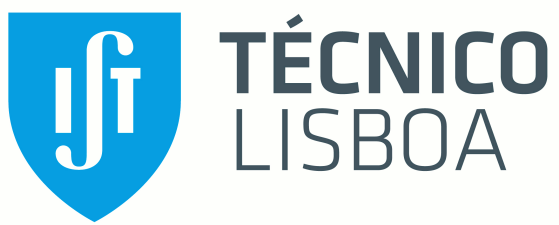
Lecture 4: Linear Models II

André Martins, Francisco Melo, Mário Figueiredo



Deep Learning Course, Fall 2021

Today's Roadmap

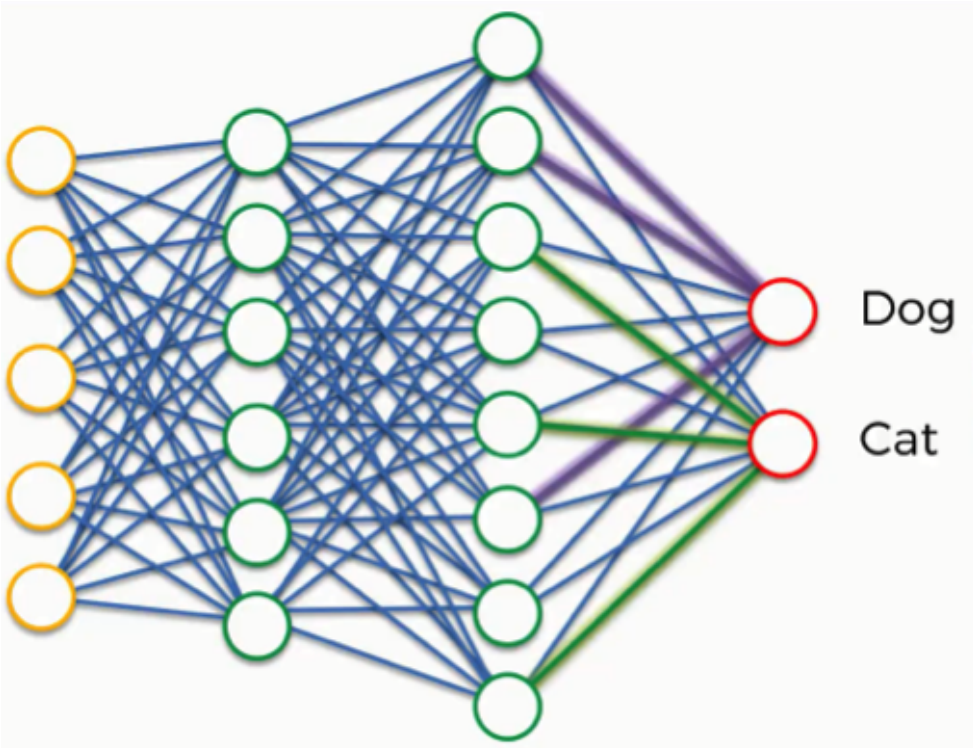
- Logistic regression
- Regularization and optimization
- Stochastic gradient descent.

Why Linear Classifiers?

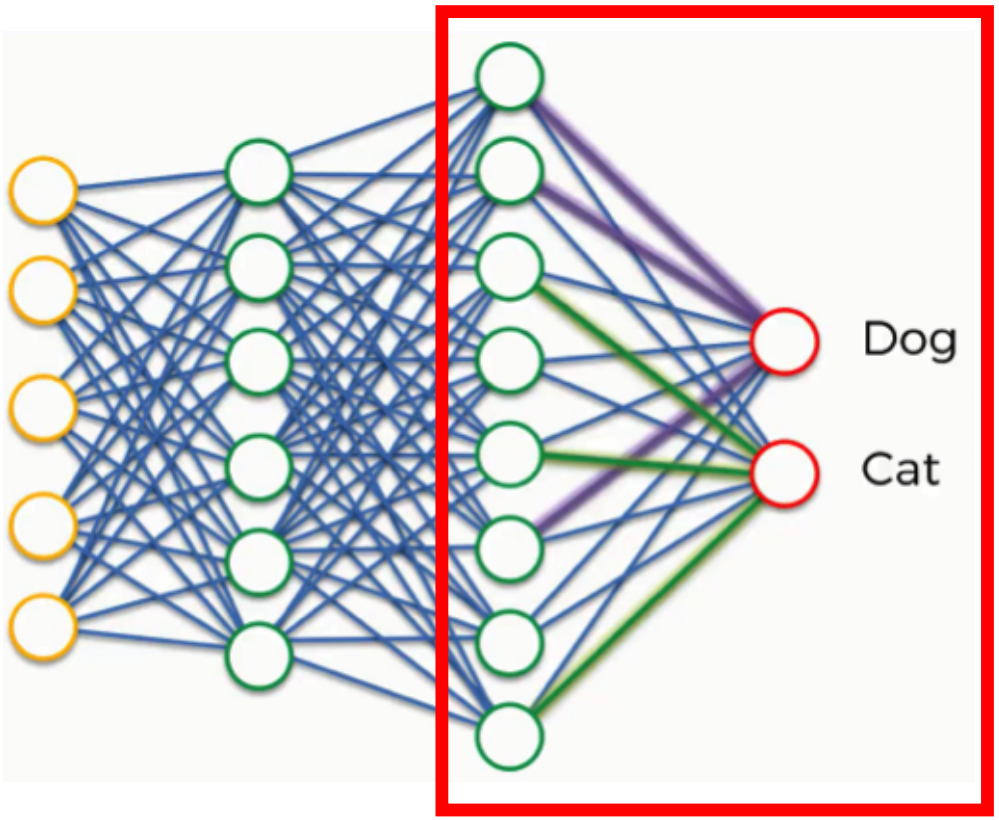
I know the course title promised "deep", but...

- The underlying machine learning concepts are the same
- The theory (statistics and optimization) are much better understood
- Linear classifiers are still widely used (and very effective when data is scarce)
- Linear classifiers are **a component of neural networks**.

Linear Classifiers and Neural Networks

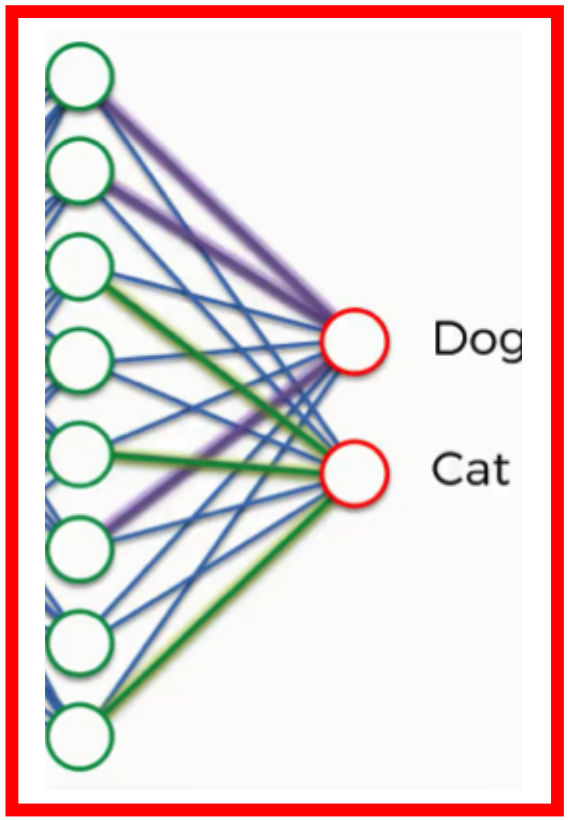


Linear Classifiers and Neural Networks



Linear Classifier

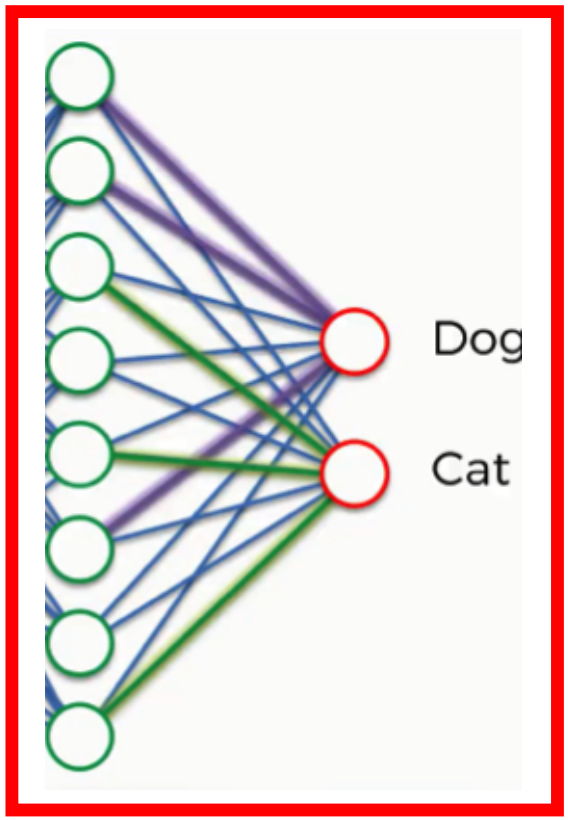
Linear Classifiers and Neural Networks



Linear Classifier

Linear Classifiers and Neural Networks

Handcrafted Features



Linear Classifier

So far

We have covered:

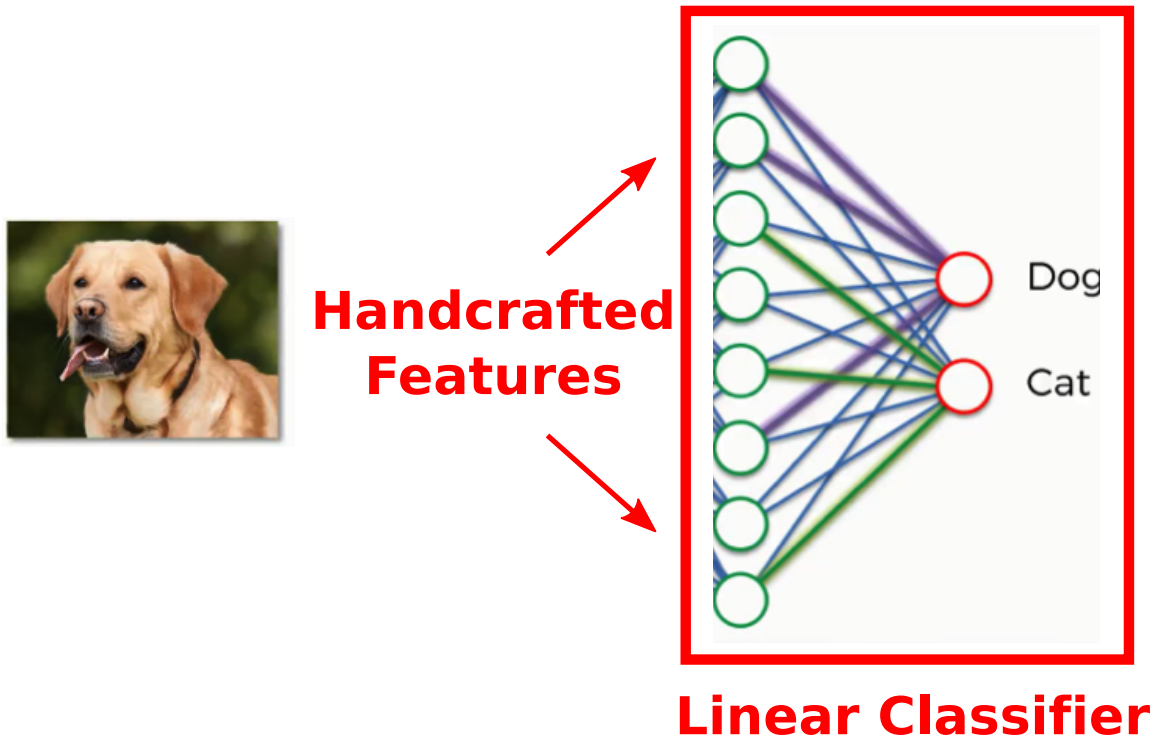
- The perceptron algorithm
- (Multinomial) Naive Bayes.

We saw that both are instances of linear classifiers.

Perceptron finds a separating hyperplane (if it exists), Naive Bayes is a generative probabilistic model

Next: a discriminative probabilistic model.

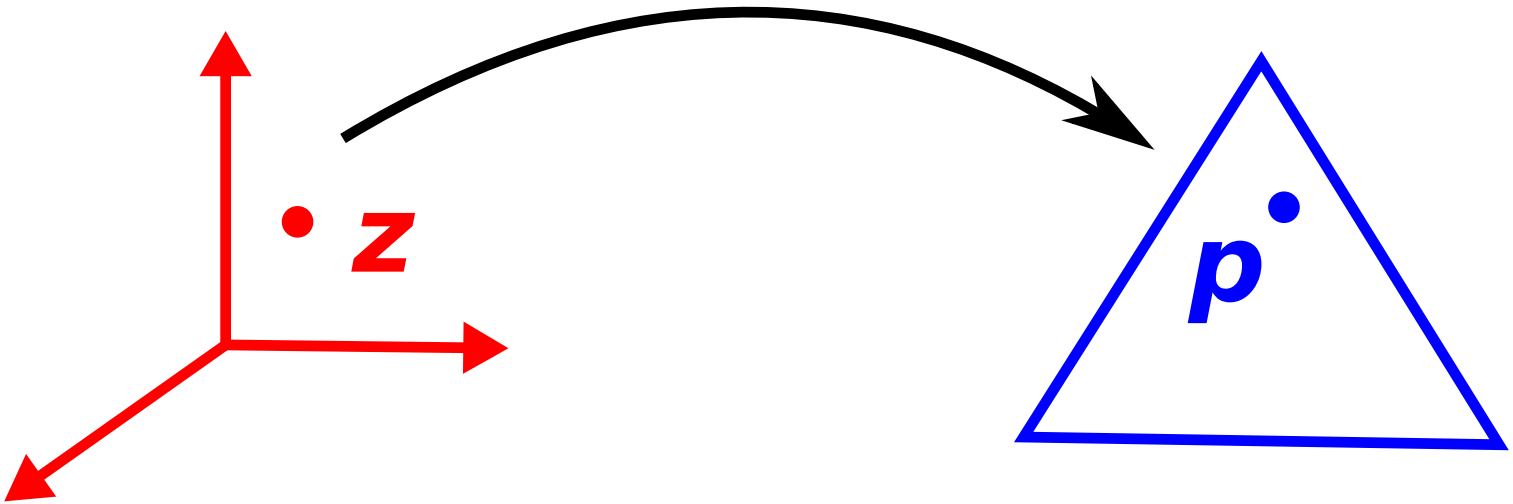
Reminder



$$\widehat{y} = \text{argmax}\left(\boldsymbol{W}\boldsymbol{\phi}(x) + \boldsymbol{b}\right), \quad \boldsymbol{W} = \begin{bmatrix} \vdots \\ \boldsymbol{w}_y^\top \\ \vdots \end{bmatrix}, \; \boldsymbol{b} = \begin{bmatrix} \vdots \\ b_y \\ \vdots \end{bmatrix}.$$

Key Problem

How to map from a set of label scores $\mathbb{R}^{|\mathcal{Y}|}$ to a probability distribution over $\mathcal{Y}$?



We'll see an important mapping: softmax (next).

Outline

Logistic Regression

Recall: a linear model gives the score for each class, $\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x)$.

Define a conditional probability:

$$P(y|x) = \frac{\exp(\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x))}{Z_x}, \qquad \text{where } Z_x = \sum_{y' \in \mathcal{Y}} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x))$$

This operation (exponentiating and normalizing) is called the softmax transformation (more later!)

Note: still a linear classifier

$$\begin{aligned} \arg\max_y \; P(y|x) &= \arg\max_y \; \frac{\exp(\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x))}{Z_x} \\ &= \arg\max_y \; \exp(\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x)) \\ &= \arg\max_y \; \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x) \end{aligned}$$

Binary Logistic Regression

Binary labels ($\mathcal{Y} = \{\pm 1\}$)

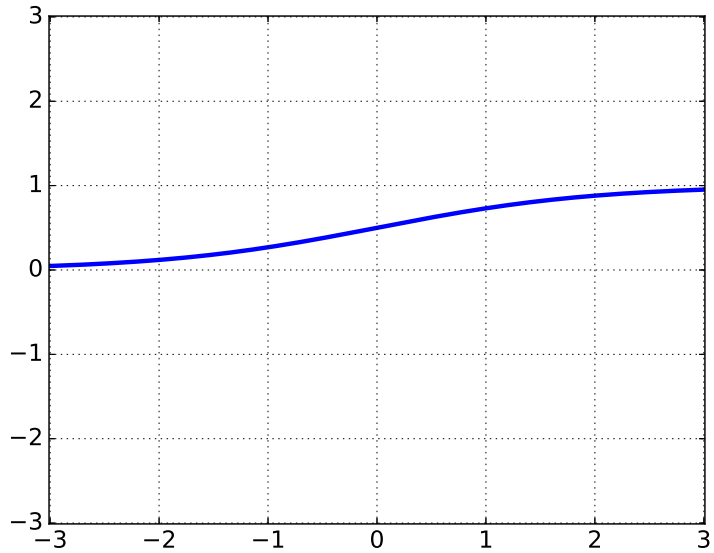
Scores: 0 for negative class, $\boldsymbol{w} \cdot \boldsymbol{\phi}(x)$ for positive class

$$\begin{aligned} P(y = +1 \mid x) &= \frac{\exp(\boldsymbol{w} \cdot \boldsymbol{\phi}(x))}{1 + \exp(\boldsymbol{w} \cdot \boldsymbol{\phi}(x))} \\ &= \frac{1}{1 + \exp(-\boldsymbol{w} \cdot \boldsymbol{\phi}(x))} \\ &= \sigma(\boldsymbol{w} \cdot \boldsymbol{\phi}(x)). \end{aligned}$$

This is called a sigmoid transformation (more later!)

Sigmoid Transformation

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



- Widely used in neural networks (wait for tomorrow!)
- Can be regarded as a 2D softmax
- "Squashes" a real number between 0 and 1
- The output can be interpreted as a probability
- Positive, bounded, strictly increasing

Multinomial Logistic Regression

$$P_{\boldsymbol{W}}(y \mid x) = \frac{\exp(\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x))}{Z_x}$$

- How do we learn weights $\boldsymbol{W}$?
- Set $\boldsymbol{W}$ to maximize the conditional log-likelihood of training data:

$$\widehat{\boldsymbol{W}} = \arg\max_{\boldsymbol{W}} \log \left(\prod_{t=1}^{N} P_{\boldsymbol{W}}(y_t|x_t) \right) = \arg\min_{\boldsymbol{W}} - \sum_{t=1}^{N} \log P_{\boldsymbol{W}}(y_t|x_t) =$$

$$= \arg\min_{\boldsymbol{W}} \sum_{t=1}^{N} \left(\log \sum_{y'_t} \exp(\boldsymbol{w}_{y'_t} \cdot \boldsymbol{\phi}(x_t)) - \boldsymbol{w}_{y_t} \cdot \boldsymbol{\phi}(x_t) \right),$$

i.e., set $\boldsymbol{W}$ to assign as much probability mass as possible to the correct labels!

Logistic Regression

- This objective function is convex
- Therefore any local minimum is a global minimum
- No closed form solution, but lots of numerical techniques
  - Gradient methods (gradient descent, conjugate gradient)
  - Quasi-Newton methods (L-BFGS, ...)

Logistic Regression

- This objective function is convex
- Therefore any local minimum is a global minimum
- No closed form solution, but lots of numerical techniques
  - Gradient methods (gradient descent, conjugate gradient)
  - Quasi-Newton methods (L-BFGS, ...)

- Logistic Regression = Maximum Entropy: maximize entropy subject to constraints on features
- Proof left as an exercise!

Recap: Convex functions

Pro: Guarantee of a global minima ✓

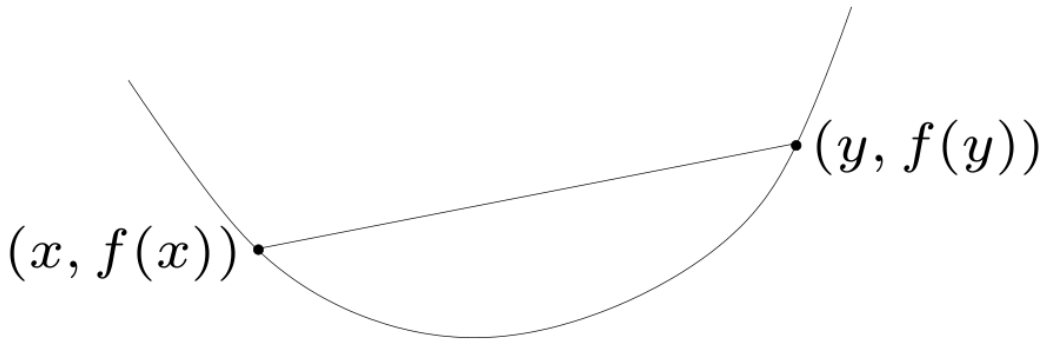


Figure: Illustration of a convex function. The line segment between any two points on the graph lies entirely above the curve.

Recap: Iterative Descent Methods

Goal: find the minimum/minimizer of $f : \mathbb{R}^d \to \mathbb{R}$

- Proceed in **small steps** in the **optimal direction** till a **stopping criterion** is met.
- **Gradient descent**: updates of the form: $x^{(k+1)} \leftarrow x^{(k)} - \eta_k \nabla f(x^{(k)})$

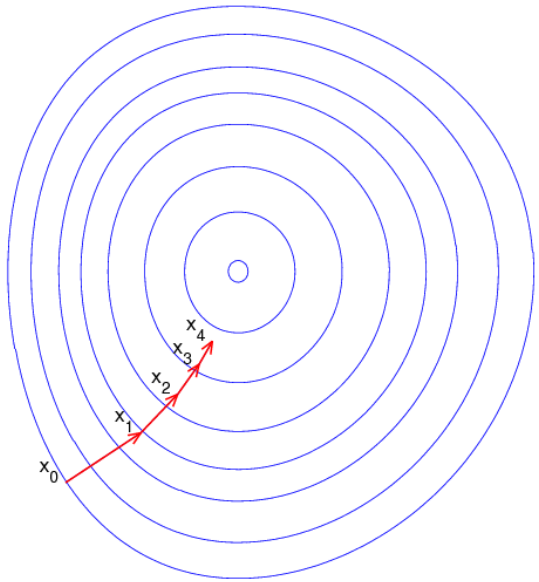


Figure: Illustration of gradient descent. The red lines correspond to steps taken in the negative gradient direction.

Gradient Descent

- Our loss function in logistic regression is

$$L(\boldsymbol{W}; (x, y)) = \log \sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) - \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x).$$

- We want to find $\arg\min_{\boldsymbol{W}} \sum_{t=1}^{N} L(\boldsymbol{W}; (x_t, y_t))$
  - Set $\boldsymbol{W}^0 = \mathbf{0}$
  - Iterate until convergence (for suitable stepsize η_k):

$$\begin{aligned} \boldsymbol{W}^{k+1} &= \boldsymbol{W}^k - \eta_k \nabla_{\boldsymbol{W}} \left(\sum_{t=1}^{N} L(\boldsymbol{W}; (x_t, y_t)) \right) \\ &= \boldsymbol{W}^k - \eta_k \sum_{t=1}^{N} \nabla_{\boldsymbol{W}} L(\boldsymbol{W}^k; (x_t, y_t)) \end{aligned}$$

- $\nabla_{\boldsymbol{W}} L(\boldsymbol{W})$ is gradient of L w.r.t. $\boldsymbol{W}$
- $L(\boldsymbol{W})$ convex $\Rightarrow$ gradient descent will reach the global optimum $\boldsymbol{W}$.

Stochastic Gradient Descent

It turns out this works with a Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set $\boldsymbol{W}^0 = \mathbf{0}$
- Iterate until convergence
  - Pick (x_t, y_t) randomly
  - Update $\boldsymbol{W}^{k+1} = \boldsymbol{W}^k - \eta_k \nabla_{\boldsymbol{W}} L(\boldsymbol{W}^k; (x_t, y_t))$
- i.e. we approximate the true gradient with a noisy, unbiased, gradient, based on a single sample
- Variants exist in-between (mini-batches)
- All guaranteed to find the optimal $\boldsymbol{W}$ (for suitable step sizes)

Computing the Gradient

- For this to work, we need to compute $\nabla_{\boldsymbol{W}} L(\boldsymbol{W}; (x_t, y_t))$, where

$$L(\boldsymbol{W}; (x, y)) = \log \sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) - \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x)$$

- Some reminders:
  1. $\nabla_{\boldsymbol{W}} \log F(\boldsymbol{W}) = \frac{1}{F(\boldsymbol{W})} \nabla_{\boldsymbol{W}} F(\boldsymbol{W})$
  2. $\nabla_{\boldsymbol{W}} \exp F(\boldsymbol{W}) = \exp(F(\boldsymbol{W})) \nabla_{\boldsymbol{W}} F(\boldsymbol{W})$

- We denote by

$$\boldsymbol{e}_y = [0, \ldots, 0, \underbrace{1}_{y}, 0, \ldots, 0]^\top$$

  the one-hot vector representation of class y.

Computing the Gradient

$$
\begin{aligned}
\nabla_{\boldsymbol{W}} L(\boldsymbol{W};(x,y)) &= \nabla_{\boldsymbol{W}} \left(\log \sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) - \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x) \right) \\
&= \nabla_{\boldsymbol{W}} \log \sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) - \nabla_{\boldsymbol{W}} \boldsymbol{w}_y \cdot \boldsymbol{\phi}(x) \\
&= \frac{1}{\sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x))} \sum_{y'} \nabla_{\boldsymbol{W}} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) - \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top \\
&= \frac{1}{Z_x} \sum_{y'} \exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x)) \nabla_{\boldsymbol{W}} \boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x) - \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top \\
&= \sum_{y'} \frac{\exp(\boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x))}{Z_x} \boldsymbol{e}_{y'} \boldsymbol{\phi}(x)^\top - \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top \\
&= \sum_{y'} P_{\boldsymbol{W}}(y'|x) \boldsymbol{e}_{y'} \boldsymbol{\phi}(x)^\top - \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top \\
&= \left(\begin{bmatrix} \vdots \\ P_{\boldsymbol{W}}(y'|x) \\ \vdots \end{bmatrix} - \boldsymbol{e}_y \right) \boldsymbol{\phi}(x)^\top .
\end{aligned}
$$

Logistic Regression Summary

- Define conditional probability

$$P_{\boldsymbol{W}}(y|x) = \frac{\exp(\boldsymbol{w}_y \cdot \boldsymbol{\phi}(x))}{Z_x}$$

- Set weights to maximize conditional log-likelihood of training data:

$$\boldsymbol{W} = \arg\max_{\boldsymbol{W}} \sum_t \log P_{\boldsymbol{W}}(y_t|x_t) = \arg\min_{\boldsymbol{W}} \sum_t L(\boldsymbol{W}; (x_t, y_t))$$

- Can find the gradient and run gradient descent (or any gradient-based optimization algorithm)

$$\nabla_{\boldsymbol{W}} L(\boldsymbol{W}; (x, y)) = \sum_{y'} P_{\boldsymbol{W}}(y'|x) \boldsymbol{e}_{y'} \boldsymbol{\phi}(x)^\top - \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top$$

The Story So Far

- Naive Bayes is generative: maximizes joint likelihood
  - closed form solution (boils down to counting and normalizing)
- Logistic regression is discriminative: maximizes conditional likelihood
  - also called log-linear model and max-entropy classifier
  - no closed form solution
  - stochastic gradient updates look like

$$\boldsymbol{W}^{k+1} = \boldsymbol{W}^{k} + \eta \left(\boldsymbol{e}_y \boldsymbol{\phi}(x)^\top - \sum_{y'} P_{\boldsymbol{w}}(y'|x) \boldsymbol{e}_{y'} \boldsymbol{\phi}(x)^\top \right)$$

- Perceptron is a discriminative, non-probabilistic classifier
  - perceptron's updates look like

$$\boldsymbol{W}^{k+1} = \boldsymbol{W}^{k} + \boldsymbol{e}_y \boldsymbol{\phi}(x)^\top - \boldsymbol{e}_{\widehat{y}} \boldsymbol{\phi}(x)^\top$$

SGD updates for logistic regression and perceptron's updates look similar!

Other Options: Maximizing Margin

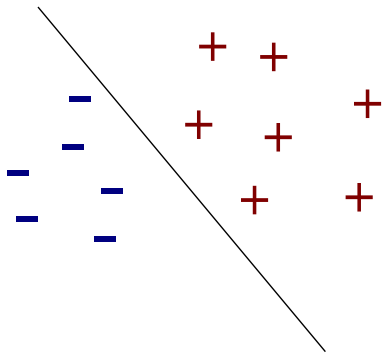
- For a training set $\mathcal{D}$
- Margin of a weight matrix $\boldsymbol{W}$ is smallest γ such that

$$\boldsymbol{w}_{y_t} \cdot \boldsymbol{\phi}(x_t) - \boldsymbol{w}_{y'} \cdot \boldsymbol{\phi}(x_t) \geq \gamma$$

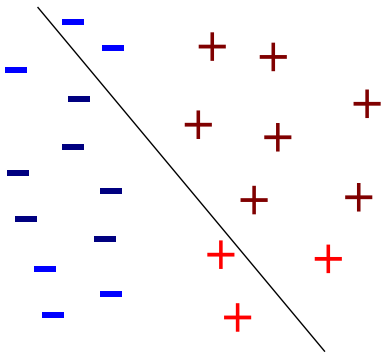
- for every training instance $(x_t, y_t) \in \mathcal{D}$, $y' \in \mathcal{Y}$

Margin

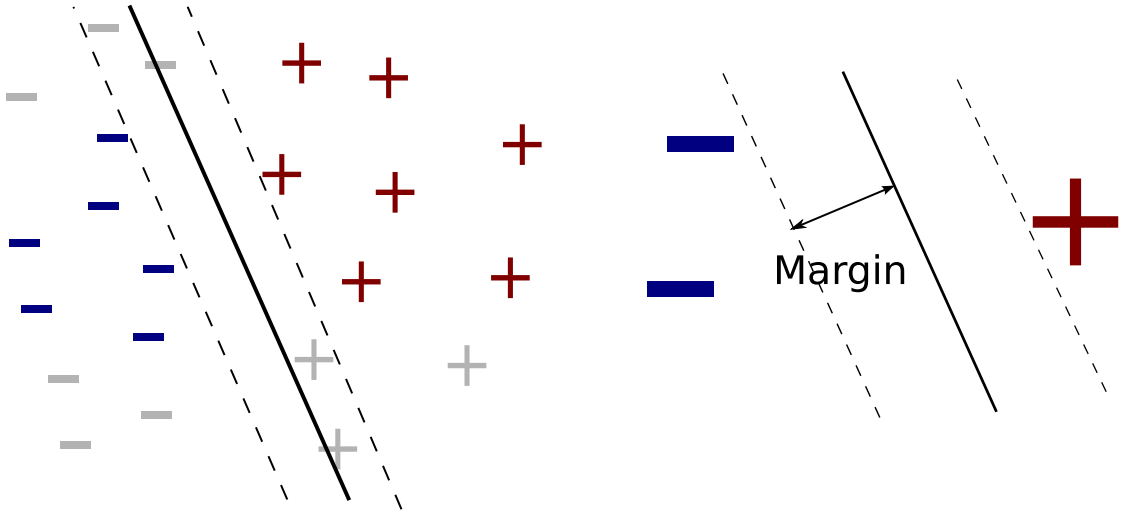
Training



Testing



Denote the value of the margin by γ



Maximizing Margin

- Intuitively maximizing margin makes sense
- More importantly, generalization error to unseen test data is proportional to the inverse of the margin

$$\epsilon \propto \frac{R^2}{\gamma^2 \times N}$$

- **Perceptron**:
  - If a training set is separable by some margin, the perceptron will find a $\boldsymbol{W}$ that separates the data
  - However, the perceptron does not pick $\boldsymbol{W}$ to maximize the margin!
- Support Vector Machines do this (not covered today)

Summary

What we saw

- Linear Classifiers
  - Naive Bayes
  - Logistic Regression
  - Perceptron
  - Support Vector Machines (not covered)

What is next

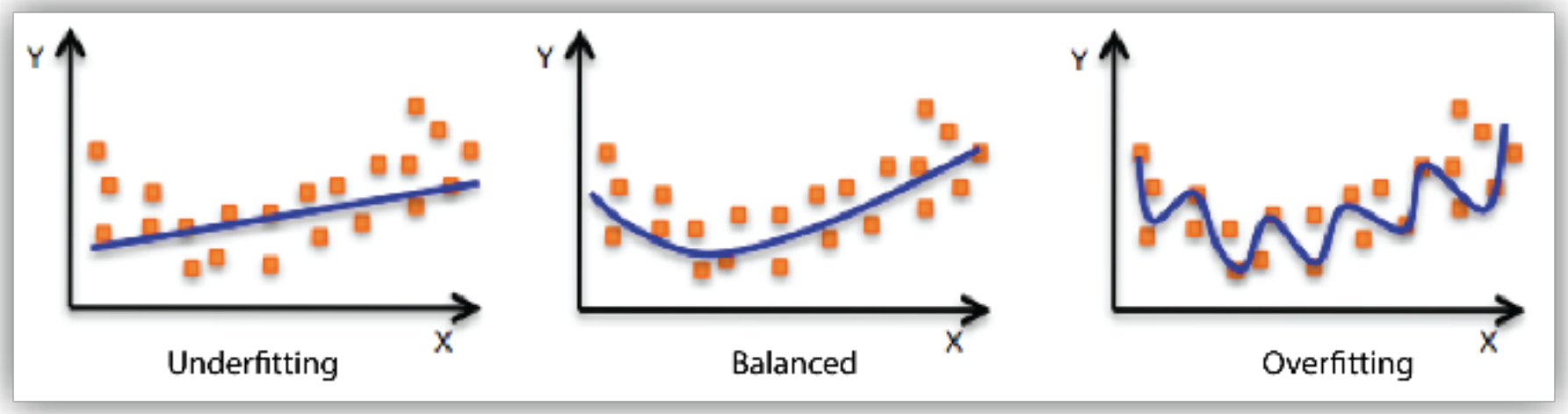
- Regularization
- Softmax
- Non-linear classifiers

Outline

Regularization

Overfitting

If the model is too complex (too many parameters) and the data is scarce, we run the risk of overfitting:



- We saw one example already when talking about add-one smoothing in Naive Bayes!

Regularization

In practice, we regularize models to prevent overfitting

$$\arg\min_{\boldsymbol{W}} \sum_{t=1}^{N} L(\boldsymbol{W}; (x_t, y_t)) + \lambda \Omega(\boldsymbol{W}),$$

where $\Omega(\boldsymbol{W})$ is the regularization function, and λ controls how much to regularize.

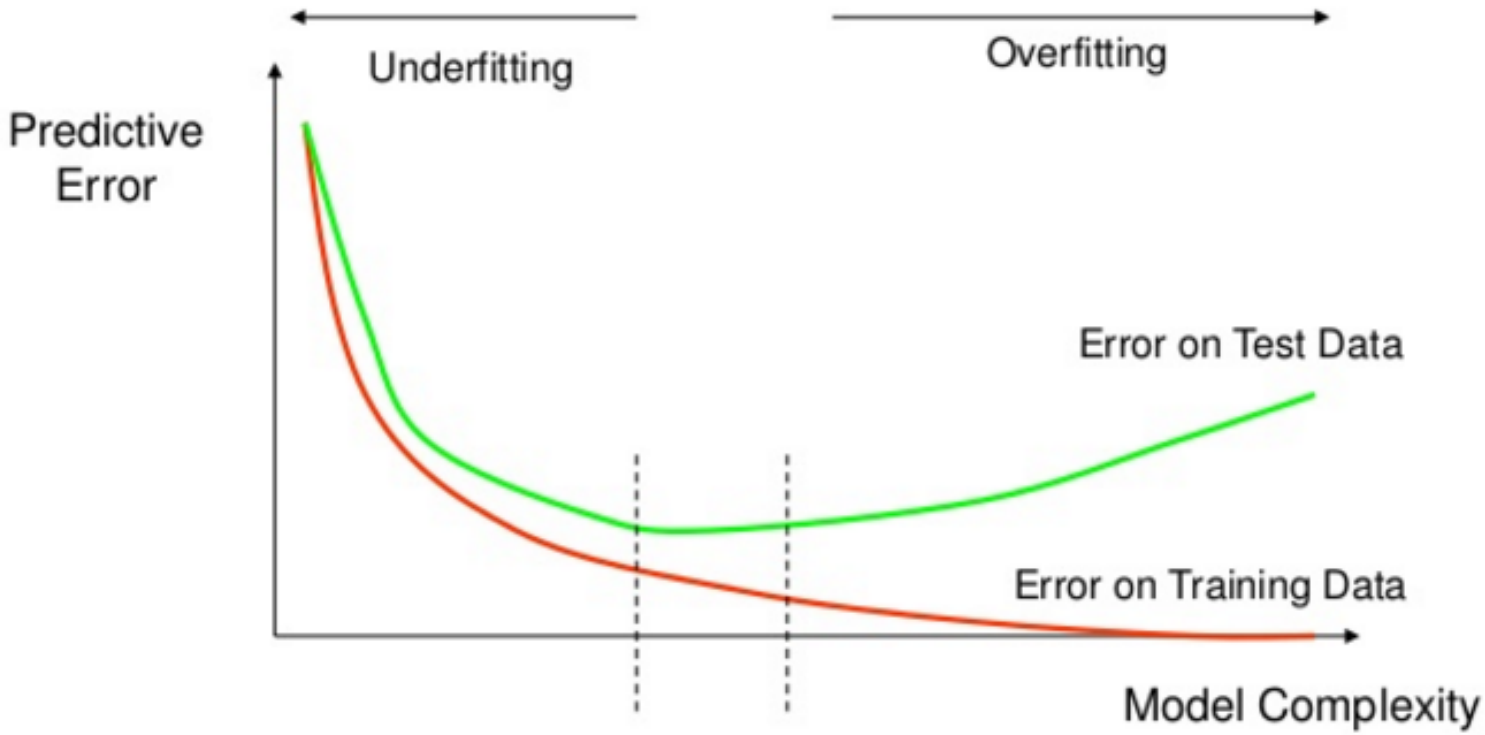
- Gaussian prior (ℓ_2), promotes smaller weights:

$$\Omega(\boldsymbol{W}) = \|\boldsymbol{W}\|_2^2 = \sum_y \|\boldsymbol{w}_y\|_2^2 = \sum_y \sum_j w_{y,j}^2.$$

- Laplacian prior (ℓ_1), promotes sparse weights!

$$\Omega(\boldsymbol{W}) = \|\boldsymbol{W}\|_1 = \sum_y \|\boldsymbol{w}_y\|_1 = \sum_y \sum_j |w_{y,j}|$$

Empirical Risk Minimization



Logistic Regression with ℓ_2 Regularization

$$\sum_{t=1}^{N} L(\boldsymbol{W}; (x_t, y_t)) + \lambda\Omega(\boldsymbol{W}) = -\sum_{t=1}^{N} \log\left(\exp(\boldsymbol{w}_{y_t} \cdot \boldsymbol{\phi}(x_t))/Z_x\right) + \frac{\lambda}{2}\|\boldsymbol{W}\|^2$$

- What is the new gradient?

$$\sum_{t=1}^{N} \nabla_{\boldsymbol{W}} L(\boldsymbol{W}; (x_t, y_t)) + \nabla_{\boldsymbol{W}} \lambda\Omega(\boldsymbol{W})$$

- We know $\nabla_{\boldsymbol{W}} L(\boldsymbol{W}; (x_t, y_t))$
- Just need $\nabla_{\boldsymbol{W}} \frac{\lambda}{2}\|\boldsymbol{W}\|^2 = \lambda\boldsymbol{W}$

Loss Function

Should match as much as possible the metric we want to optimize at test time

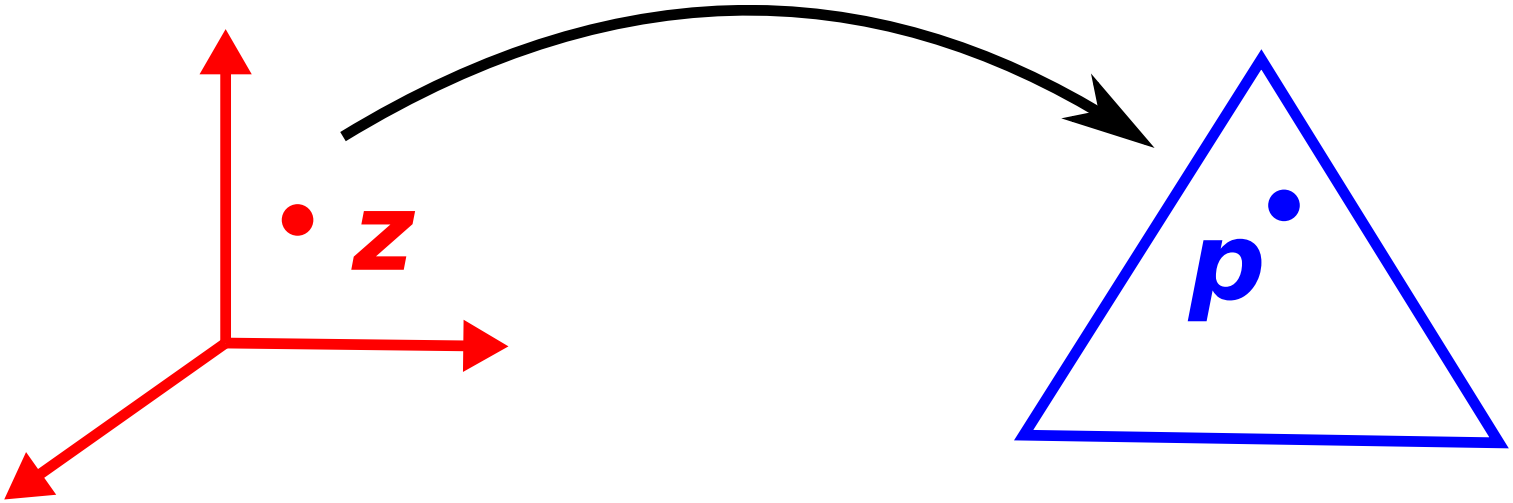
Should be well-behaved (continuous, maybe smooth) to be amenable to optimization (this rules out the 0/1 loss)

Some examples:

- Squared loss for regression
- Negative log-likelihood (cross-entropy): multinomial logistic regression
- Hinge loss: support vector machines
- Sparsemax loss for multi-class and multi-label classification (Martins and Astudillo, 2016)

Recap

How to map from a set of label scores $\mathbb{R}^{|\mathcal{Y}|}$ to a probability distribution over $\mathcal{Y}$?



We already saw one example: softmax.

Another example is sparsemax (not covered): Martins and Astudillo (2016)

Recap: Softmax Transformation

The typical transformation for multi-class classification is **softmax** : $\mathbb{R}^{|\mathcal{Y}|} \rightarrow \Delta^{|\mathcal{Y}|-1}$:

$$\boxed{\textbf{softmax}(\boldsymbol{z}) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_{|\mathcal{Y}|})}{\sum_c \exp(z_c)}\right]}$$

- Underlies multinomial logistic regression!
- Strictly positive, sums to 1
- Resulting distribution has full support: $\textbf{softmax}(\boldsymbol{z}) > \mathbf{0}, \forall \boldsymbol{z}$

Recap: Multinomial Logistic Regression

- The common choice for a softmax output layer
- The classifier estimates $P(y = c \mid x; \boldsymbol{W})$
- We minimize the negative log-likelihood:

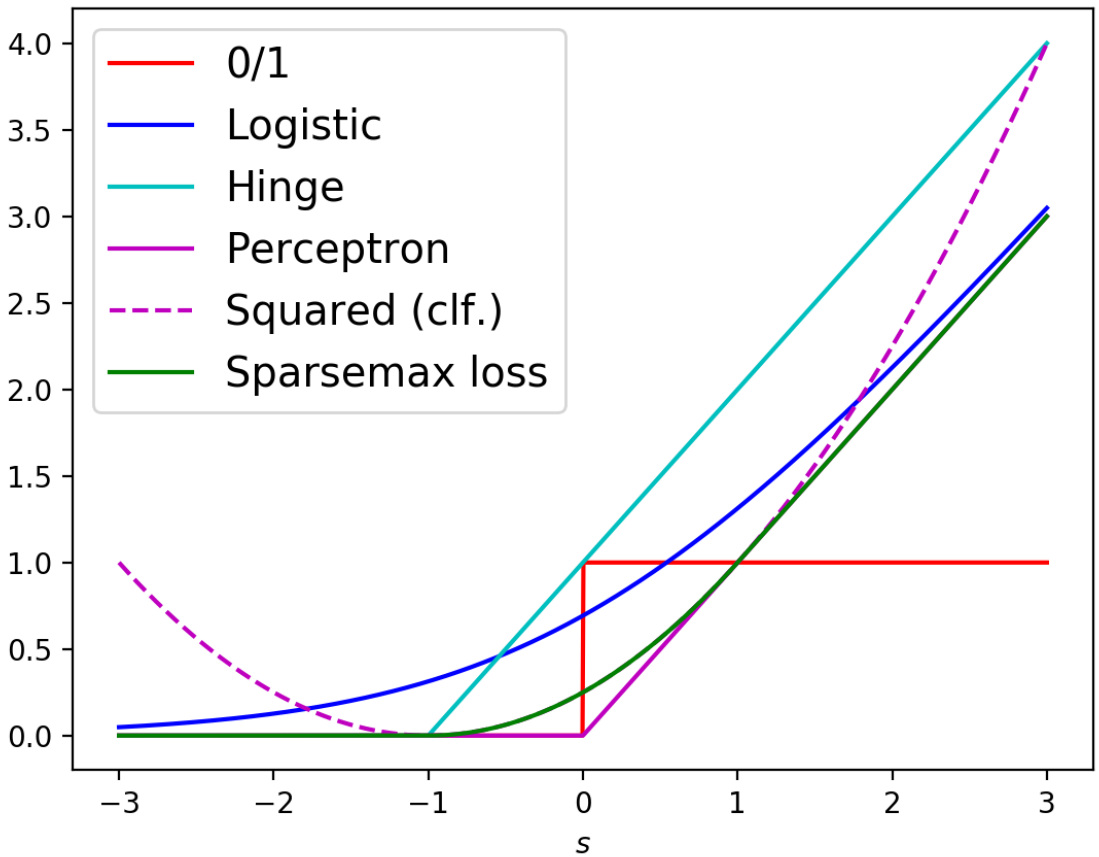
$$\begin{aligned} L(\boldsymbol{W}; (x, y)) &= -\log P(y \mid x; \boldsymbol{W}) \\ &= -\log \left[\mathbf{softmax}(\boldsymbol{z}(x))\right]_y, \end{aligned}$$

  where $z_c(x) = \boldsymbol{w}_c \cdot \boldsymbol{\phi}(x)$ is the score of class c.
- Loss gradient:

$$\nabla_{\boldsymbol{W}} L((x, y); \boldsymbol{W}) = -\left(e_y \boldsymbol{\phi}(x)^\top - \mathbf{softmax}(\boldsymbol{z}(x))\boldsymbol{\phi}(x)^\top\right)$$

Classification Losses (Binary Case)

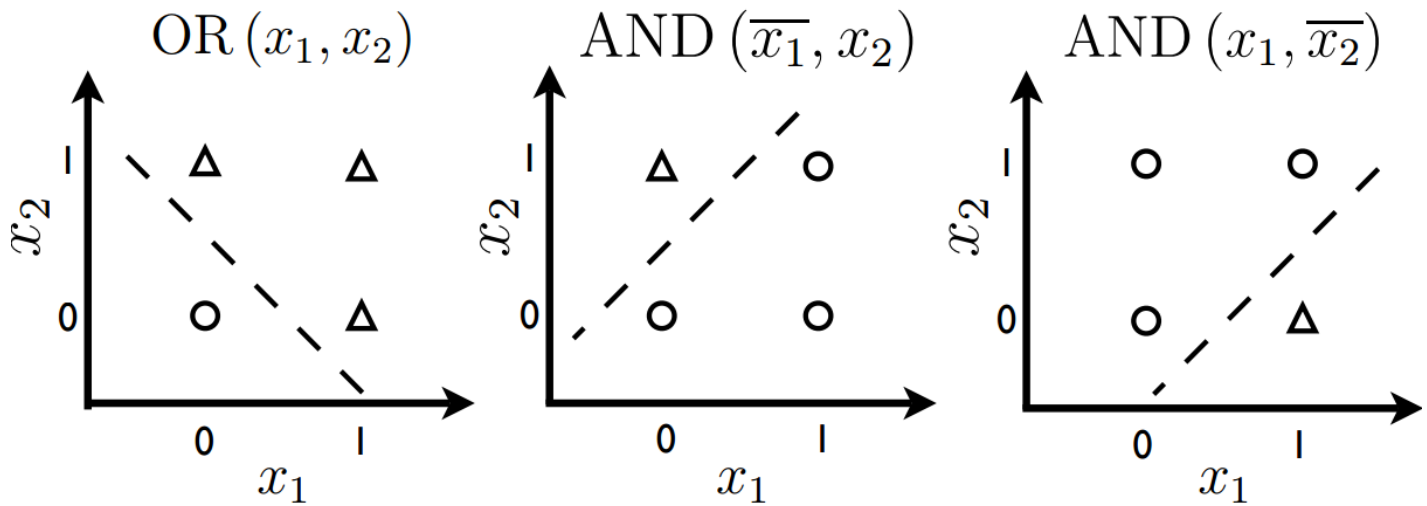
- Let the correct label be $y = +1$ and define $s = z_2 - z_1$.
- Sparsemax loss in 2D becomes a "classification Huber loss":



Outline

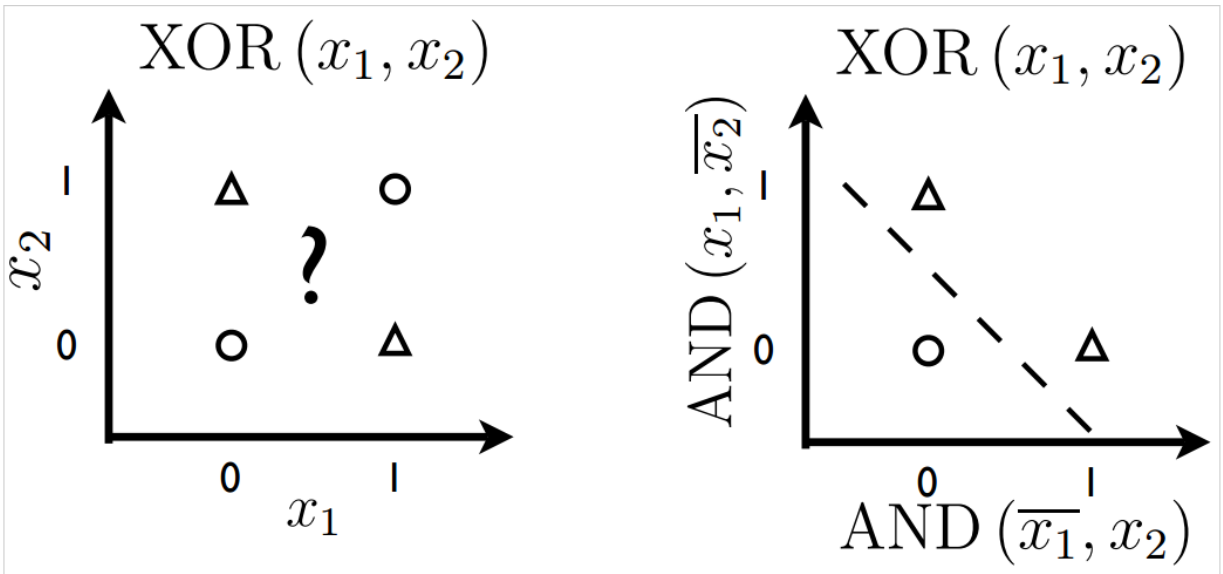
Recap: What a Linear Classifier Can Do

- It **can** solve linearly separable problems (OR, AND)



Recap: What a Linear Classifier Can't Do

- ... but it **can't** solve non-linearly separable problems such as simple XOR (unless input is transformed into a better representation):



- This was observed by Minsky and Papert (1969) (for the perceptron) and motivated strong criticisms

Summary: Linear Classifiers

We've seen

- Perceptron
- Naive Bayes
- Logistic regression
- Support vector machines (not covered)

All lead to convex optimization problems $\Rightarrow$ no issues with local minima/initialization

All assume the features are well-engineered such that the data is nearly linearly separable

What If Data Are Not Linearly Separable?

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

Kernel methods:

- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

Kernel methods:

- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels

Neural networks (next class!)

- embrace non-convexity and local minima
- instead of engineering features/kernels, engineer the model architecture

Two Views of Machine Learning

There's two big ways of building machine learning systems:

1. Feature-based: describe objects' properties (features) and build models that manipulate them
   - everything that we have seen so far.
2. Similarity-based: don't describe objects by their properties; rather, build systems based on **comparing** objects to each other
   - k-th nearest neighbors; kernel methods; Gaussian processes.

Sometimes the two are equivalent!

Nearest Neighbor Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- Instead of "training", **memorize** all the data $\mathcal{D} = \{(x_i, y_i)_{i=1}^N\}$
- Given a new input x, find its **most similar** data point x_i and predict

$$\widehat{y} = y_i$$

- Many variants (e.g. k-th nearest neighbor)
- **Disadvantage:** requires searching over the entire training data
- Specialized data structures can be used to speed up search.

Kernels

- A kernel is a similarity function between two points that is **symmetric** and **positive semi-definite**, which we denote by:

$$\kappa(x_i, x_j) \in \mathbb{R}$$

- Given dataset $\mathcal{D} = \{(x_i, y_i)_{i=1}^N\}$, the Gram matrix $\boldsymbol{K}$ is the $N \times N$ matrix defined as:

$$K_{i,j} = \kappa(x_i, x_j)$$

- **Symmetric:**

$$\kappa(x_i, x_j) = \kappa(x_j, x_i)$$

- **Positive definite:** for all non-zero $\mathbf{v}$

$$\mathbf{v}\boldsymbol{K}\mathbf{v}^T \geq 0$$

Kernels

- **Mercer's Theorem**: for any kernel $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, there exists some feature mapping $\phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}$, s.t.:

$$\kappa(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

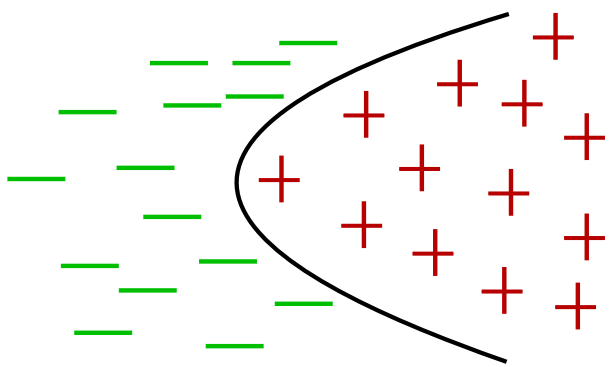
- That is: a kernel corresponds to some a mapping in some **implicit** feature space!
- Kernel trick: take a feature-based algorithm (SVMs, perceptron, logistic regression) and replace all explicit feature computations by **kernel evaluations**!

$$\boldsymbol{w}_y \cdot \phi(x) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \alpha_{i,y} \kappa(x, x_i) \quad \text{for some } \alpha_{i,y} \in \mathbb{R}$$

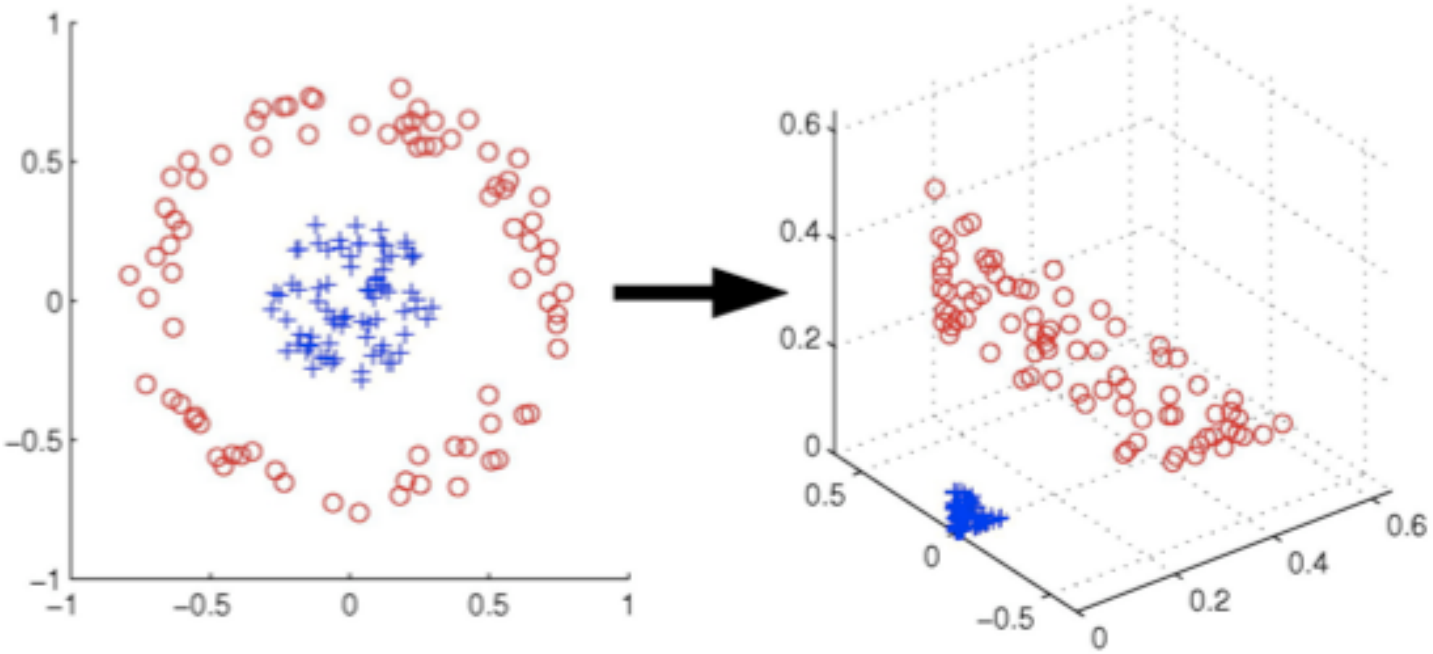
- Extremely popular idea in the 1990-2000s!

Kernels = Tractable Non-Linearity

- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space
- Computing a non-linear kernel is sometimes better computationally than calculating the corresponding dot product in the high dimension feature space
- Many models can be "kernelized" – learning algorithms generally solve the dual optimization problem (also convex)
- Drawback: quadratic dependency on dataset size



Linear Classifiers in High Dimension



$$\begin{array}{ccc} \Re^2 & \longrightarrow & \Re^3 \\ (x_1, x_2) & \longmapsto & (z_1, z_2, z_3) = (x_1^2, \sqrt{2}x_1x_2, x_2^2) \end{array}$$

Popular Kernels

- Polynomial kernel

$$\kappa(x_i, x_j) = (\phi(x_i) \cdot \phi(x_j) + 1)^d$$

- Gaussian radial basis kernel

$$\kappa(x_i, x_j) = exp(\frac{-||\phi(x_i) - \phi(x_j)||^2}{2\sigma})$$

- String kernels (Lodhi et al., 2002; Collins and Duffy, 2002)
- Tree kernels (Collins and Duffy, 2002)

Conclusions

- Linear classifiers are a broad class including well-known ML methods such as perceptron, Naive Bayes, logistic regression, support vector machines
- They all involve manipulating weights and features
- They either lead to closed-form solutions or convex optimization problems (no local minima)
- Stochastic gradient descent algorithms are useful if training datasets are large
- However, they require manual specification of feature representations

References I

Collins, M. and Duffy, N. (2002). Convolution kernels for natural language. *Advances in Neural Information Processing Systems*, 1:625–632.

Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444.

Martins, A. F. T. and Astudillo, R. (2016). From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification. In *Proc. of the International Conference on Machine Learning*.

Minsky, M. and Papert, S. (1969). Perceptrons.