. margin hyperplane**

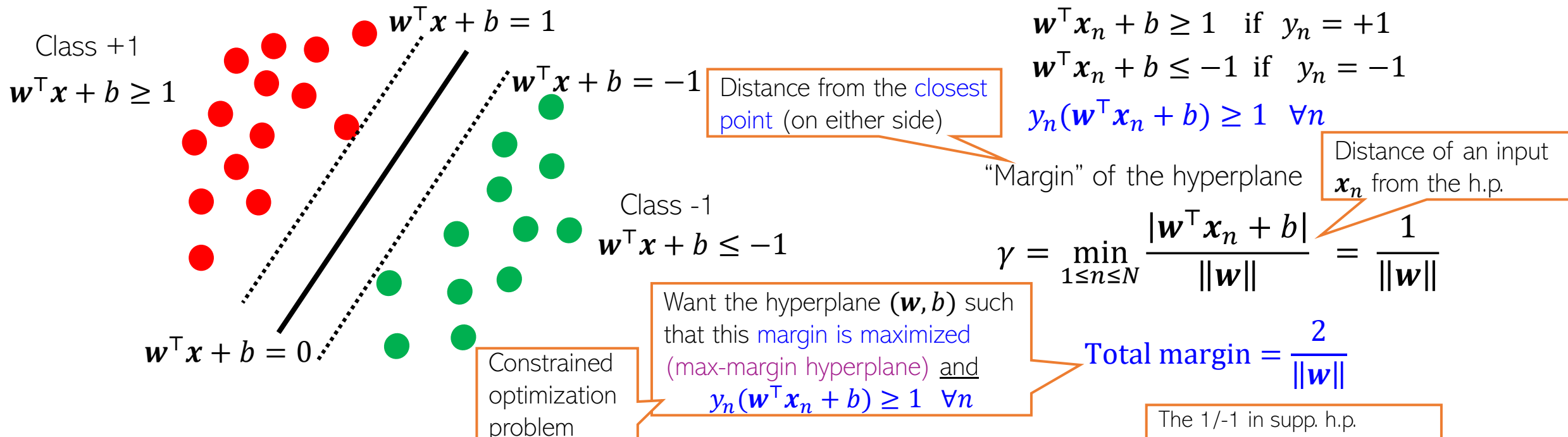
$\gamma > 0$ is some pre-specified margin

Support Vector Machine (SVM)

29

SVM originally proposed by Vapnik and colleagues in early 90s

- Hyperplane based classifier. Ensures a large margin around the hyperplane
- Will assume a linear hyperplane to be of the form $\mathbf{w}^T \mathbf{x} + b = 0$ (nonlinear ext. later)

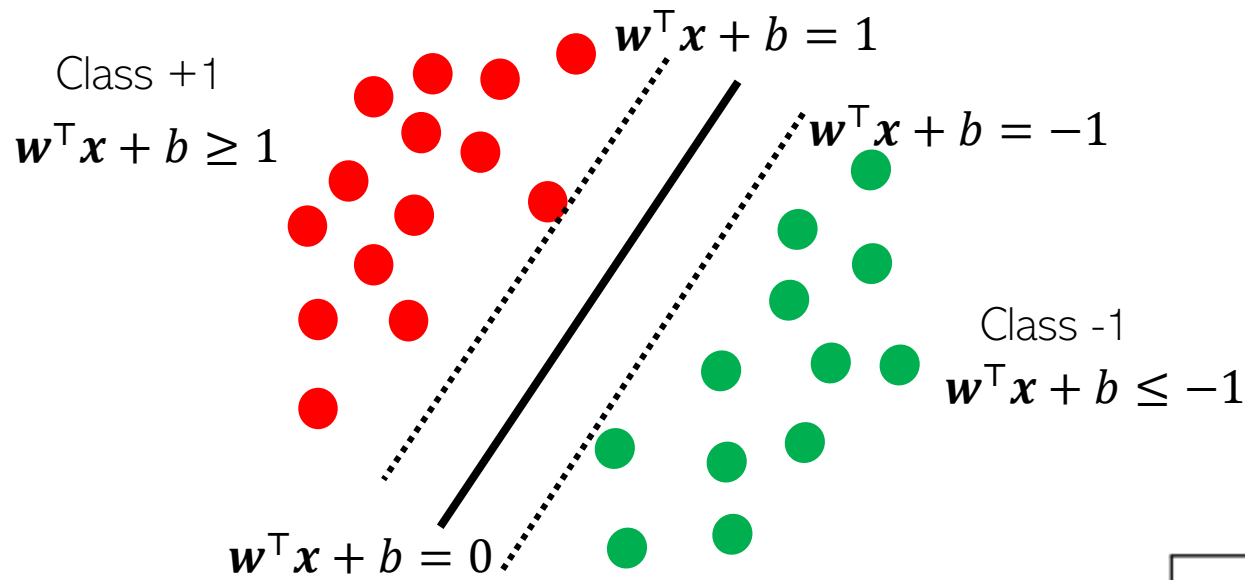


- Two other "supporting" hyperplanes defining a "no man's land"
 - Ensure that zero training examples fall in this region (will relax later)
 - The SVM idea: Position the hyperplane s.t. this region is as "wide" as possible

The 1/-1 in supp. h.p. equations is arbitrary; can replace by any scalar m/-m and solution won't change, except a simple scaling of \mathbf{w}

Hard-Margin SVM

- Hard-Margin: Every training example must fulfil margin condition $y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1$
- Meaning: Must not have any example in the no-man's land

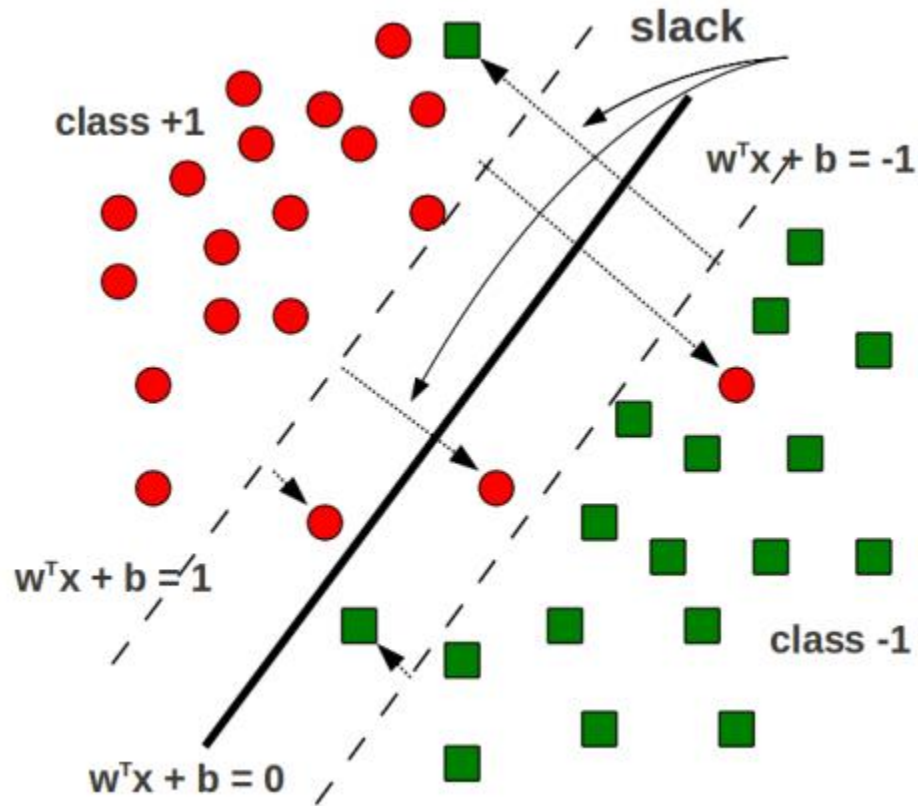


Constrained optimization problem with N inequality constraints. Objective and constraints both are convex

- Also want to maximize margin $2\gamma = \frac{2}{\|\mathbf{w}\|}$
- Equivalent to minimizing $\|\mathbf{w}\|^2$ or $\frac{\|\mathbf{w}\|^2}{2}$
- The objective func. for hard-margin SVM

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1, \quad n = 1, \dots, N \end{aligned}$$

Soft-Margin SVM (More Commonly Used)



- Allow some training examples to fall within the no-man's land (margin region)
- Even okay for some training examples to fall totally on the wrong side of h.p.
- Extent of “violation” by a training input (\mathbf{x}_n, y_n) is known as **slack** $\xi_n \geq 0$
- $\xi_n > 1$ means totally on the wrong side

$$\mathbf{w}^T \mathbf{x}_n + b \geq 1 - \xi_n \quad \text{if } y_n = +1$$

$$\mathbf{w}^T \mathbf{x}_n + b \leq -1 + \xi_n \quad \text{if } y_n = -1$$

Soft-margin constraint: $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n \quad \forall n$

Soft-Margin SVM (Contd)

- Goal: Still want to maximize the margin such that
 - Soft-margin constraints $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n$ are satisfied for all training ex.
 - Do not have too many margin violations (sum of slacks $\sum_{n=1}^N \xi_n$ should be small)

Sum of slacks is like the training error

- The objective func. for soft-margin SVM

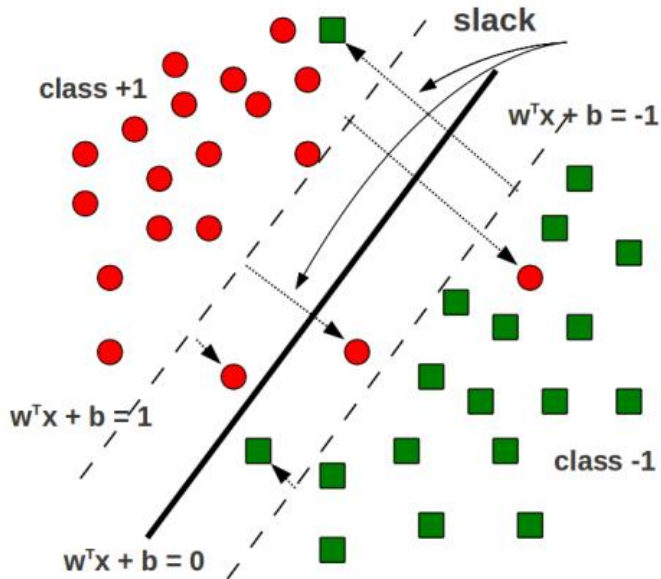
$$\min_{\mathbf{w}, b, \xi} f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n$$

subject to $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N$

Annotations:

- Inversely prop. to margin (points to $\frac{\|\mathbf{w}\|^2}{2}$)
- Trade-off hyperparam (points to C)
- training error (points to $\sum_{n=1}^N \xi_n$)
- Constrained optimization problem with $2N$ inequality constraints. Objective and constraints both are convex (points to the entire problem)

- Hyperparameter C controls the trade off between large margin and small training error (need to tune)
 - Large C : small training error but also small margin (bad)
 - Small C : large margin but large training error (bad)

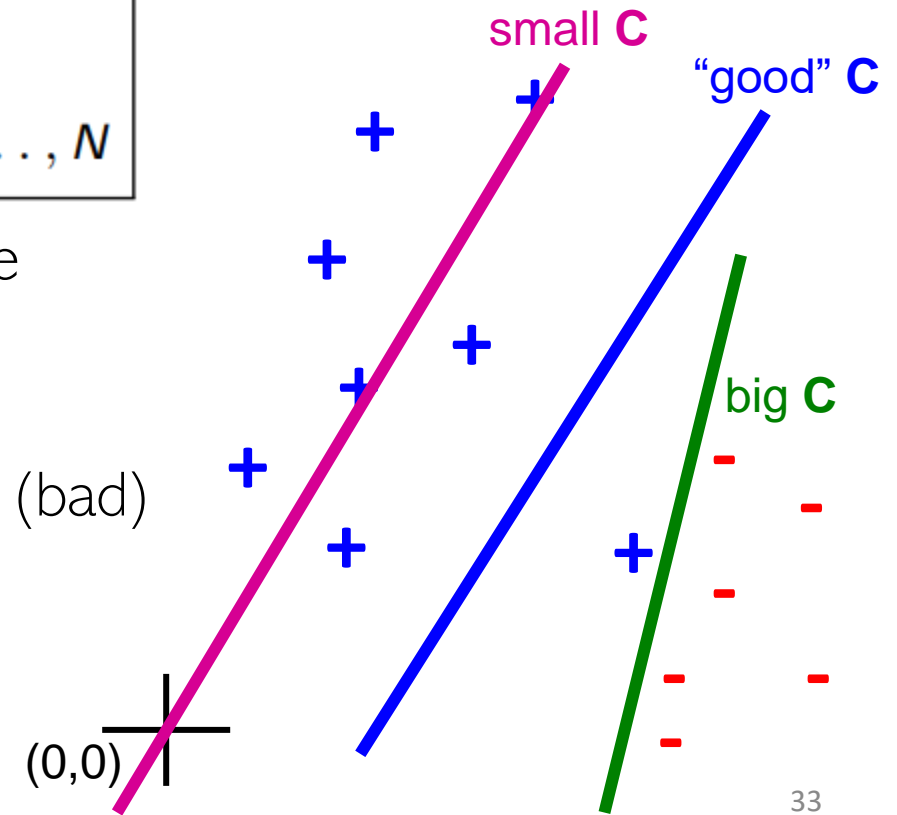


Choosing C

$$\min_{\mathbf{w}, b, \xi} f(\mathbf{w}, b, \xi) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n$$

subject to $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N$

- Hyperparameter C controls the trade off between large margin and small training error (need to tune)
 - $C = \infty$: Only want to \mathbf{w}, \mathbf{b} that separate the data
 - Large C : Small training error but also small margin. (bad)
 - Small C : large margin but large training error (bad)
 - $C = 0$: Can set ξ_i to anything, then $\mathbf{w} = \mathbf{0}$ (basically ignores the data)



Solving Hard-Margin SVM

- The hard-margin SVM optimization problem is

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & f(\mathbf{w}, b) = \frac{\|\mathbf{w}\|^2}{2} \\ \text{subject to} \quad & 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad n = 1, \dots, N \end{aligned}$$

- A [constrained optimization](#) problem. One option is to solve using [Lagrange's method](#)
- Introduce Lagrange multipliers α_n ($n = 1, \dots, N$), one for each constraint, and solve

$$\min_{\mathbf{w}, b} \max_{\alpha \geq 0} \mathcal{L}(\mathbf{w}, b, \alpha) = \frac{\|\mathbf{w}\|^2}{2} + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)\}$$

- $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_N]$ denotes the vector of Lagrange multipliers
- It is easier (and helpful; we will soon see why) to solve the dual: min and then max

Solving Hard-Margin SVM

- The dual problem (min then max) is

$$\max_{\alpha \geq 0} \min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \alpha) = \frac{\mathbf{w}^T \mathbf{w}}{2} + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)\}$$

Note: if we ignore the bias term b then we don't need to handle the constraint $\sum_{n=1}^N \alpha_n y_n = 0$ (problem becomes a bit more easy to solve)



Otherwise, the α_n 's are coupled and some opt. techniques such as co-ordinate ascent can't easily be applied

- Take (partial) derivatives of \mathcal{L} w.r.t. \mathbf{w} and b and setting them to zero gives (verify)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n \quad \frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^N \alpha_n y_n = 0$$

α_n tells us how important training example (\mathbf{x}_n, y_n) is

- The solution \mathbf{w} is simply a weighted sum of all the training inputs
- Substituting $\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n$ in the Lagrangian, we get the dual problem as (verify)

$$\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{m,n=1}^N \alpha_m \alpha_n y_m y_n (\mathbf{x}_m^T \mathbf{x}_n)$$

Note that inputs appear only as pairwise dot products. This will be useful later on when we make SVM nonlinear using kernel methods



$$\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \alpha^T \mathbf{1} - \frac{1}{2} \alpha^T \mathbf{G} \alpha$$

\mathbf{G} is an $N \times N$ p.s.d. matrix, also called the Gram Matrix, $G_{nm} = y_n y_m \mathbf{x}_n^T \mathbf{x}_m$, and $\mathbf{1}$ is a vector of all 1s

This is also a “quadratic program” (QP) – a quadratic function of the variables α

Maximizing a concave function (or minimizing a convex function) s.t. $\alpha \geq 0$ and $\sum_{n=1}^N \alpha_n y_n = 0$. Many methods to solve it.

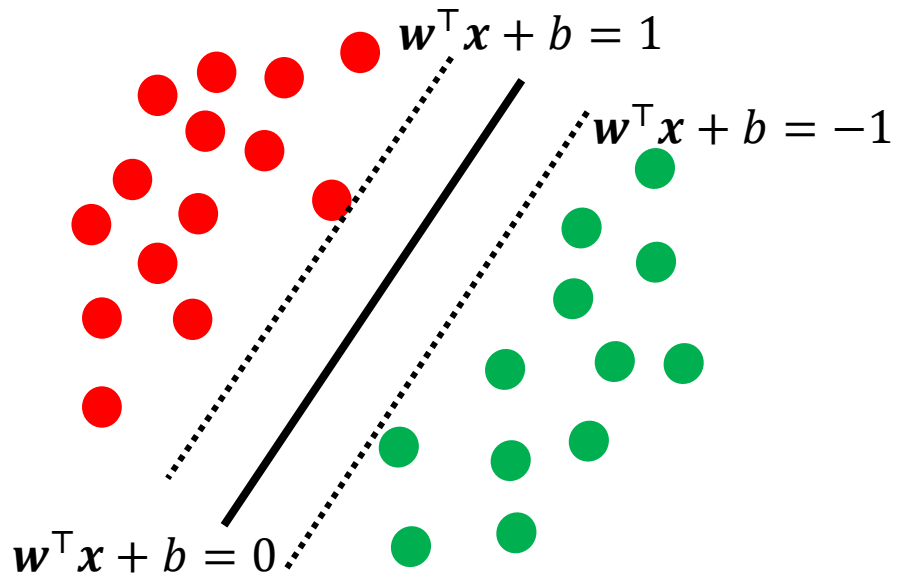
Solving Hard-Margin SVM

- Once we have the α_n 's by solving the dual, we can get \mathbf{w} and b as

$$\mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n \quad (\text{we already saw this})$$

$$b = -\frac{1}{2} \left(\min_{n:y_n=+1} \mathbf{w}^T \mathbf{x}_n + \max_{n:y_n=-1} \mathbf{w}^T \mathbf{x}_n \right) \quad (\text{exercise})$$

- A nice property: Most α_n 's in the solution will be zero (sparse solution)



- Reason: KKT conditions
- For the optimal α_n 's, we must have
- Thus α_n nonzero only if $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$, i.e., the training example lies on the boundary

$$\alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)\} = 0$$

- These examples are called support vectors

Solving Soft-Margin SVM

- Recall the soft-margin SVM optimization problem

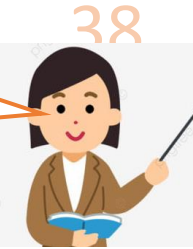
$$\begin{aligned} \min_{\mathbf{w}, b, \boldsymbol{\xi}} \quad & f(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ \text{subject to} \quad & 1 \leq y_n(\mathbf{w}^T \mathbf{x}_n + b) + \xi_n, \quad -\xi_n \leq 0 \quad n = 1, \dots, N \end{aligned}$$

- Here $\boldsymbol{\xi} = [\xi_1, \xi_2, \dots, \xi_N]$ is the vector of [slack variables](#)
- Introduce Lagrange multipliers α_n, β_n for each constraint and solve Lagrangian

$$\min_{\mathbf{w}, b, \boldsymbol{\xi}} \max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta} \geq 0} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^N \beta_n \xi_n$$

- The terms in red color above were not present in the hard-margin SVM
- Two set of dual variables $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_N]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \dots, \beta_N]$
- Will eliminate the primal var $\mathbf{w}, b, \boldsymbol{\xi}$ to get dual problem containing the dual variables

Solving Soft-Margin SVM



- The Lagrangian problem to solve

Note: if we ignore the bias term b then we don't need to handle the constraint $\sum_{n=1}^N \alpha_n y_n = 0$ (problem becomes a bit more easy to solve)

Otherwise, the α_n 's are coupled and some opt. techniques such as co-ordinate aspect can't easily applied

$$\min_{\mathbf{w}, b, \xi} \max_{\alpha \geq 0, \beta \geq 0} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \alpha_n \{1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) - \xi_n\} - \sum_{n=1}^N \beta_n \xi_n$$

- Take (partial) derivatives of \mathcal{L} w.r.t. \mathbf{w} , b , and ξ_n and setting to zero gives

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{n=1}^N \alpha_n y_n \mathbf{x}_n, \quad \frac{\partial \mathcal{L}}{\partial b} = 0 \Rightarrow \sum_{n=1}^N \alpha_n y_n = 0, \quad \frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Rightarrow C - \alpha_n - \beta_n = 0$$

Weighted sum of training inputs

- Using $C - \alpha_n - \beta_n = 0$ and $\beta_n \geq 0$, we have $\alpha_n \leq C$ (for hard-margin, $\alpha_n \geq 0$)

- Substituting these in the Lagrangian \mathcal{L} gives the Dual problem

The dual variables β don't appear in the dual problem!

Given α , \mathbf{w} and b can be found just like the hard-margin SVM case

$$\max_{\alpha \leq C, \beta \geq 0} \mathcal{L}_D(\alpha, \beta) = \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{m,n=1}^N \alpha_m \alpha_n y_m y_n (\mathbf{x}_m^T \mathbf{x}_n) \quad \text{s.t.} \quad \sum_{n=1}^N \alpha_n y_n = 0$$

Maximizing a concave function (or minimizing a convex function) s.t. $\alpha \leq C$ and $\sum_{n=1}^N \alpha_n y_n = 0$. Many methods to solve it.

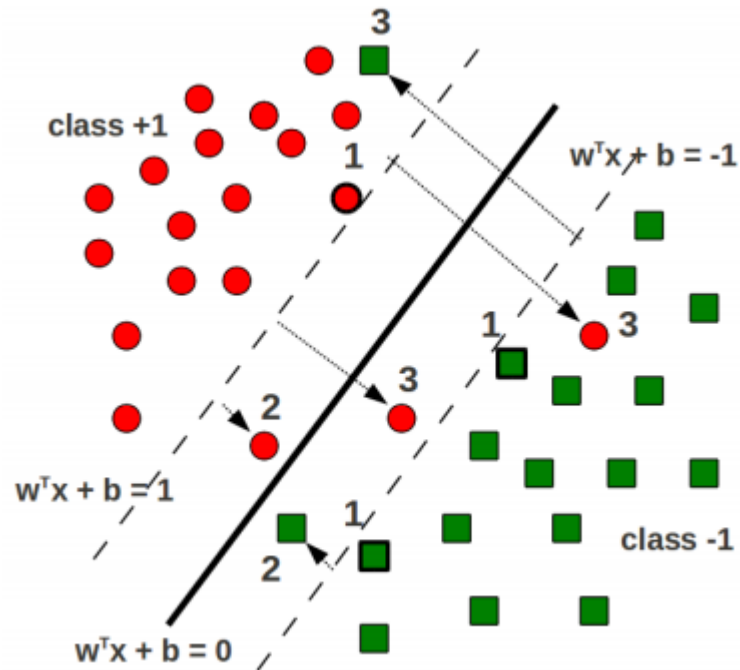
$$\max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \alpha^T \mathbf{1} - \frac{1}{2} \alpha^T \mathbf{G} \alpha$$

In the solution, α will still be sparse just like the hard-margin SVM case. Nonzero α_n correspond to the support vectors

(Note: For various SVM solvers, can see "Support Vector Machine Solvers" by Bottou and Lin)

Support Vectors in Soft-Margin SVM

- The hard-margin SVM solution had only one type of support vectors
 - All lied on the supporting hyperplanes $\mathbf{w}^T \mathbf{x}_n + b = 1$ and $\mathbf{w}^T \mathbf{x}_n + b = -1$
- The soft-margin SVM solution has three types of support vectors (with nonzero α_n)



1. Lying on the supporting hyperplanes
2. Lying within the margin region but still on the correct side of the hyperplane
3. Lying on the wrong side of the hyperplane (misclassified training examples)

SVMs via Dual Formulation: Some Comments

- Recall the final dual objectives for hard-margin and soft-margin SVM

Hard-Margin SVM: $\max_{\alpha \geq 0} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$

Soft-Margin SVM: $\max_{\alpha \leq C} \mathcal{L}_D(\alpha) = \alpha^\top \mathbf{1} - \frac{1}{2} \alpha^\top \mathbf{G} \alpha$

Note: Both these ignore the bias term b otherwise will need another constraint $\sum_{n=1}^N \alpha_n y_n = 0$

- The dual formulation is nice due to two primary reasons
 - Allows conveniently handling the margin based constraint (via Lagrangians)
 - Allows learning nonlinear separators by replacing inner products in $G_{nm} = y_n y_m \mathbf{x}_n^\top \mathbf{x}_m$ by general kernel-based similarities (more on this when we talk about kernels)
- However, dual formulation can be expensive if N is large (esp. compared to D)
 - Need to solve for N variables $\alpha = [\alpha_1, \alpha_2, \dots, \alpha_N]$
 - Need to pre-compute and store $N \times N$ gram matrix \mathbf{G}
- Lot of work on speeding up SVM in these settings (e.g., can use co-ord. descent for α)

Solving for SVM in the Primal

- Maximizing margin subject to constraints led to the soft-margin formulation of SVM

$$\begin{aligned} \arg \min_{\mathbf{w}, b, \xi} \quad & \frac{\|\mathbf{w}\|^2}{2} + C \sum_{n=1}^N \xi_n \\ \text{subject to} \quad & y_n(\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \dots, N \end{aligned}$$

- Note that slack ξ_n is the same as $\max\{0, 1 - y_n(\mathbf{w}^\top \mathbf{x}_n + b)\}$, i.e., hinge loss for (\mathbf{x}_n, y_n)
- Thus the above is equivalent to minimizing the ℓ_2 regularized hinge loss

$$\mathcal{L}(\mathbf{w}, b) = \sum_{n=1}^N \max\{0, 1 - y_n(\mathbf{w}^\top \mathbf{x}_n + b)\} + \frac{\lambda}{2} \mathbf{w}^\top \mathbf{w}$$

- Sum of slacks is like sum of hinge losses, C and λ play similar roles
- Can learn (\mathbf{w}, b) directly by minimizing $\mathcal{L}(\mathbf{w}, b)$ using (stochastic) (sub)grad. descent
 - Hinge-loss version preferred for linear SVMs, or with other regularizers on \mathbf{w} (e.g., ℓ_1)

SVM: Summary

- A hugely (perhaps the most!) popular classification algorithm
- Reasonably mature, highly optimized SVM softwares freely available (perhaps the reason why it is more popular than various other competing algorithms)
- Some popular ones: libSVM, LIBLINEAR, sklearn also provides SVM
- Lots of work on scaling up SVMs* (both large \mathbf{N} and large \mathbf{D})
- Extensions beyond binary classification (e.g., multiclass, structured outputs)
- Can even be used for regression problems (Support Vector Regression)
- Nonlinear extensions possible via kernels

* See: “Support Vector Machine Solvers” by Bottou and Lin