

# Lecture 3: Linear Models II

André Martins, Francisco Melo, Mário Figueiredo



Deep Learning Course, Winter 2022-2023

# Today's Roadmap

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

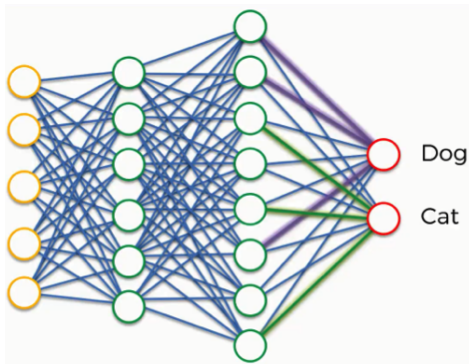
# Why Linear Classifiers?

We know the course title promised “**deep**”, ...

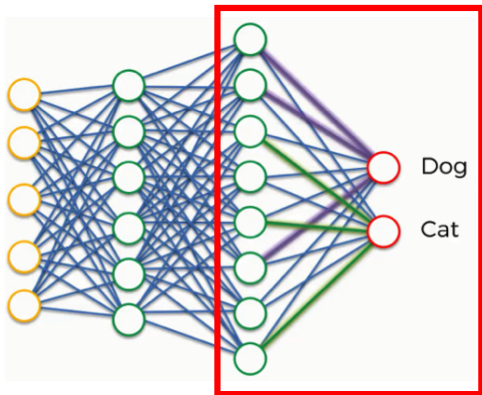
...but (as explained in Lecture 2):

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

# Linear Classifiers and Neural Networks

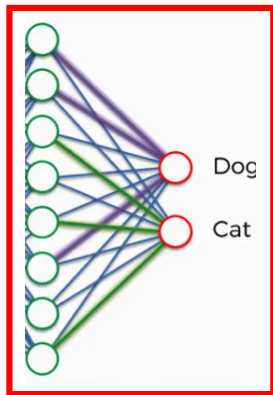


# Linear Classifiers and Neural Networks



**Linear Classifier**

# Linear Classifiers and Neural Networks

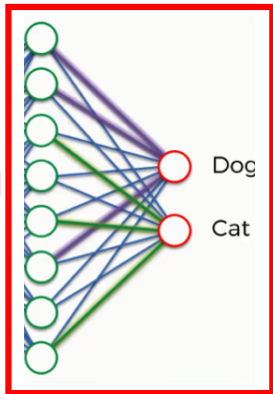


**Linear Classifier**

# Linear Classifiers and Neural Networks



**Handcrafted  
Features**

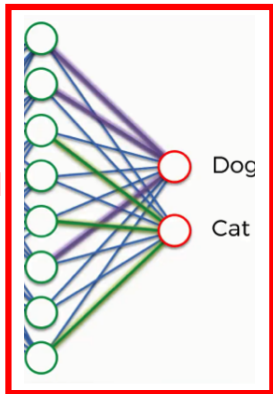


**Linear Classifier**

# Linear Classifiers and Neural Networks



**Handcrafted  
Features**

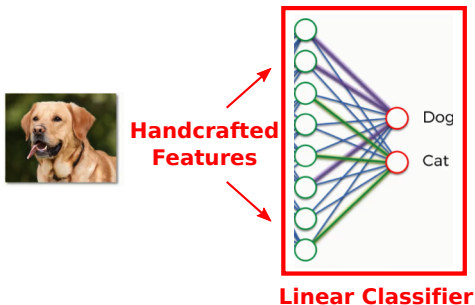


**Linear Classifier**

...in fact, these may also be **learned features**, from all the previous layers of some **deep neural network**.



## Reminder: Linear Classifier

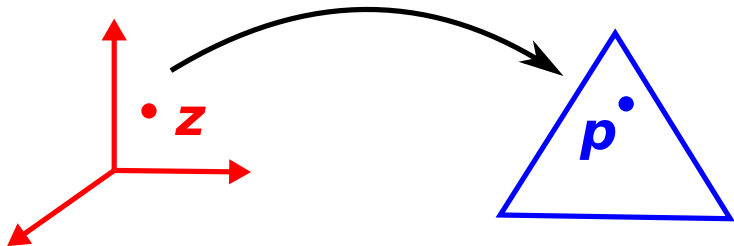


$$\hat{y} = \arg \max_y ((\mathbf{W}\phi(x))_y), \quad \mathbf{W} = \begin{bmatrix} w_1^T \\ \vdots \\ w_{|y|}^T \end{bmatrix}$$

As before, assume  $\phi_0(x) = 1$ , absorbing the bias terms.

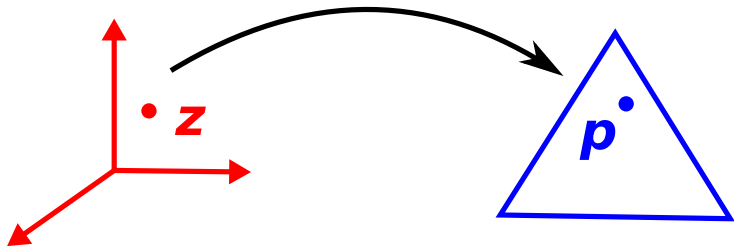
# Key Problem

Mapping from a vector of scores  $\mathbb{R}^{|\mathcal{Y}|}$  to a probability distribution over  $\mathcal{Y}$ ?



# Key Problem

Mapping from a vector of scores  $\mathbb{R}^{|\mathcal{Y}|}$  to a probability distribution over  $\mathcal{Y}$ ?



We will see an important mapping: **softmax** (next).

# Outline

- ① Logistic Regression
- ② Regularization
- ③ Non-Linear Classifiers

# Logistic Regression

A linear model gives a **score** for each class  $y$ :  $w_y^T \phi(x)$ ,

# Logistic Regression

A linear model gives a **score** for each class  $y$ :  $w_y^T \phi(x)$ ,

...from which we may compute a **conditional (posterior) probability**:

$$P(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}, \quad \text{where } Z_x = \sum_{y' \in \mathcal{Y}} \exp(w_{y'}^T \phi(x))$$

# Logistic Regression

A linear model gives a **score** for each class  $y$ :  $w_y^T \phi(x)$ ,

...from which we may compute a **conditional (posterior) probability**:

$$P(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}, \quad \text{where } Z_x = \sum_{y' \in \mathcal{Y}} \exp(w_{y'}^T \phi(x))$$

This is called the **softmax transformation**

# Logistic Regression

A linear model gives a **score** for each class  $y$ :  $w_y^T \phi(x)$ ,

...from which we may compute a **conditional (posterior) probability**:

$$P(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}, \quad \text{where } Z_x = \sum_{y' \in \mathcal{Y}} \exp(w_{y'}^T \phi(x))$$

This is called the **softmax transformation**

Still a linear classifier

$$\begin{aligned} \arg \max_y P(y|x) &= \arg \max_y \frac{\exp(w_y^T \phi(x))}{Z_x} \\ &= \arg \max_y \exp(w_y^T \phi(x)) \\ &= \arg \max_y w_y^T \phi(x) \end{aligned}$$



# Logistic Regression

A linear model gives a **score** for each class  $y$ :  $w_y^T \phi(x)$ ,

...from which we may compute a **conditional (posterior) probability**:

$$P(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}, \quad \text{where } Z_x = \sum_{y' \in \mathcal{Y}} \exp(w_{y'}^T \phi(x))$$

This is called the **softmax transformation**

Still a linear classifier

$$\begin{aligned} \arg \max_y P(y|x) &= \arg \max_y \frac{\exp(w_y^T \phi(x))}{Z_x} \\ &= \arg \max_y \exp(w_y^T \phi(x)) \\ &= \arg \max_y w_y^T \phi(x) \end{aligned}$$

The decision boundaries are still hyperplanes w.r.t.  $\phi(x)$ .

# Binary Logistic Regression

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

# Binary Logistic Regression

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

Scores: 0 for negative class,  $w^T \phi(x)$  for positive class

# Binary Logistic Regression

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

Scores: 0 for negative class,  $w^T \phi(x)$  for positive class

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

# Binary Logistic Regression

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

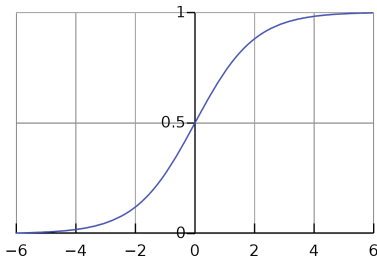
Scores: 0 for negative class,  $w^T \phi(x)$  for positive class

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

This is called a **sigmoid transformation**.

# Sigmoid Transformation

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



- Widely used in neural networks.
- Maps  $\mathbb{R}$  into  $[0, 1]$ .
- The output can be interpreted as a probability.
- Positive, bounded, strictly increasing.

# Multinomial Logistic Regression

$$P_{\mathbf{w}}(y | x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

# Multinomial Logistic Regression

$$P_{\mathbf{w}}(y | x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

- How to learn weights  $\mathbf{W}$ ?



# Multinomial Logistic Regression

$$P_{\mathbf{w}}(y | x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

- How to learn weights  $\mathbf{W}$ ?
- Maximize the **conditional log-likelihood** of training data:

$$\begin{aligned}\widehat{\mathbf{W}} &= \arg \max_{\mathbf{w}} \log \left( \prod_{t=1}^N P_{\mathbf{w}}(y_t | x_t) \right) \\ &= \arg \min_{\mathbf{w}} - \sum_{t=1}^N \log P_{\mathbf{w}}(y_t | x_t) \\ &= \arg \min_{\mathbf{w}} \sum_{t=1}^N \left( \underbrace{\log \sum_{y'} \exp(w_{y'}^T \phi(x_t))}_{Z_{x_t}} - w_{y_t}^T \phi(x_t) \right),\end{aligned}$$

# Multinomial Logistic Regression

$$P_{\mathbf{w}}(y | x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

- How to learn weights  $\mathbf{W}$ ?
- Maximize the **conditional log-likelihood** of training data:

$$\begin{aligned}\widehat{\mathbf{W}} &= \arg \max_{\mathbf{W}} \log \left( \prod_{t=1}^N P_{\mathbf{w}}(y_t | x_t) \right) \\ &= \arg \min_{\mathbf{W}} - \sum_{t=1}^N \log P_{\mathbf{w}}(y_t | x_t) \\ &= \arg \min_{\mathbf{W}} \sum_{t=1}^N \left( \underbrace{\log \sum_{y'} \exp(w_{y'}^T \phi(x_t))}_{Z_{x_t}} - w_{y_t}^T \phi(x_t) \right),\end{aligned}$$

- i.e., choose  $\mathbf{W}$  to maximize the probability of the true labels.

# Logistic Regression

- This objective function is **strictly convex**.

# Logistic Regression

- This objective function is **strictly convex**.
- Proof left as exercise! (hint, compute the Hessian).

# Logistic Regression

- This objective function is **strictly convex**.
- Proof left as exercise! (hint, compute the Hessian).
- Therefore, any local minimum is a global minimum.

# Logistic Regression

- This objective function is **strictly convex**.
- Proof left as exercise! (hint, compute the Hessian).
- Therefore, any local minimum is a global minimum.
- No closed-form solution; many numerical methods have been proposed
  - ✓ Gradient methods (gradient descent, conjugate gradient).
  - ✓ Quasi-Newton methods (L-BFGS, ...).

## Recap: Gradient Descent

- Goal: minimize  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ , for differentiable  $f$

## Recap: Gradient Descent

- Goal: minimize  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ , for differentiable  $f$
- Take **small steps** in the **negative gradient direction** until a **stopping criterion** is met:

$$x^{(t+1)} \leftarrow x^{(t)} - \eta_t \nabla f(x^{(t)})$$



## Recap: Gradient Descent

- Goal: minimize  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ , for differentiable  $f$
- Take **small steps** in the **negative gradient direction** until a **stopping criterion** is met:

$$x^{(t+1)} \leftarrow x^{(t)} - \eta_t \nabla f(x^{(t)})$$

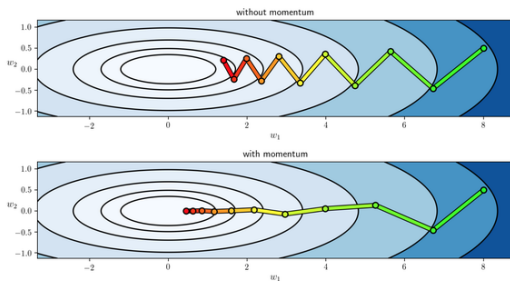
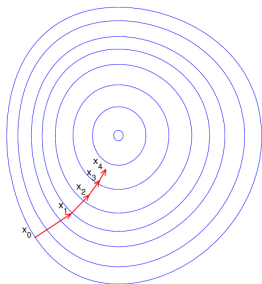
- Choosing the **step-size**: crucial for convergence and performance.

# Recap: Gradient Descent

- Goal: minimize  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ , for differentiable  $f$
- Take **small steps** in the **negative gradient direction** until a **stopping criterion** is met:

$$x^{(t+1)} \leftarrow x^{(t)} - \eta_t \nabla f(x^{(t)})$$

- Choosing the **step-size**: crucial for convergence and performance.
- GD may work well, or not so well. There are many ways to improve it.



# Gradient Descent

- **Loss function** in logistic regression is

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x).$$

# Gradient Descent

- **Loss function** in logistic regression is

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x).$$

- We want to find:  $\arg \min_{\mathbf{W}} \sum_{t=1}^N L(\mathbf{W}; (x_t, y_t))$

✓ Set  $\mathbf{W}^{(0)} = \mathbf{0}$

✓ Iterate until convergence (for suitable stepsize  $\eta_k$ ):

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

# Gradient Descent

- **Loss function** in logistic regression is

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x).$$

- We want to find:  $\arg \min_{\mathbf{W}} \sum_{t=1}^N L(\mathbf{W}; (x_t, y_t))$

✓ Set  $\mathbf{W}^{(0)} = \mathbf{0}$

✓ Iterate until convergence (for suitable stepsize  $\eta_k$ ):

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

- $L(\mathbf{W})$  convex  $\Rightarrow$  gradient descent will reach the global optimum  $\mathbf{W}$ , with appropriate choice of step-size  $\eta_k$ .

# Stochastic Gradient Descent

Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set  $\mathbf{W}^{(0)} = \mathbf{0}$
- Iterate until convergence:
  - Pick  $(x_t, y_t)$  randomly.
  - Update  $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L(\mathbf{W}^{(k)}; (x_t, y_t))$

# Stochastic Gradient Descent

Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set  $\mathbf{W}^{(0)} = \mathbf{0}$
- Iterate until convergence:
  - Pick  $(x_t, y_t)$  randomly.
  - Update  $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L(\mathbf{W}^{(k)}; (x_t, y_t))$
- *i.e.* approximate the gradient with a noisy, unbiased, version based on a single sample

# Stochastic Gradient Descent

Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set  $\mathbf{W}^{(0)} = \mathbf{0}$
- Iterate until convergence:
  - Pick  $(x_t, y_t)$  randomly.
  - Update  $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L(\mathbf{W}^{(k)}; (x_t, y_t))$
- *i.e.* approximate the gradient with a noisy, unbiased, version based on a single sample
- Variant between GD and SGD: mini-batch (pick  $B$ , rather than 1 pair)

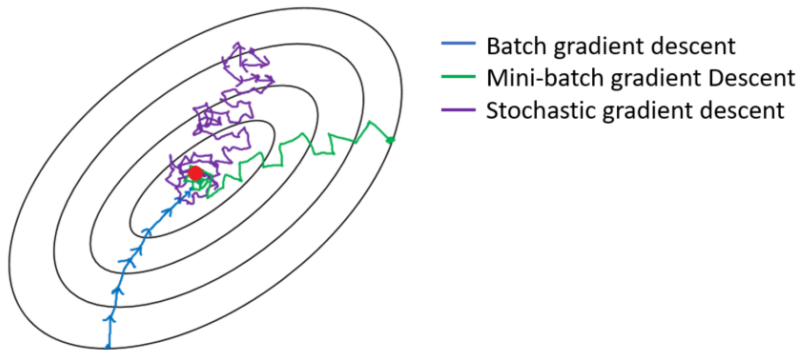


# Stochastic Gradient Descent

Monte Carlo approximation of the gradient (more frequent updates, convenient with large datasets):

- Set  $\mathbf{W}^{(0)} = \mathbf{0}$
- Iterate until convergence:
  - Pick  $(x_t, y_t)$  randomly.
  - Update  $\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \eta_k \nabla_{\mathbf{W}} L(\mathbf{W}^{(k)}; (x_t, y_t))$
- *i.e.* approximate the gradient with a noisy, unbiased, version based on a single sample
- Variant between GD and SGD: mini-batch (pick  $B$ , rather than 1 pair)
- All guaranteed to find the optimal  $\mathbf{W}$  (for suitable step sizes)

# Stochastic vs Batch Gradient Descent



# Computing the Gradient

- We need to compute  $\nabla_{\mathbf{w}} L(\mathbf{W}; (x_t, y_t))$ , where

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x)$$

# Computing the Gradient

- We need to compute  $\nabla_{\mathbf{W}} L(\mathbf{W}; (x_t, y_t))$ , where

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x)$$

- Some reminders (we next omit the subscript  $\mathbf{W}$  for simplicity)

①  $\nabla \log F(\mathbf{W}) = \frac{\nabla F(\mathbf{W})}{F(\mathbf{W})}$

②  $\nabla \exp F(\mathbf{W}) = \exp(F(\mathbf{W})) \nabla F(\mathbf{W})$

# Computing the Gradient

- We need to compute  $\nabla_{\mathbf{W}} L(\mathbf{W}; (x_t, y_t))$ , where

$$L(\mathbf{W}; (x, y)) = \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x)$$

- Some reminders (we next omit the subscript  $\mathbf{W}$  for simplicity)

①  $\nabla \log F(\mathbf{W}) = \frac{\nabla F(\mathbf{W})}{F(\mathbf{W})}$

②  $\nabla \exp F(\mathbf{W}) = \exp(F(\mathbf{W})) \nabla F(\mathbf{W})$

- We denote by

$$e_y = [0, \dots, 0, 1, 0, \dots, 0]^T, \quad \text{1 in } y\text{-th position}$$

the **one-hot** vector representation of class  $y$ .

# Computing the Gradient

$$\begin{aligned}\nabla L(\mathbf{W}; (x, y)) &= \nabla \left( \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - w_y^T \phi(x) \right) \\ &= \nabla \log \sum_{y'} \exp(w_{y'}^T \phi(x)) - \nabla w_y^T \phi(x) \\ &= \frac{1}{\sum_{y'} \exp(w_{y'}^T \phi(x))} \sum_{y'} \nabla \exp(w_{y'}^T \phi(x)) - e_y \phi(x)^T \\ &= \frac{1}{Z_x} \sum_{y'} \exp(w_{y'}^T \phi(x)) \nabla w_{y'}^T \phi(x) - e_y \phi(x)^T \\ &= \sum_{y'} \frac{\exp(w_{y'}^T \phi(x))}{Z_x} e_{y'} \phi(x)^T - e_y \phi(x)^T \\ &= \sum_{y'} P_{\mathbf{W}}(y'|x) e_{y'} \phi(x)^T - e_y \phi(x)^T \\ &= \left( \begin{bmatrix} P_{\mathbf{W}}(1|x) \\ \vdots \\ P_{\mathbf{W}}(|\mathcal{Y}||x) \end{bmatrix} - e_y \right) \phi(x)^T.\end{aligned}$$

# Logistic Regression Summary

- Define conditional probability

$$P_{\mathbf{w}}(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

# Logistic Regression Summary

- Define conditional probability

$$P_{\mathbf{W}}(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

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

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



# Logistic Regression Summary

- Define conditional probability

$$P_{\mathbf{W}}(y|x) = \frac{\exp(w_y^T \phi(x))}{Z_x}$$

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

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

- Run gradient descent, or another gradient-based algorithm, using

$$\nabla L(\mathbf{W}; (x, y)) = \sum_{y'} P_{\mathbf{W}}(y'|x) e_{y'} \phi(x)^T - e_y \phi(x)^T$$

# The Story So Far

# The Story So Far

# The Story So Far

- Logistic regression is **discriminative**: maximizes conditional likelihood
  - ✓ also called log-linear model and max-entropy classifier

# The Story So Far

- Logistic regression is **discriminative**: maximizes conditional likelihood
  - ✓ also called log-linear model and max-entropy classifier
  - ✓ no closed-form solution

# The Story So Far

- 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

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

# The Story So Far

- **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

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

- The **perceptron** is also **discriminative**, but non-probabilistic classifier

# The Story So Far

- **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

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

- The **perceptron** is also **discriminative**, but non-probabilistic classifier
  - ✓ perceptron's updates look like

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{e}_y \phi(x)^\top - \mathbf{e}_{\hat{y}} \phi(x)^\top$$



# The Story So Far

- **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

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

- The **perceptron** is also **discriminative**, but non-probabilistic classifier
  - ✓ perceptron's updates look like

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{e}_y \phi(x)^\top - \mathbf{e}_{\hat{y}} \phi(x)^\top$$

- SGD updates for logistic regression and the perceptron look similar!

## Other Options: Maximizing Margin

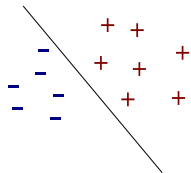
- For a training set  $\mathcal{D} = \{(x_t, y_t), t = 1, \dots, N\}$
- **Margin** achieved by a weight matrix  $\mathbf{W}$ : smallest  $\gamma$  such that, for every training instance  $(x_t, y_t) \in \mathcal{D}$ ,

$$w_{y_t}^T \phi(x_t) \geq w_{y'}^T \phi(x_t) + \gamma, \quad \text{for any } y' \neq y_t$$

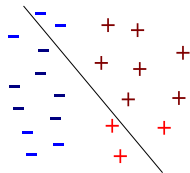
- i.e., score of the correct label is at least  $\gamma$  higher than that of any other label.

# Margin

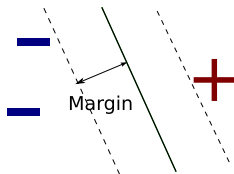
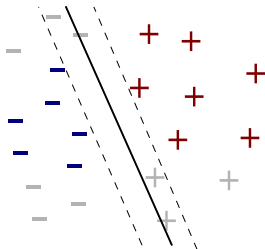
Training



Testing



Denote the value of the margin by  $\gamma$



# Maximizing Margin

- Intuitively maximizing margin makes sense

# Maximizing Margin

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

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

# Maximizing Margin

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

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

- **Perceptron:**
  - ✓ If a training set is separable by some margin, the perceptron will find a  $\mathbf{W}$  that separates the data
  - ✓ However, the perceptron **does not** pick  $\mathbf{W}$  to maximize the margin!

# Maximizing Margin

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

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

- **Perceptron:**
  - ✓ If a training set is separable by some margin, the perceptron will find a  $\mathbf{W}$  that separates the data
  - ✓ However, the perceptron **does not** pick  $\mathbf{W}$  to maximize the margin!
  - ✓ **Support vector machines** (SVM) do exactly this (not covered)

# Summary

## What we saw:

- Linear Classifiers
  - ✓ Logistic regression
  - ✓ Perceptron
  - ✓ Support vector machines (not covered)

## Next:

- Regularization
- Softmax
- Non-linear classifiers

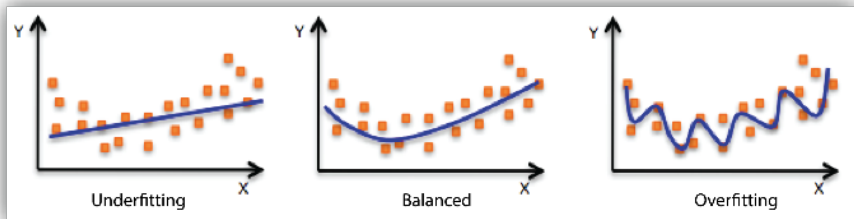


# Outline

- ① Logistic Regression
- ② Regularization
- ③ Non-Linear Classifiers

# Overfitting

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



# Regularization

In practice, we **regularize** to prevent overfitting

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

$\Omega(\mathbf{W})$  is the regularization function, and  $\lambda$  controls its weight.

# Regularization

In practice, we **regularize** to prevent overfitting

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

$\Omega(\mathbf{W})$  is the regularization function, and  $\lambda$  controls its weight.

- $\ell_2$  regularization promotes **smaller** weights:

$$\Omega(\mathbf{W}) = \sum_y \|w_y\|_2^2 = \sum_y \sum_i w_{y,i}^2.$$

# Regularization

In practice, we **regularize** to prevent overfitting

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

$\Omega(\mathbf{W})$  is the regularization function, and  $\lambda$  controls its weight.

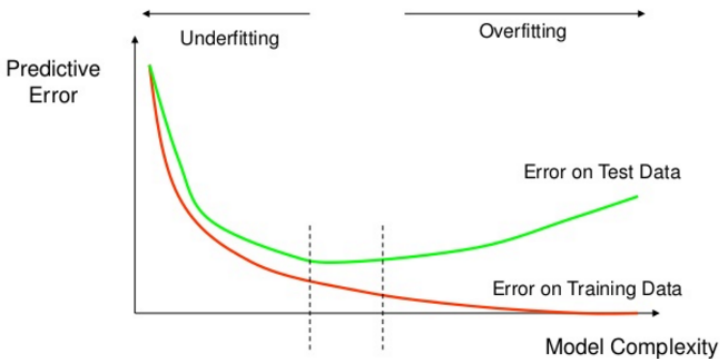
- $\ell_2$  regularization promotes **smaller** weights:

$$\Omega(\mathbf{W}) = \sum_y \|w_y\|_2^2 = \sum_y \sum_i w_{y,i}^2.$$

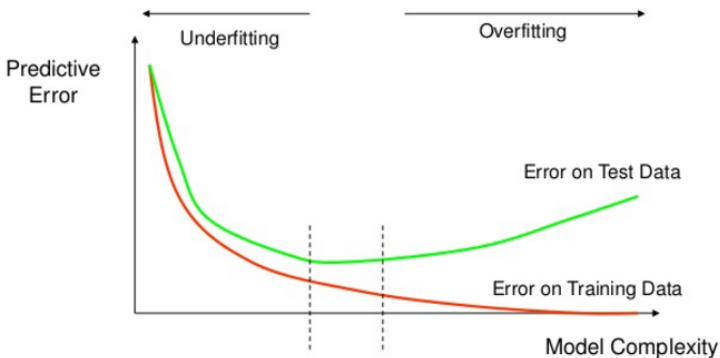
- $\ell_1$  regularization promotes **smaller** and **sparse** weights!

$$\Omega(\mathbf{W}) = \sum_y \|w_y\|_1 = \sum_y \sum_i |w_{y,i}|$$

# Empirical Risk Minimization: The Classical View

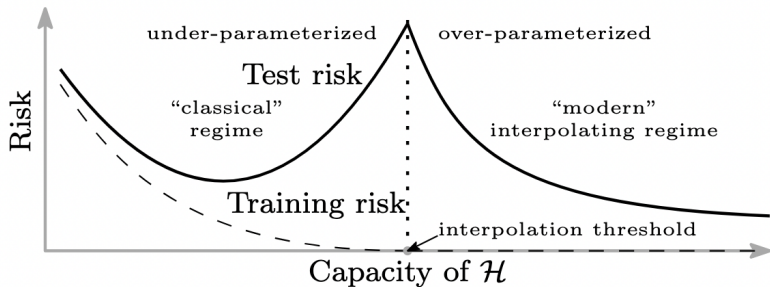


# Empirical Risk Minimization: The Classical View



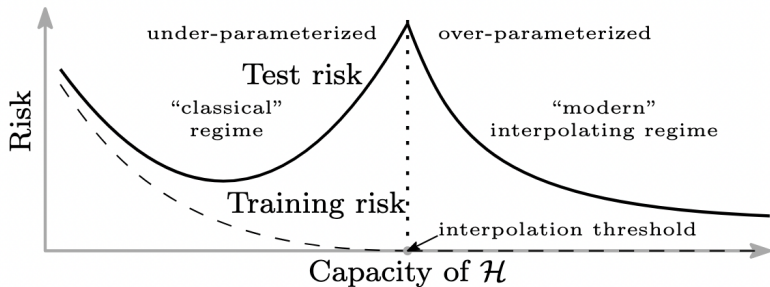
...doesn't explain the excellent performance of deep networks, which are very complex (have many parameters).

# Empirical Risk Minimization: Double Descent





# Empirical Risk Minimization: Double Descent



Fully understanding [double descent](#) is an open research topic.

# Logistic Regression with $\ell_2$ Regularization

- $\ell_2$ -regularized (a.k.a. ridge or weight decay) logistic regression

$$\min - \sum_{t=1}^N \underbrace{\log \left( \exp(w_{y_t}^T \phi(x_t)) / Z_x \right)}_{L(\mathbf{W}; (x_t, y_t))} + \frac{\lambda}{2} \|\mathbf{W}\|_F^2$$

## Logistic Regression with $\ell_2$ Regularization

- $\ell_2$ -regularized (a.k.a. ridge or weight decay) logistic regression

$$\min - \sum_{t=1}^N \underbrace{\log \left( \exp(w_{y_t}^T \phi(x_t)) / Z_x \right)}_{L(\mathbf{w}; (x_t, y_t))} + \frac{\lambda}{2} \|\mathbf{w}\|_F^2$$

- The gradient is now

$$\sum_{t=1}^N \nabla L(\mathbf{w}; (x_t, y_t)) + \frac{\lambda}{2} \nabla \|\mathbf{w}\|_2^2$$

# Logistic Regression with $\ell_2$ Regularization

- $\ell_2$ -regularized (a.k.a. ridge or weight decay) logistic regression

$$\min - \sum_{t=1}^N \underbrace{\log \left( \exp(w_{y_t}^T \phi(x_t)) / Z_x \right)}_{L(\mathbf{W}; (x_t, y_t))} + \frac{\lambda}{2} \|\mathbf{W}\|_F^2$$

- The gradient is now

$$\sum_{t=1}^N \nabla L(\mathbf{W}; (x_t, y_t)) + \frac{\lambda}{2} \nabla \|\mathbf{W}\|_2^2$$

- For gradient descent, we also need  $\nabla \|\mathbf{W}\|_2^2 = 2\mathbf{W}$

# Logistic Regression with $\ell_2$ Regularization

- $\ell_2$ -regularized (a.k.a. ridge or weight decay) logistic regression

$$\min - \sum_{t=1}^N \underbrace{\log \left( \exp(w_{y_t}^T \phi(x_t)) / Z_x \right)}_{L(\mathbf{W}; (x_t, y_t))} + \frac{\lambda}{2} \|\mathbf{W}\|_F^2$$

- The gradient is now

$$\sum_{t=1}^N \nabla L(\mathbf{W}; (x_t, y_t)) + \frac{\lambda}{2} \nabla \|\mathbf{W}\|_2^2$$

- For gradient descent, we also need  $\nabla \|\mathbf{W}\|_2^2 = 2\mathbf{W}$
- GD step (similarly for SGD) with **weight decay**:

$$\mathbf{W}^{(k+1)} = \underbrace{(1 - \eta_k \lambda)}_{\text{weight decay } (1 - \eta_k \lambda) < 1} \mathbf{W}^{(k)} - \eta_k \sum_{t=1}^N \nabla L(\mathbf{W}; (x_t, y_t))$$

# Loss Functions

- Should match the metric we want to optimize at test time

# Loss Functions

- Should match the metric we want to optimize at test time
- Should be well-behaved (convex, hopefully smooth) for optimization (rules out the 0/1 loss)

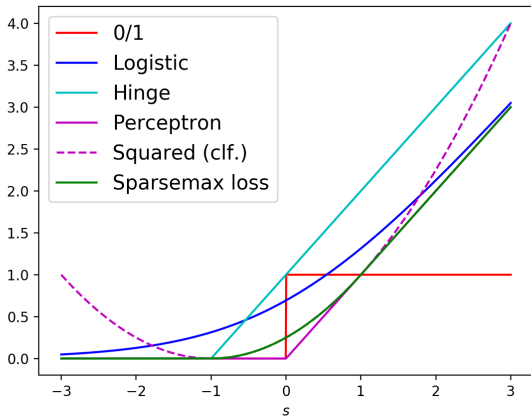
# Loss Functions

- Should match the metric we want to optimize at test time
- Should be well-behaved (convex, hopefully smooth) for optimization (rules out the 0/1 loss)
- Some examples:
  - ✓ Squared loss for regression
  - ✓ Negative log-likelihood (a.k.a. cross-entropy) for multinomial logistic regression
  - ✓ Hinge loss for support vector machines (not covered)
  - ✓ Sparsemax loss for multi-class and multi-label classification (not covered)



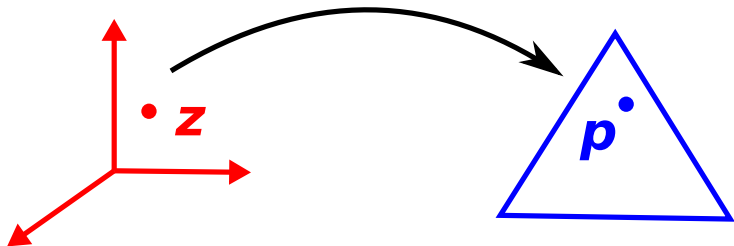
# Classification Losses (Binary Case)

- The horizontal axis shows  $s = -y w^T \phi(x)$ .



## Recap

Mapping a label score vector  $z \in \mathbb{R}^{|\mathcal{Y}|}$  to a probability distribution in  $\mathcal{Y}$ ?



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:

$$\mathbf{softmax} : \mathbb{R}^{|\mathcal{Y}|} \rightarrow \Delta_{|\mathcal{Y}|-1}$$

$$\mathbf{softmax}(z) = \left[ \frac{\exp(z_1)}{\sum_c \exp(z_c)}, \dots, \frac{\exp(z_{|\mathcal{Y}|})}{\sum_c \exp(z_c)} \right]^T$$

## Recap: Softmax Transformation

The typical transformation for multi-class classification:

$$\mathbf{softmax} : \mathbb{R}^{|\mathcal{Y}|} \rightarrow \Delta_{|\mathcal{Y}|-1}$$

$$\mathbf{softmax}(z) = \left[ \frac{\exp(z_1)}{\sum_c \exp(z_c)}, \dots, \frac{\exp(z_{|\mathcal{Y}|})}{\sum_c \exp(z_c)} \right]^T$$

- Underlies **multinomial logistic regression**

## Recap: Softmax Transformation

The typical transformation for multi-class classification:

$$\mathbf{softmax} : \mathbb{R}^{|\mathcal{Y}|} \rightarrow \Delta_{|\mathcal{Y}|-1}$$

$$\mathbf{softmax}(z) = \left[ \frac{\exp(z_1)}{\sum_c \exp(z_c)}, \dots, \frac{\exp(z_{|\mathcal{Y}|})}{\sum_c \exp(z_c)} \right]^T$$

- Underlies **multinomial logistic regression**
- Strictly positive, sums to 1

## Recap: Softmax Transformation

The typical transformation for multi-class classification:

$$\mathbf{softmax} : \mathbb{R}^{|y|} \rightarrow \Delta_{|y|-1}$$

$$\mathbf{softmax}(z) = \left[ \frac{\exp(z_1)}{\sum_c \exp(z_c)}, \dots, \frac{\exp(z_{|y|})}{\sum_c \exp(z_c)} \right]^T$$

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

## Recap: Multinomial Logistic Regression

- The common choice for a softmax output layer

## Recap: Multinomial Logistic Regression

- The common choice for a softmax output layer
- The classifier estimates  $P(y \mid x; \mathbf{W})$



## Recap: Multinomial Logistic Regression

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

$$\begin{aligned}L(\mathbf{W}; (x, y)) &= -\log P(y | x; \mathbf{W}) \\ &= -\log (\mathbf{softmax}(z(x)))_y,\end{aligned}$$

where  $z_c(x) = w_c^T \phi(x)$  is the score of class  $c$  for sample  $x$ .

## Recap: Multinomial Logistic Regression

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

$$\begin{aligned}L(\mathbf{W}; (x, y)) &= -\log P(y | x; \mathbf{W}) \\ &= -\log (\mathbf{softmax}(z(x)))_y,\end{aligned}$$

where  $z_c(x) = w_c^T \phi(x)$  is the score of class  $c$  for sample  $x$ .

- Loss gradient:

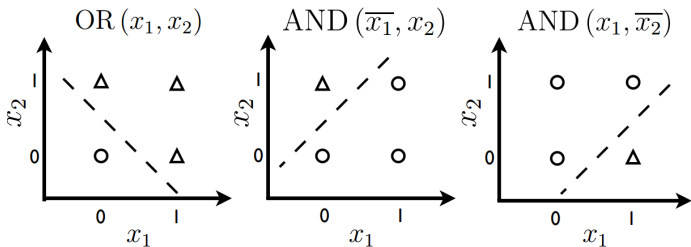
$$\nabla L(\mathbf{W}; (x, y)) = (\mathbf{softmax}(z(x)) - e_y) \phi(x)^T$$

# Outline

- ① Logistic Regression
- ② Regularization
- ③ Non-Linear Classifiers

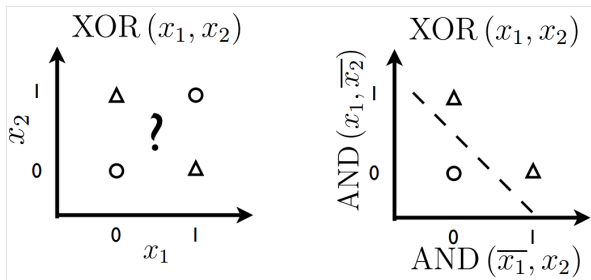
# Recap: What a Linear Classifier Can Do

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



## Recap: What a Linear Classifier **Can't** Do

- ... but it **cannot** 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 have seen
  - ✓ Perceptron
  - ✓ Logistic regression
  - ✓ Support vector machines (not covered)

# Summary: Linear Classifiers

- We have seen
  - ✓ Perceptron
  - ✓ Logistic regression
  - ✓ Support vector machines (not covered)
- All lead to **convex** optimization problems: no issues with local minima.

# Summary: Linear Classifiers

- We have seen
  - ✓ Perceptron
  - ✓ Logistic regression
  - ✓ Support vector machines (not covered)
- All lead to **convex** optimization problems: no issues with local minima.
- 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 object properties (features) and build models that manipulate them.
  - ✓ What that we have seen so far.

# Two Views of Machine Learning

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

- 1 **Feature-based**: describe object properties (features) and build models that manipulate them.
  - ✓ What 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.

# Two Views of Machine Learning

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

- 1 **Feature-based**: describe object properties (features) and build models that manipulate them.
  - ✓ What 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(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters



# Nearest Neighbor(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- No training, just **memorize** the data  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$

# Nearest Neighbor(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- No training, just **memorize** the data  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$
- Given a new input  $x$ , find its **nearest** data point  $x_i$  and predict

$$\hat{y}(x) = y_n, \text{ where } n = \arg \min_i \|x - x_i\|$$

# Nearest Neighbor(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- No training, just **memorize** the data  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$
- Given a new input  $x$ , find its **nearest** data point  $x_i$  and predict

$$\hat{y}(x) = y_n, \text{ where } n = \arg \min_i \|x - x_i\|$$

- Variants:  $k$ -NN returns the majority class in the  $k$  nearest neighbours.

# Nearest Neighbor(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- No training, just **memorize** the data  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$
- Given a new input  $x$ , find its **nearest** data point  $x_i$  and predict

$$\hat{y}(x) = y_n, \text{ where } n = \arg \min_i \|x - x_i\|$$

- Variants:  $k$ -NN returns the majority class in the  $k$  nearest neighbours.
- **Disadvantage:** requires searching over the entire training data

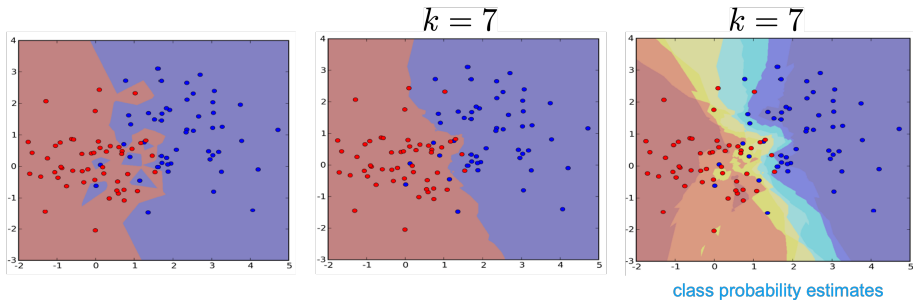
# Nearest Neighbor(s) (NN) Classifier

- Not a linear classifier!
- In its simplest version, doesn't require any parameters
- No training, just **memorize** the data  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$
- Given a new input  $x$ , find its **nearest** data point  $x_i$  and predict

$$\hat{y}(x) = y_n, \text{ where } n = \arg \min_i \|x - x_i\|$$

- Variants:  $k$ -NN returns the majority class in the  $k$  nearest neighbours.
- **Disadvantage:** requires searching over the entire training data
- Specialized data structures can be used to speed up search.

# Nearest Neighbour(s) Classifiers



# Kernels

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

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

# Kernels

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

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

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

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



# Kernels

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

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

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

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

- **Symmetric:**

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

# Kernels

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

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

- Given dataset  $\mathcal{D} = \{(x_i, y_i), i = 1, \dots, N\}$ , the **Gram matrix**  $\mathbf{K}$  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  $\mathbf{v}$

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

# Kernels

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

$$\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

# Kernels

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

$$\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

- A kernel corresponds to a mapping into an **implicit** feature space.

# Kernels

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

$$\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

- A kernel corresponds to a mapping into an **implicit** feature space.
- The kernel is equivalent to an inner product in that feature space.

# Kernels

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

$$\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

- A kernel corresponds to a mapping into an **implicit** feature space.
- The kernel is equivalent to an inner product in that feature space.
- **Kernel trick:** take a feature-based algorithm (SVM, perceptron, logistic regression) and replace all explicit feature computations by **kernel evaluations**:

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

# Kernels

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

$$\kappa(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

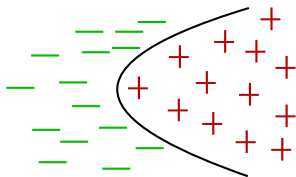
- A kernel corresponds to a mapping into an **implicit** feature space.
- The kernel is equivalent to an inner product in that feature space.
- **Kernel trick:** take a feature-based algorithm (SVM, perceptron, logistic regression) and replace all explicit feature computations by **kernel evaluations**:

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

- Extremely popular idea in the 1990-2000s!

# Kernels = Tractable Non-Linearity

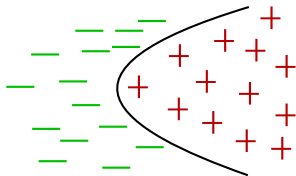
- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space





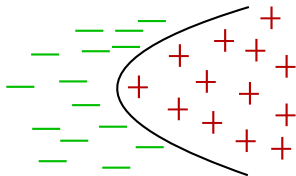
# Kernels = Tractable Non-Linearity

- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space
- Computing a non-linear kernel is often cheaper than calculating the corresponding inner product in the high-dimensional feature space.



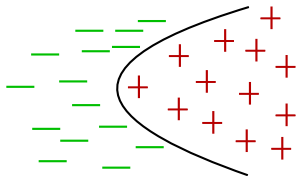
# Kernels = Tractable Non-Linearity

- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space
- Computing a non-linear kernel is often cheaper than calculating the corresponding inner product in the high-dimensional feature space.
- Many models can be “kernelized”.

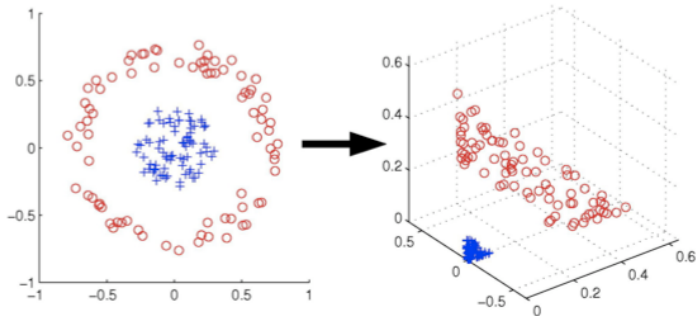


# Kernels = Tractable Non-Linearity

- A linear classifier in a higher dimensional feature space is a non-linear classifier in the original space
- Computing a non-linear kernel is often cheaper than calculating the corresponding inner product in the high-dimensional feature space.
- Many models can be “kernelized”.
- Drawback: **quadratic** dependency on dataset size



# Linear Classifiers in High Dimension



- Example:  $\phi([x_1, x_2]) = [x_1^2, x_2^2, \sqrt{2} x_1 x_2]$
- Kernel trick; let  $x = [x_1, x_2]$  and  $x' = [x'_1, x'_2]$ ;

$$\langle \phi(x), \phi(x') \rangle = \langle [x_1, x_2], [x'_1, x'_2] \rangle^2 = k(x, x')$$

# Popular Kernels

- Polynomial kernel of order  $d$ :

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle + 1)^d$$

# Popular Kernels

- Polynomial kernel of order  $d$ :

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle + 1)^d$$

- Gaussian radial basis kernel

$$\kappa(x, x') = \exp\left(\frac{-\|\phi(x) - \phi(x')\|^2}{2\sigma^2}\right)$$

# Popular Kernels

- Polynomial kernel of order  $d$ :

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle + 1)^d$$

- Gaussian radial basis kernel

$$\kappa(x, x') = \exp\left(\frac{-\|\phi(x) - \phi(x')\|^2}{2\sigma^2}\right)$$

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

# Popular Kernels

- Polynomial kernel of order  $d$ :

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle + 1)^d$$

- Gaussian radial basis kernel

$$\kappa(x, x') = \exp\left(\frac{-\|\phi(x) - \phi(x')\|^2}{2\sigma^2}\right)$$

- 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 methods such as **perceptrons**, **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.