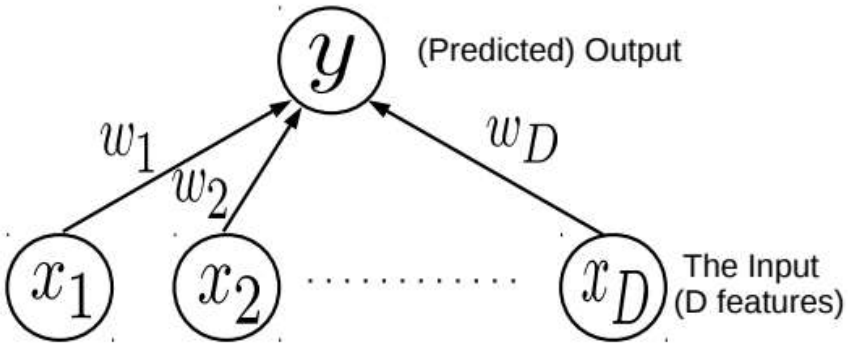
Linear Models and Learning via Optimization

PHAM VAN KHANH

Linear Models

- Suppose we want to learn to map inputs $\boldsymbol{x} \in \mathbb{R}^D$ to real-valued outputs $\boldsymbol{y} \in \mathbb{R}$
- Linear model: Assume output to be a linear weighted combination of the D input features

$$y = \sum_{d=1}^{D} w_d x_d = \boldsymbol{w}^\top \boldsymbol{x}$$



This defines a linear model with D parameters given by a "weight vector" $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$

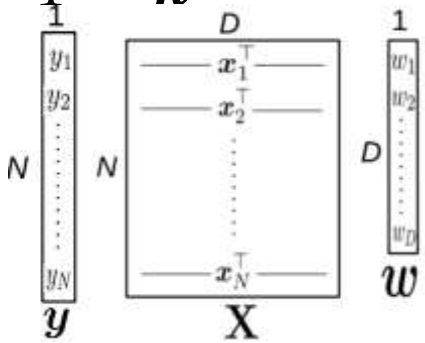
Each of these weights have a simple interpretation: $\boldsymbol{w_d}$ is the "weight" or contribution of the $\boldsymbol{d^{th}}$ feature in making this prediction

The "optimal" weights are unknown and have to be learned by solving an optimization problem, using some **training data**

- This simple model can be used for Linear Regression
- This simple model can also be used as a "building block" for more complex models, e.g.,
  - Classification (binary/multiclass/multi-output/multi-label) and various other ML/deep learning models

Linear Regression

- Given: Training data with N input-output pairs $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^{N}$, $\boldsymbol{x}_n \in \mathbb{R}^D$, $y_n \in \mathbb{R}$



- Goal: Learn a model to predict the output for new test in
- Assume the function that approximates the I/O re
linear model $y_n \approx f(\boldsymbol{x}_n) = \boldsymbol{w}^\top \boldsymbol{x}_n \quad (n = 1,2,\ldots,N)$

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

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

otal error or loss of this mo ning

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

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

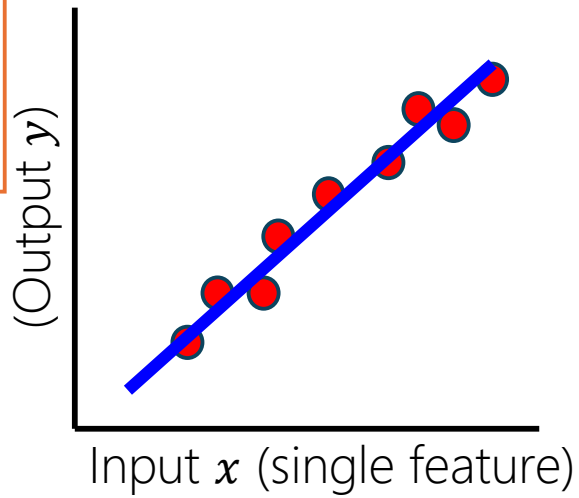
$\ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n)$ measures the prediction error or "loss" or "deviation" of the model on a single training input $(\boldsymbol{x}_n, y_n)$

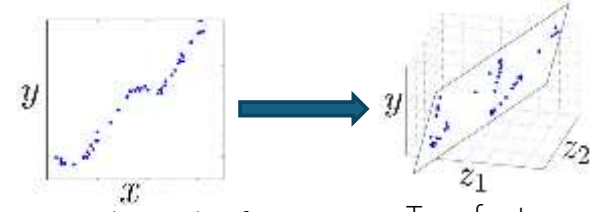
Linear Regression: Pictorially

- 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?

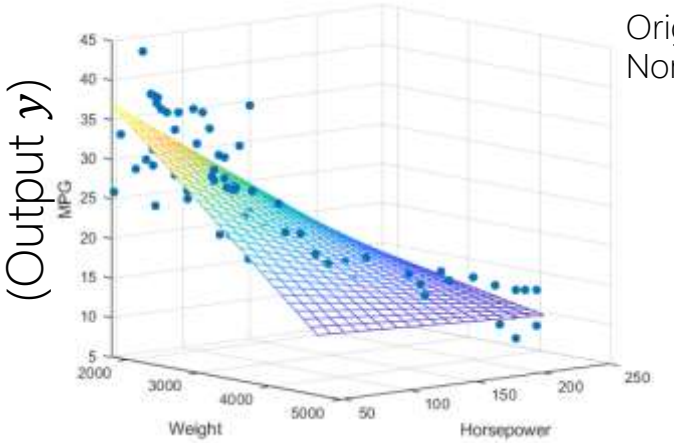
Do linear models become useless in such cases?





Original (single) feature
Nonlinear curve needed

Two features
Can fit a plane (linear)



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

$$y \approx \boldsymbol{w}^{\top}\boldsymbol{\phi}(\boldsymbol{x})$$

The transformation $\boldsymbol{\phi}(.)$ 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 of the unseen (test) inputs well

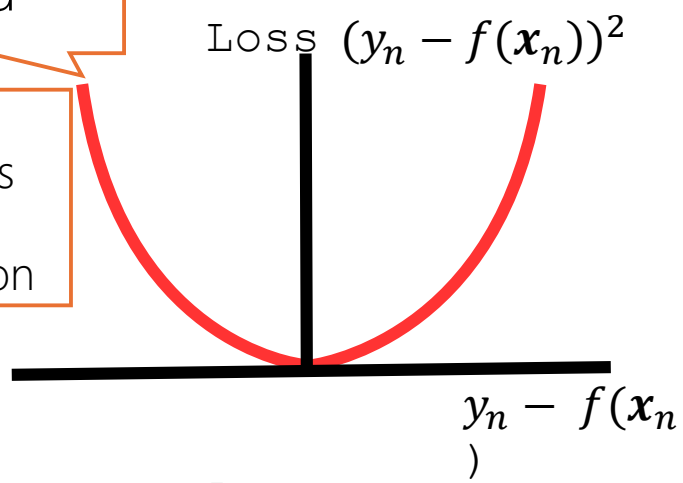
Loss Functions for Regre

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

- Many possible loss functions for regression prob

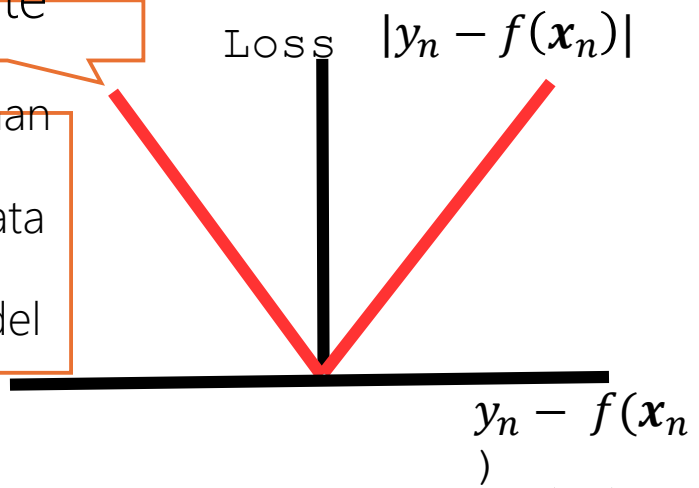
Squared loss

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



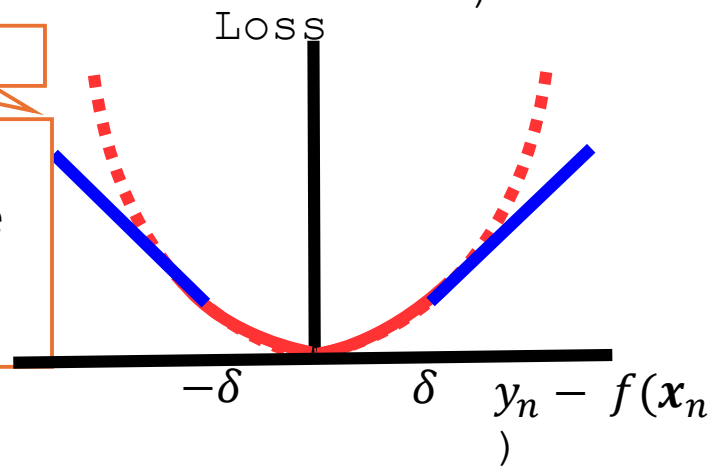
Absolute loss

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



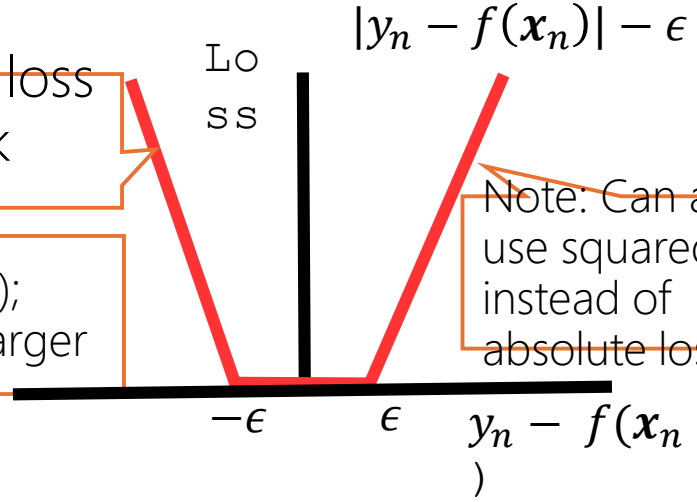
Huber loss

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)

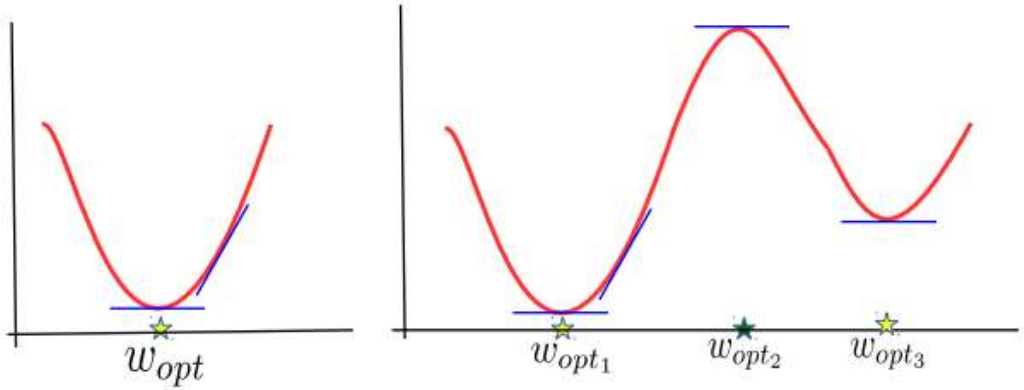
Zero loss for small errors (say up to ϵ); absolute loss for larger errors



Note: Can also use squared loss instead of absolute loss

Minimizing Loss Func using First-Order Optimality

- Use basic calculus to find the minima



Called "first order" since only gradient is used and gradient provides the first order info about the function being optimized

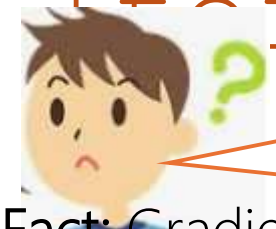
The approach works only for very simple problems where the objective is convex and there are no constraints on the values $\boldsymbol{w}$ can take

- First order optimality: The gradient $\boldsymbol{g}$ must be equal to zero at the optima

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}}[L(\boldsymbol{w})] = \mathbf{0}$$

- Sometimes, setting $\boldsymbol{g} = \mathbf{0}$ and solving for $\boldsymbol{w}$ gives a closed form solution
- If closed form solution is not available, the gradient vector $\boldsymbol{g}$ can still be used in iterative optimization algos, like gradient descent (GD)

Minimizing Loss Func. using Iterative Optimization



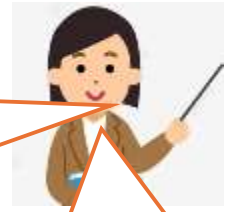
Can I used this approach to solve maximization problems?

For max. problems we can use gradient ascent

$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta_t \boldsymbol{g}^{(t)}$

Will move in the direction of the gradient

Iterative since it requires several steps/iterations to find the optimal solution



Fact: Gradient gives the direction of steepest change in function's value

For convex functions, GD will converge to the global minima

Good initialization needed for non-convex

Gradient Descent

- Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
- For iteration $t = 0,1,2, \ldots$ (or until convergence)
  - Calculate the gradient $\boldsymbol{g}^{(t)}$ using the current iterates $\boldsymbol{w}^{(t)}$
  - Set the learning rate η_t
  - Move in the opposite direction of gradient

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

The learning rate very imp. Should be set carefully (fixed or chosen adaptively). Will discuss some strategies later

Sometimes may be tricky to to assess convergence. Will see some methods later

Linear Regression with Squared Loss

- In this case, the loss func will be

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

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

- Let us find the $\boldsymbol{w}$ that optimizes (minimizes) the above squared loss
- Let's use first order optimality

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

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

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

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

Closed form solutions to ML problems are rare.

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

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

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

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

Chain rule of calculus

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

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

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

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

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

Problem(s) with the Solution!

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

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

- Problem: The matrix $\boldsymbol{X}^\top \boldsymbol{X}$ may not be invertible
  - This may lead to non-unique solutions for $\boldsymbol{w}_{opt}$
- Problem: Overfitting since we only minimized loss defined on training data
  - Weights $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$ may become arbitrarily large to fit the data perfectly
  - Such weights may perform poorly on the test data however
- One Solution: Minimize a **regularized objective** $L(\boldsymbol{w}) + \lambda$

$R(\boldsymbol{w})$ is called the Regularizer and measures the "magnitude" of $\boldsymbol{w}$

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

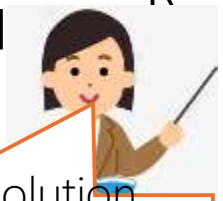
Regularized Least Squares (a.k.a. Ridge Regression)

- Recall that the regularized objective is of the form $L_{reg}(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})$

- One possible/popular regularizer: the squared Euclidean (ℓ_2 squared) norm of $\boldsymbol{w}$

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

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 DxD matrix $\boldsymbol{X}^\top \boldsymbol{X}$(like adding a ridge/mountain to some land)

- this regularizer, we have the regularized least

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

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

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

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

Other Ways to Control Over

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

ℓ_1 norm regularizer

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

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

ℓ_0 norm regularizer (counts number of nonzeros in $\boldsymbol{w}$

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)

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

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

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

Using such regularizers gives a sparse weight vector $\boldsymbol{w}$ as solution

sparse means many entries in $\boldsymbol{w}$ 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 w
  - 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

Gradient Descent for Linear/Ridge Regression

- Just use the GD algorithm with the gradient expressions we derived ☺
- Iterative updates for linear regression will be of the form

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

$$= \boldsymbol{w}^{(t)} + \eta_t \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n$$

Unlike the closed form solution of least squares regression, here we have iterative updates but do not require the expensive matrix inversion of the $D \times D$ matrix $\boldsymbol{X}^\top \boldsymbol{X}$

- Similar updates for ridge regression as well (with the gradient expression being slightly different; left as an exercise)

- More on iterative optimization methods later

Gradient Descent for Linear/Ridge Regression

- Just use the GD algorithm with the gradient expressions we derived
- Iterative updates for linear regression will be of the for

Also, we usually work with average gradient so the gradient term is divided by N

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

Note the form of each term in the gradient expression update: Amount of current $\boldsymbol{w}'$ s error on the n^{th} training example multiplied by the input $\boldsymbol{x}_n$

Unlike the closed form solution $(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{y}$ of least squares regression, here we have iterative updates but do not require the expensive matrix inversion of the $D \times D$ matrix $\boldsymbol{X}^\top \boldsymbol{X}$

$$= \boldsymbol{w}^{(t)} + \eta_t \sum_{n=1}^{N} \left(y_n - \boldsymbol{w}^{(t)\top} \boldsymbol{x}_n \right) \boldsymbol{x}_n$$

- Similar updates for ridge regression as well (with the gradient expression being slightly different; left as an exercise)

- More on iterative optimization methods later

ℓ_2 regularization and "Smoothness"

- The regularized objective we minimized is

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

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

- Minimizing $L_{reg}(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$ gives a solution for $\boldsymbol{w}$ that
  - Keeps the training error small
  - Has a small ℓ_2 squared norm $\boldsymbol{w}^\top \boldsymbol{w} = \sum_{d=1}^{D}$ [illegible]

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

- Small entries in $\boldsymbol{w}$ are good since they lead to "smooth" [illegible]

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

$\boldsymbol{x}_n =$ | 1.2 | 0.5 | 2.4 | 0.3 | 0.8 | 0.1 | 0.9 | 2.1 | $y_n = 0.8$

$\boldsymbol{x}_m =$ | 1.2 | 0.5 | 2.4 | 0.3 | 0.8 + ϵ | 0.1 | 0.9 | 2.1 | $y_m = 100$

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

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

A typical $\boldsymbol{w}$ 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 Over

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

ℓ_1 norm regularizer

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

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

ℓ_0 norm regularizer (counts number of nonzeros in $\boldsymbol{w}$

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)

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

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

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

Using such regularizers gives a sparse weight vector $\boldsymbol{w}$ as solution (will see the reason in detail later)

sparse means many entries in $\boldsymbol{w}$ 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 w
  - 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

Linear Regression as Solving System of Linear Eqs

- The form of the lin. reg. model $\boldsymbol{y} \approx \boldsymbol{Xw}$ is akin to a system of linear equation
- Assuming N training examples with D features each, we have

First training example: $y_1 = x_{11}w_1 + x_{12}w_2 + \ldots + x_{1D}w_D$

Second training example: $y_2 = x_{21}w_1 + x_{22}w_2 + \ldots + x_{2D}w_D$

$\vdots$

N-th training example: $y_N = x_{N1}w_1 + x_{N2}w_2 + \ldots + x_{ND}w_D$

Note: here x_{nd} denotes the d^{th} feature of the n^{th} training example

N equations and D unknowns here $(w_1, w_2, \ldots, w_D)$

- Usually we will either have $N > D$ or $N < D$
  - Thus we have an underdetermined ($N < D$) or overdetermined ($N > D$) system
  - Methods to solve over/underdetermined systems can be used for lin reg as well
  - Many of these methods don't require expensive matrix inversion

Solving lin reg as system of lin eq.

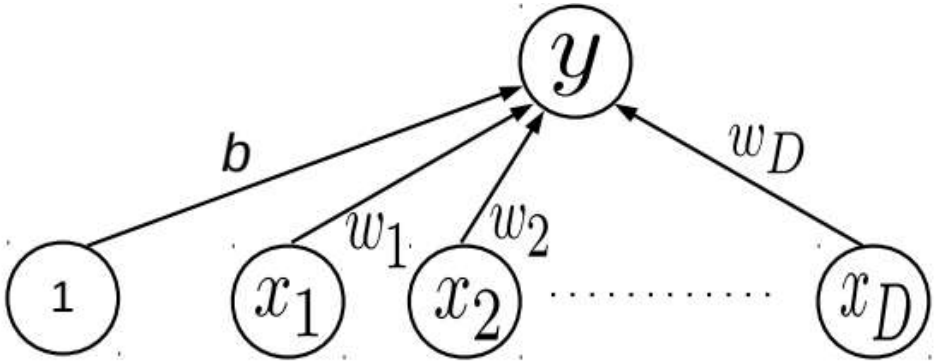
$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$ → $\boldsymbol{Aw} = \boldsymbol{b}$ where $\boldsymbol{A} = \boldsymbol{X}^\top \boldsymbol{X}$, and $\boldsymbol{b} =$

Now solve this!

System of lin. Eqns with D equations and D unknowns

The bias term

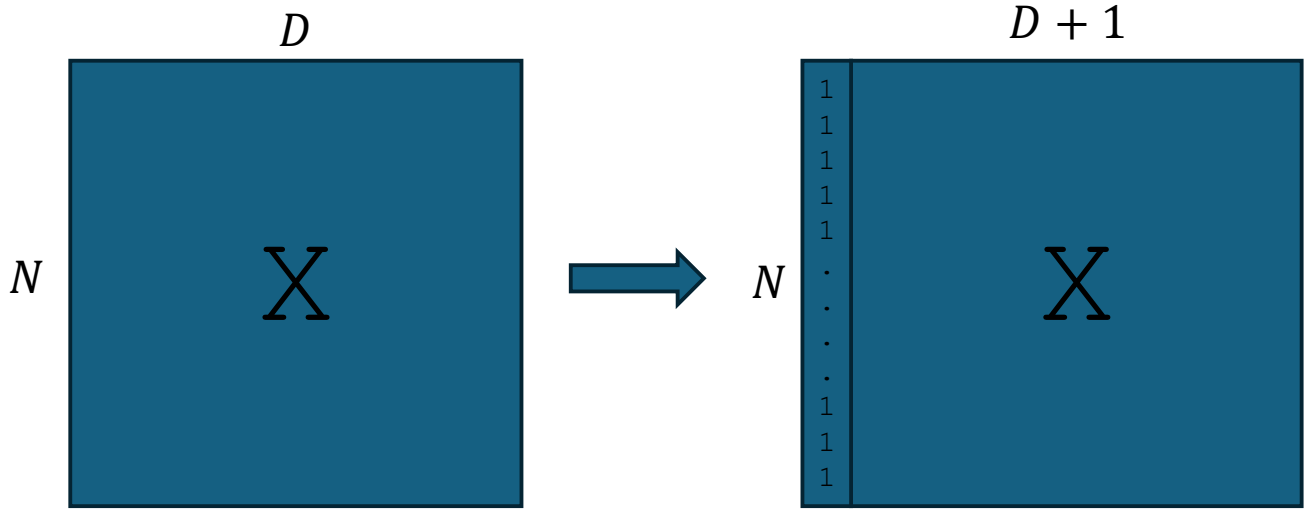
- Linear models usually also have a bias term $\boldsymbol{b}$ in addition to the weights



$$y = \sum_{d=1}^{D} w_d x_d + b = \boldsymbol{w}^\top \boldsymbol{x} + b$$

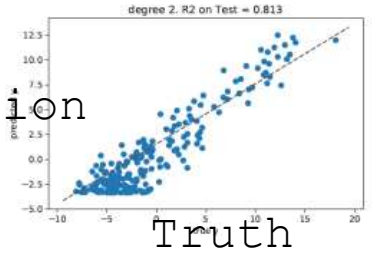
Can append a constant feature "1" for each input and again rewrite as $y = \tilde{\boldsymbol{w}}^\top \tilde{\boldsymbol{x}}$ where now both $\tilde{\boldsymbol{x}} = [1, \boldsymbol{x}]$ and $\tilde{\boldsymbol{w}} = [b, \boldsymbol{w}]$ are in $\mathbb{R}^{D+1}$

We will assume the same and omit the explicit bias for simplicity of notation



Evaluation Measures for Regression Models

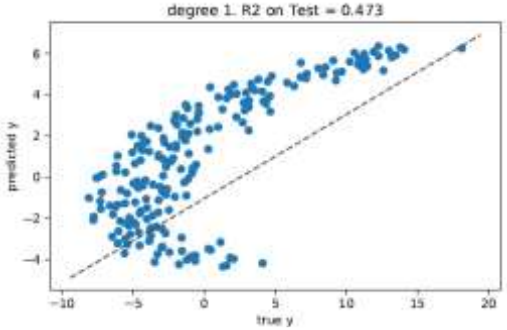
- Plotting the prediction $\hat{y}_n$ vs truth y_n for the validation/test s

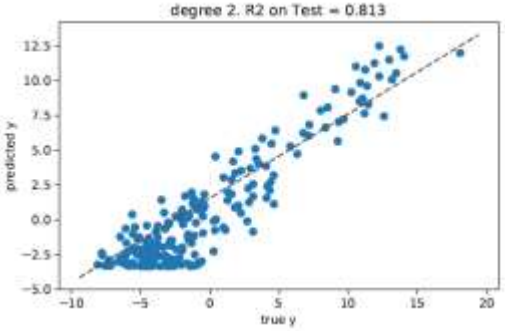


- Mean Squared Error (MSE) and Mean Absolute Error (MAE) on val./test set

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 \qquad MAE = \frac{1}{N}\sum_{n=1}^{N}|y_n - \hat{y}_n|$$

Plots of true vs predicted outpu and R^2 for two regression mod





- RMSE (Root Mean Squared Error) $\triangleq \sqrt{MSE}$
- Coefficient of determination or R^2

$$R^2 = 1 - \frac{\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2}$$

"relative" error w.r.t. a model that makes a constant prediction $\bar{y}$ for all inputs

A "base" model that always predicts the mean $\bar{y}$ will have $R^2 = 0$ and the perfect model will have $R^2 = 1$. Worse than base models can even have negative R^2

$\bar{y}$ is empirical mean of true responses, i.e., $\frac{1}{N}\sum_{n=1}^{N} y_n$

Pic from MLAPP (Murphy)

Linear Models for Classification

Linear Models for Classification

- A linear model $y = \boldsymbol{w}^\top \boldsymbol{x}$ can also be used in classification
- For binary classification, can treat $\boldsymbol{w}^\top \boldsymbol{x}_n$ as the "score" of input $\boldsymbol{x}_n$ and eith
  - Threshold the score to get a binary label

$$y_n = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

Large positive score means positive label, otherwise negative label

Note that $\log \frac{\mu_n}{1-\mu_n} = \boldsymbol{w}^\top \boldsymbol{x}_n$ (the score) is also called the log-odds ratio, and often also logits

  - Convert the score into a probability

$$\mu_n = p(y = 1|\boldsymbol{x}_n, \boldsymbol{w}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

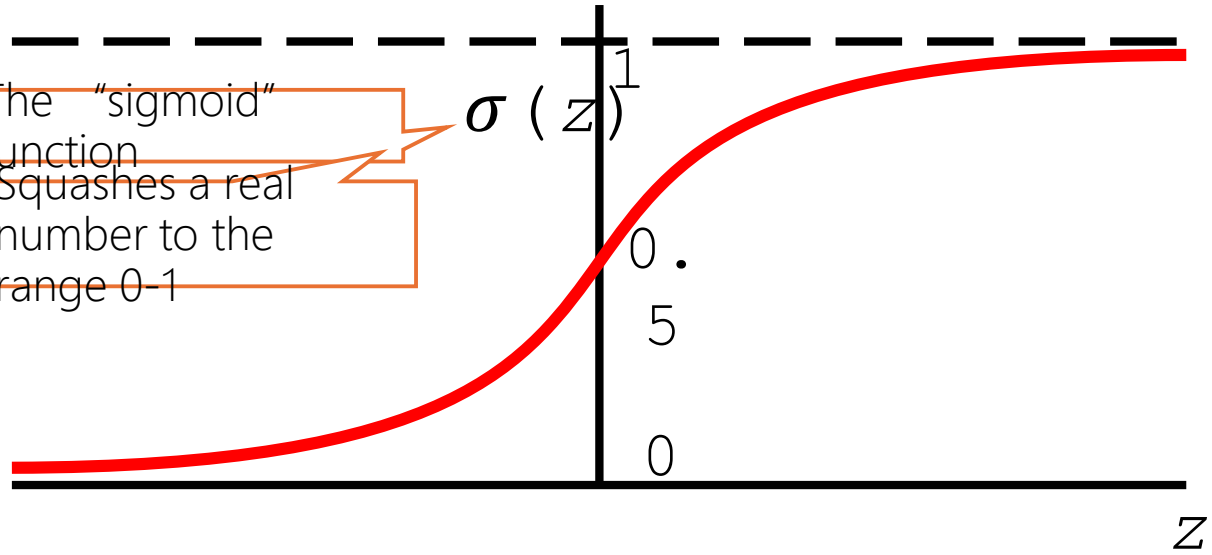
$$= \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x}_n)}$$

$$= \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}{1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}$$

Popularly known as "logistic regression" (LR) model (misnomer: it is not a regression model but a classification model), a probabilistic model for binary classification

The "sigmoid" function

Squashes a real number to the range 0-1

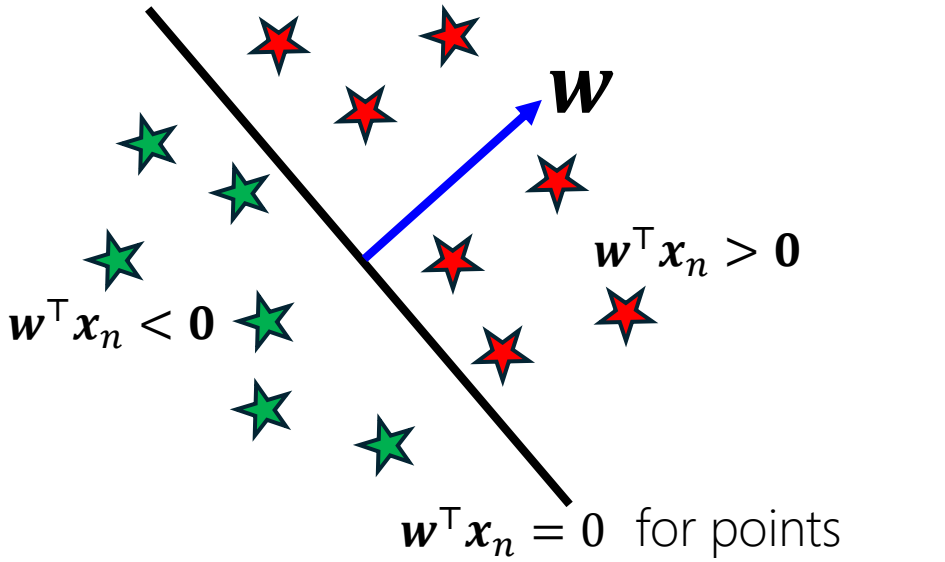


- Note: In LR, if we assume the label y_n as -1/+1 (not 0/1) then we can write

$$p(y_n|\boldsymbol{w}, \boldsymbol{x}_n) = \frac{1}{1 + \exp(-y_n \boldsymbol{w}^\top \boldsymbol{x}_n)} = \sigma(y_n \boldsymbol{w}^\top \boldsymbol{x}_n)$$

Linear Models: The Decision Boundary

- Decision boundary is where the score $\boldsymbol{w}^\top \boldsymbol{x}_n$ changes its sign



- Decision boundary is where both classes have equal probability for the input $\boldsymbol{x}_n$
- For logistic reg. at decision boundary

$$p(y_n = 1|\boldsymbol{w}, \boldsymbol{x}_n) = p(y_n = 0|\boldsymbol{w}, \boldsymbol{x}_n)$$

$$\frac{\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}{1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)} = \frac{1}{1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}$$

$$\exp(\boldsymbol{w}^\top \boldsymbol{x}_n) = 1$$

$$\boldsymbol{w}^\top \boldsymbol{x}_n = 0$$

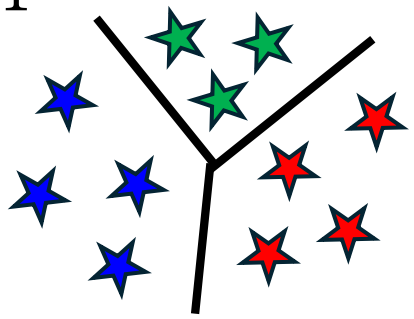
- Therefore, both views are equivalent

Linear Models for (Multi-class) Classification

- If there are $K > 2$ classes, we use K weight vectors $\{\boldsymbol{w}_i\}_{i=1}^{K}$ to define the model

$D \times K$ weight matrix

$$\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$$



- The prediction rule is as follows

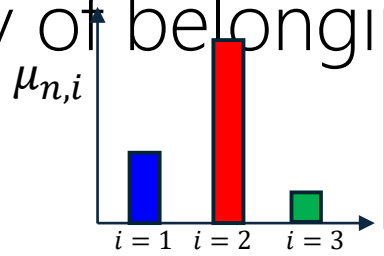
$$y_n = \text{argmax}_{i \in \{1,2,\ldots,K\}} \boldsymbol{w}_i^\top \boldsymbol{x}_n$$

- Can think of $\boldsymbol{w}_i^\top \boldsymbol{x}_n$ as the score/similarity of the input w.r.t. the i^{th} class
- Can also use these scores to compute the probability of belonging to each class

$$\mu_{n,i} = p(y_n = i | \boldsymbol{W}, \boldsymbol{x}_n) = \frac{\exp(\boldsymbol{w}_i^\top \boldsymbol{x}_n)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^\top \boldsymbol{x}_n)}$$

"softmax" classification

Multi-class extension of logistic regression



Note: Just like logistic regression, the scores $\boldsymbol{w}_i^\top \boldsymbol{x}_n$ are called logits (K logits in this case)

Probability of $\boldsymbol{x}_n$ belonging to class i

$$\boldsymbol{\mu}_n = [\mu_{n,1}, \mu_{n,2}, \ldots, \mu_{n,K}]$$

Vector of probabilities of $\boldsymbol{x}_n$ belonging to each of the K classes

Class i with largest $\boldsymbol{w}_i^\top \boldsymbol{x}_n$ has the largest probability

$$\sum_{i=1}^{K} \mu_{n,i} = 1$$

Probabilities must sum to 1

Note: We actually need only $K - 1$ weight vectors in softmax classification. Think why?

Linear Classification: Interpreting weight vectors

- Recall that multi-class classification prediction rule is

$$y_n = \text{argmax}_{i \in \{1,2,\ldots,K\}} \boldsymbol{w}_i^\top \boldsymbol{x}_n$$

- Can think of $\boldsymbol{w}_i^\top \boldsymbol{x}_n$ as the score of the input for the i^{th} class (or similarity of $\boldsymbol{x}_n$ with $\boldsymbol{w}_i$)

- Once learned (we will see the methods later), these K weight vectors (one for each class) can sometimes have nice interpretations, especially when the inputs

The learned weight vectors of each of the 4 classes "unflattened" and visualized as images – they kind of look like a "average" of what the images from that class should look like

$\boldsymbol{w}_{car}$

$\boldsymbol{w}_{frog}$

$\boldsymbol{w}_{horse}$

$\boldsymbol{w}_{cat}$

That's why the dot product of each of these weight vectors with an image from the correct class will be expected to be the largest

These images sort of look like class prototypes if I were using LwP ☺

Yeah, "sort of". ☺ No wonder why LwP (with Euclidean distances) acts like a linear model. ☺

Loss Functions for Classification

- Assume true label to be $y_n \in \{0,1\}$ and the score of a linear model to be $\boldsymbol{w}^\top \boldsymbol{x}_n$
- One possibility is to use squared loss just like we used in regression

$$l(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n) = (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

- Will be easy to optimize (same solution as the regression case)
- Can also consider other loss functions used in regression
  - Basically, pretend that the binary label is actually a continuous value and treat the problem as regression where the output can only be one of two possible values
- However, regression loss functions aren't ideal since y_n is discrete (binary/categorical)
- Using the score $\boldsymbol{w}^\top \boldsymbol{x}_n$ or the probability $\mu_n = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n)$ of belonging to the positive class, we have specialized loss function for binary classification

Loss Functions for Classification: Cross-Entropy

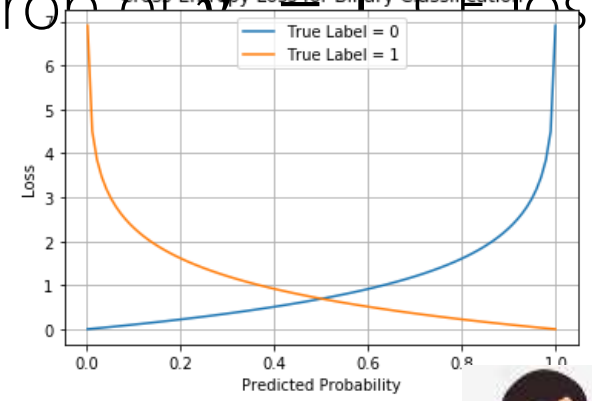
- Cross-entropy (CE) is a popular loss function for binary classification. Used in logistic reg.

- Assuming true $y_n \in \{0,1\}$ and $\mu_n = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n)$ as predicted prob. of $y_n = 1$, CE loss is

$$L(\boldsymbol{w}) = -\left[\sum_{n=1}^{N} y_n \log \mu_n + (1 - y_n)\log(1 - \mu_n)\right]$$

Very large loss if y_n is 1and μ_n close to 0, or y_n is 0and μ_n close to 1

This is precisely what we want from a good loss function for binary classification



- For multi-class classification, the CE loss is defined as

$$L(\boldsymbol{W}) = -\sum_{n=1}^{N}\sum_{i=1}^{K} y_{n,i} \log \mu_{n,i}$$

CE loss is also convex in $\boldsymbol{w}$ (can prove easily using definition of convexity; will see later). Therefore unique solution is obtained when we minimize it

$y_{n,i} = 1$ if true label of $\boldsymbol{x}_n$ is class i and 0 otherwise. $\mu_{n,i}$ is the predicted probability of $\boldsymbol{x}_n$ belonging to class i

Note: Sometimes we divide the loss function (not just CE but others too like squared loss) by the number of training examples N(doesn't make a difference to the solution; just a scaling factor. All relevant quantities, such as gradients will also get divided by N

Cross-Entropy Loss: The Gradient

- The expression for the gradient of binary cross-entropy loss

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = -\sum_{n=1}^{N} (y_n - \mu_n)\, \boldsymbol{x}_n$$

Note the μ_n is a function of $\boldsymbol{w}$

Using this, we can now do gradient descent to learn the optimal $\boldsymbol{w}$ for logistic regression:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

Note the form of each term in the gradient expression: Amount of current $\boldsymbol{w}'$ s error in predicting the label of the n^{th} training example multiplied by the input $\boldsymbol{x}_n$

- The expression for the gradient of multi-class cross-entropy loss

Need to calculate the gradient for each of the K weight vectors

$$\boldsymbol{g}_i = \nabla_{\boldsymbol{w}_i} L(\boldsymbol{W}) = -\sum_{n=1}^{N} (y_{n,i} - \mu_{n,i})\, \boldsymbol{x}_n$$

Using these gradients, we can now do gradient descent to learn the optimal $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$ For the softmax classification model

Note the form of each term in the gradient expression: Amount of current $\boldsymbol{W}'$ s error in predicting the label of the n^{th} training example multiplied by the input $\boldsymbol{x}_n$

Linear Models for Classification

- A linear model $y = \boldsymbol{w}^\top \boldsymbol{x}$ can also be used in classification
- For binary classification, can treat $\boldsymbol{w}^\top \boldsymbol{x}_n$ as the "score" of input $\boldsymbol{x}_n$ and eith
  - Threshold the score to get a binary label

$$y_n = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

Large positive score means positive label, otherwise negative label

Note that $\log \frac{\mu_n}{1-\mu_n} = \boldsymbol{w}^\top \boldsymbol{x}_n$ (the score) is also called the log-odds ratio, and often also logit

  - Convert the score into a probability

$$\mu_n = p(y = 1|\boldsymbol{x}_n, \boldsymbol{w}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n)$$

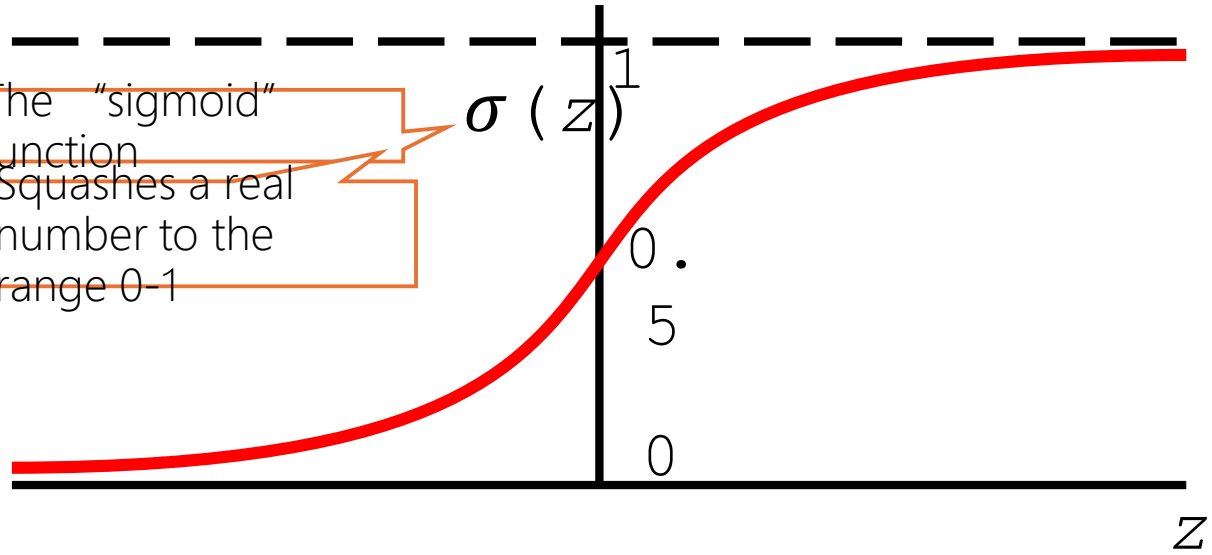
$$= \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x}_n)}$$

$$= \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}{1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}$$

Popularly known as "logistic regression" (LR) model (misnomer: it is not a regression model but a classification model), a probabilistic model for binary classification

The "sigmoid" function

Squashes a real number to the range 0-1



- Note: In LR, if we assume the label y_n as -1/+1 (not 0/1) then we can write

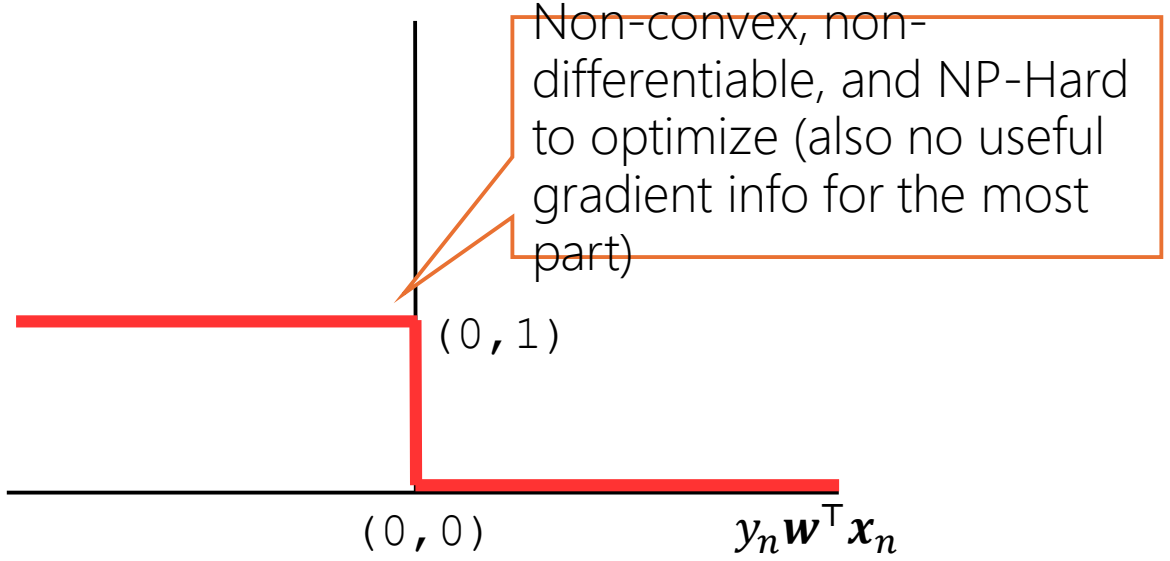
$$p(y_n|\boldsymbol{w}, \boldsymbol{x}_n) = \frac{1}{1 + \exp(-y_n \boldsymbol{w}^\top \boldsymbol{x}_n)} = \sigma(y_n \boldsymbol{w}^\top \boldsymbol{x}_n)$$

Some Other Loss Functions for Binary Classification

- Assume true label as y_n and prediction as $\hat{y}_n = \mathbf{sign}[\boldsymbol{w}^\top \boldsymbol{x}_n]$

- The zero-one loss is the most natural loss function for classification

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

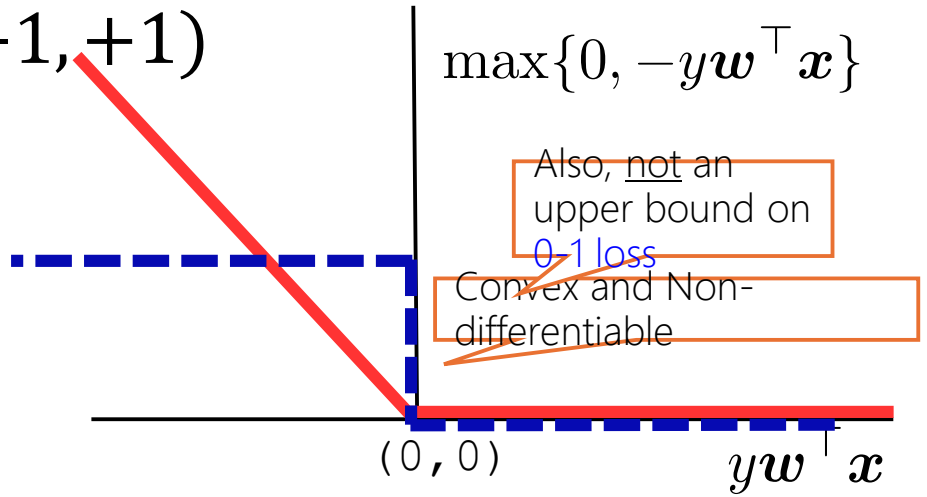


- Since zero-one loss is hard to minimize, we use some surrogate loss function
  - Popular examples: Cross-entropy (same as logistic loss), hinge loss , etc
  - Note: Ideally, surrogate loss (approximation of zero-one) must be an upper

Some Other Loss Func for Binary Classification

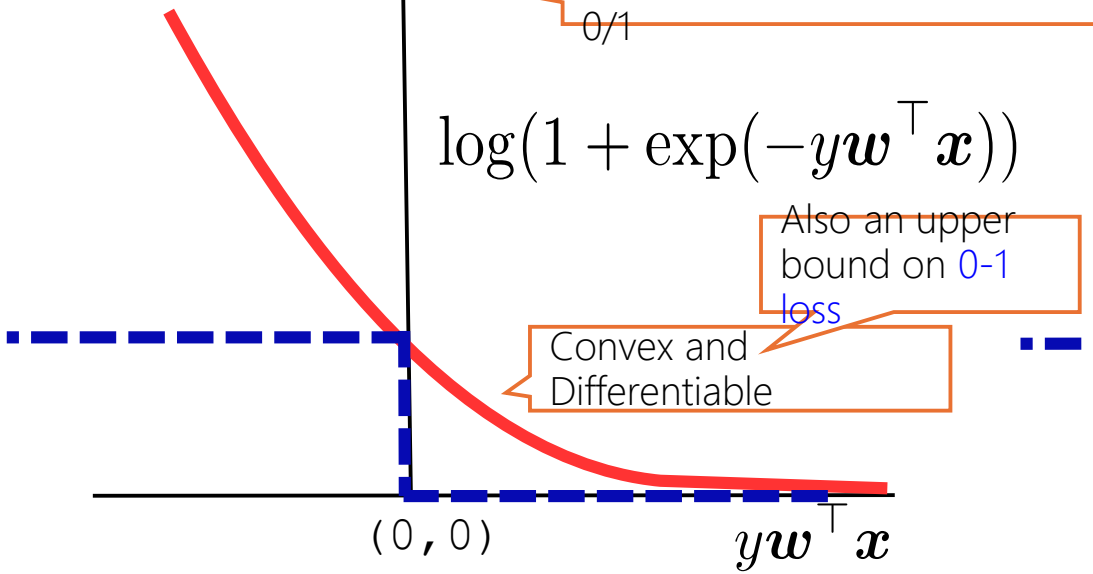
- For an ideal loss function, assuming $y_n \in (-1, +1)$
  - Large positive $y_n \boldsymbol{w}^\top \boldsymbol{x}_n \Rightarrow$ small/zero loss
  - Large negative $y_n \boldsymbol{w}^\top \boldsymbol{x}_n \Rightarrow$ large/non-zero loss
  - Small (large) loss if predicted probability of the the true label is large (small)

"Perceptron" Loss



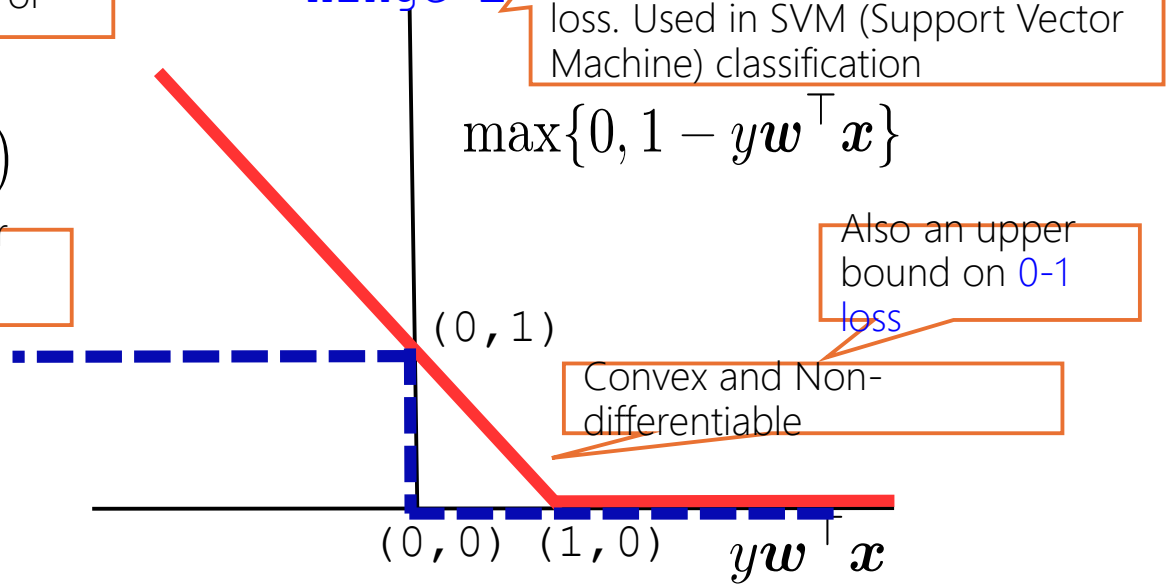
Log(istic)

Same as cross-entropy loss (logistic reg.) if we assume labels to be -1/+1 instead of 0/1



Hinge Lo

Very popular like cross-entropy loss. Used in SVM (Support Vector Machine) classification



Evaluation Measures for Binary Classification

- Average classification error or average accuracy (on val./test data)

$$err(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}[y_n \neq \hat{y}_n] \qquad acc(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}[y_n = \hat{y}_n]$$

- The cross-entropy loss itself (on val./test data)
- Precision, Recall, and F1 score (preferred if labels are imbalanced)
  - Precision (P): Of positive predictions by the model, what fraction is true positive
  - Recall (R): Of all true positive examples, what fraction the model predicted as positive
  - F1 score: Harmonic mean of P and R
- Confusion matrix is als m

Various other metrics such as error/accuracy, P, R, F1, etc. can be readily calculated from the confusion matrix

Evaluation Measures for Multi-class Classification

- Average classification error or average accuracy (on val./test data)

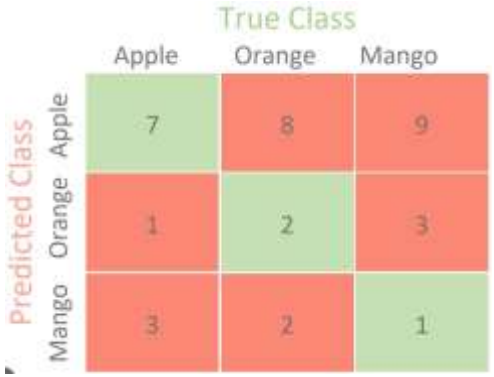
$$err(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}[y_n \neq \hat{y}_n] \qquad acc(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \mathbb{I}[y_n = \hat{y}_n]$$

- Top-k accuracy

$$\text{Top} - \text{k Accuracy} = \frac{1}{N}\sum_{n=1}^{N} \text{is\_correct\_top\_k}[y_n, \hat{S}_n]$$

y_n is the true label, $\hat{S}_n$ is the set of top-k predicted classes for $\boldsymbol{x}_n$ (based on the predicted probabilities/scores of the various classes)

- The cross-entropy loss itself (on val./test data)
- Class-wise Precision, Recall, and F1 score (preferred if labels are imbalanced)
- Confusion matrix



Various other metrics such as error/accuracy, P, R, F1, etc. can be readily calculated from the confusion matrix