CSC311 Notes

Jenci Wei

Fall 2022

Contents

1 Intro

Types of Machine Learning

- **Supervised learning**: have labelled examples of the correct behaviour
- **Reinforcement learning**: learning system (agent) interacts with the world and learns to maximize a scalar reward signal
- **Unsupervised learning**: no labelled examples, looks for interesting patterns in the data

Examples of Machine Learning

- Computer vision: object detection, semantic segmentation, etc.
- Speech: speech to text, personal assistants, etc.
- Natural language processing: machine translation, sentiment analysis, etc.
- Playing games
- Recommender systems

Supervised learning

- Given a **training set** consisting of **inputs** and corresponding **labels**

Input Vectors

- Represent the input as an *input vector* in $\mathbb{R}^d$
- *Representation*: mapping to another space that's easy to manipulate
- Training set consists of a collection of pairs of an input vector $\boldsymbol{x} \in \mathbb{R}^d$ and its corresponding *target/label* t
  - *Regression*: $t \in \mathbb{R}$
  - *Classification*: t is an element of a discrete set $\{1, \ldots, C\}$
- Denote the training set $\left\{(\boldsymbol{x}^{(1)}, t^{(1)}), \ldots, (\boldsymbol{x}^{(N)}, t^{(N)})\right\}$

2 Nearest Neighbour Methods

Nearest Neighbours

- Suppose we are given a novel in put vector $\boldsymbol{x}$ we want to classify
- Idea: find the nearest input vector to $\boldsymbol{x}$ in the training set and copy its label
- Can formalize "nearest" using Euclidean distance

$$\left\|\boldsymbol{x}^{(a)} - \boldsymbol{x}^{(b)}\right\|_2 = \sqrt{\sum_{i=1}^{d} (\boldsymbol{x}_i^{(a)} - \boldsymbol{x}_i^{(b)})^2}$$

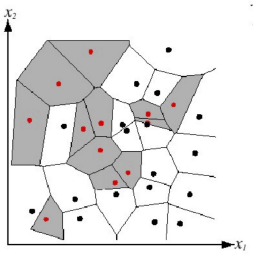
- Algorithm
  1. Find example $(\boldsymbol{x}^*, t^*)$ (from the stored training set) closest to $\boldsymbol{x}$, i.e.

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}^{(i)} \in \text{training set}}{\arg\min} \text{ distance}(\boldsymbol{x}^{(i)}, \boldsymbol{x})$$

  2. Output $y = t^*$

Decision Boundaries

- Visualise using a *Voronoi diagram*



- **Decision boundary**: the boundary between regions of input space assigned to different categories

k-Nearest Neighbours

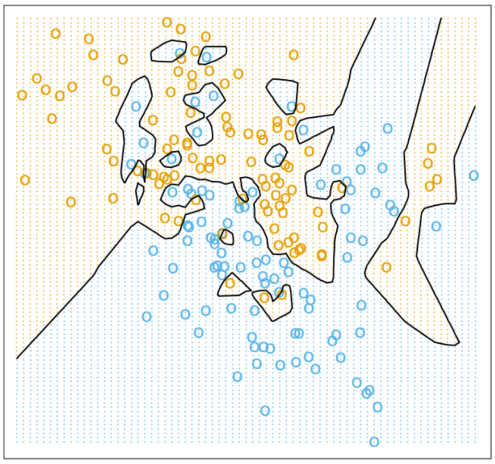
- Nearest neighbours is sensitive to noise or mislabelled data
- We can have the k nearest neighbours to "vote"
- Algorithm
  1. Find k examples $(\boldsymbol{x}^*, t^*)$ closest to the test instance $\boldsymbol{x}$
  2. Classification output is majority class

$$y = \underset{t^{(z)}}{\arg\max} \sum_{i=1}^{k} \mathbb{1}\left(t^{(z)} = t^{(i)}\right)$$

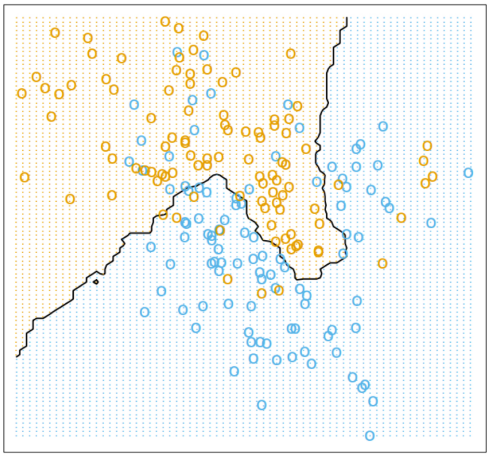
     - $\mathbb{1}$(statement) is the identity function which is equal to 1 whenever the statement is true, and 0 otherwise
     - Alternative notation: $\delta(t^{(z)}, t^{(i)}) \equiv \mathbb{1}\left(t^{(z)} = t^{(i)}\right)$

- Choosing k
  - Small k

    - Good at capturing fine-grained patterns
    - May overfit, i.e. sensitive to random noise in the training data
  - Large k
    - Makes stable predictions by averaging over lots of examples
    - May underfit, i.e. fail to capture important regularities
  - Balancing k
    - Optimal choice of k depends on tnumber of data points n
    - Choose $k < \sqrt{n}$
    - Choose k using validation set



(a) $k = 1$



(b) $k = 15$

Validation and Test Sets

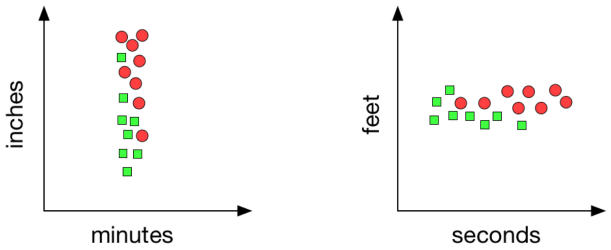
- Want our algorithm to *generalize* to data it has not seen before
- Can measure the **generalization error** (error rate on new examples) using a *test set*
- k is a *hyperparameter*, which is something we can't fit as part of the leraning algorithm itself
- We can tune hyperparameters using a **validation set**
- The test set is used only at the very end, to measure the generalization performance of the final configuration

The Curse of Dimensionality

- In high dimensions, most points are far apart, and are approximately the same distance
- Some dataset may have low **intrinsic dimension**, i.e. lie on or near a low-dimensional manifold
- The neighbourhood structure depends on the intrinsic dimension

Normalization

- Nearest neighbours can be sensitive to the ranges of different features



- We can *normalize* each dimension to be zero mean and unit variance
  - Compute mean μ_j and SD σ_j, and take

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

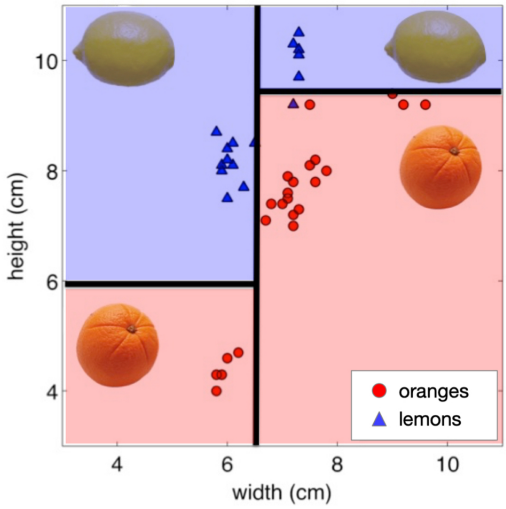
Computational Cost

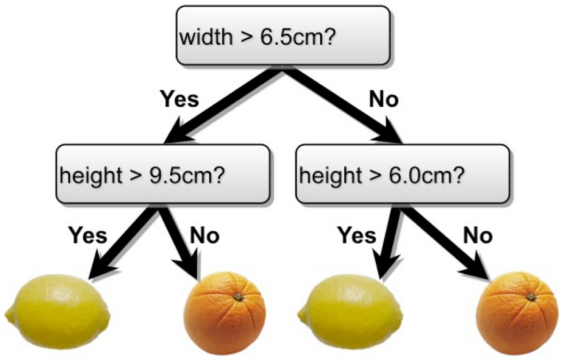
- Number of computations at training time: 0
- Number of computations at test time, per query:
  - Calculate D-dimensional Euclidean distances with N data points: $\mathcal{O}(ND)$
  - Sort the distances: $\mathcal{O}(N \log N)$
- Must be done for each query, expensive
- Need to store the entire dataset in memory

3 Decision Trees

Intuition

- Make predictions by splitting on features according to a tree structure
- Can split *continuous features* by checking whether that feature is greater than or less than some threshold
- Decision boundary is made up of axis-aligned planes





- Internal nodes test a *feature*
- Branching is determined by the *feature value*
- Leaf nodes are *outputs* (predictions)
- Hyperparameters: number of nodes, max depth, number of branches, etc.

Classification and Regression

- Each path from root to a leaf defines a region R_m of input space
- Let $\{(x^{(m_1)}, t^{(m_1)}), \ldots, (x^{(m_k)}, t^{(m_k)})\}$ be the training examples that fall into R_m
- **Regression tree**:
  - Continuous output
  - Leaf value y^m typically set to the mean value in $\{t^{(m_1)}, \ldots, t^{(m_k)}\}$
- **Classification tree**
  - Discrete output
  - Leaf value y^m typically set to the most common value in $\{t^{(m_1)}, \ldots, t^{(m_k)}\}$

Learning Decision Trees

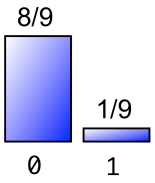
- Decision trees are *universal function approximators* (i.e. can approximate any function)
  - For any training set, we can construct a decision tree that has exactly 1 leaf for every training point, though it won't generalize

- Finding the smallest (i.e. optimal) decision tree that correctly classifies a training set is NP complete
- Can use the *greedy heuristic*
  - Start with the whole training set and an empty decision tree
  - Pick a feture and candidate split that maximally reduces a loss
  - Split on that feature and recurse on subpartitions
- Loss
  - Scalar number that assess whether we are on tract
  - Low is good, high is bad

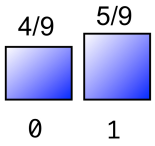
Entropy

- The **entropy** of a discrete RV is a number that quantifies the *uncertainty* inherent in its possible outcomes
- The entropy of a loaded coin with probability p of heads is given by

$$-p\log_2(p) - (1-p)\log_2(1-p)$$



$$-\frac{8}{9}\log_2\frac{8}{9} - \frac{1}{9}\log_2\frac{1}{9} \approx \frac{1}{2}$$



$$-\frac{4}{9}\log_2\frac{4}{9} - \frac{5}{9}\log_2\frac{5}{9} \approx 0.99$$

- The coin whose outcomes are more certain has a lower entropy
- In the case of $p = 0$ or $p = 1$, the entropy is 0 since we gain no certainty by observing it
- Entropy is the expected information content of a random draw from a propability distribution
- Cannot store the outcome of a random draw using fewer expected bits than the entropy without losing information
- Unit: **bits**, i.e. a fair coin flip has 1 bit of entropy
- Entropy of a discrete RV Y is given by

$$H(Y) = -\sum_{y \in Y} p(y)\log_2 p(y)$$

- Example of high entropy
  - Variable has a uniform-like distribution over many outcomes
  - Flat histogram
  - Values samples from it are less predictable
- Example of low entropy
  - Distribution is concentrated on only a few outcomes
  - Histogram is concentrated in a few areas

– Values sampled from it are more predictable

Entropy of a Joint Distribution

$$H(X,Y) = -\sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 p(x,y)$$

Conditional Entropy

$$H(Y|X=x) = -\sum_{y\in Y} p(y|x)\log_2 p(y|x)$$

where $p(y|x) = \frac{p(x,y)}{p(x)}$ and $p(x) = \sum_y p(x,y)$

Expected Condition Entropy

$$\begin{aligned} H(Y|X) &= \mathbb{E}_x[H[Y|x]] \\ &= \sum_{x\in X} p(x)H(Y|X=x) \\ &= -\sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 p(y|x) \end{aligned}$$

Properties of Entropy

- H is non-negative
- $H(X,Y) = H(X|Y) + H(Y) = H(Y|X) + H(X)$
- If X and Y are independent, then X does not affect our uncertainty about Y, so $H(Y|X) = H(Y)$
- Knowing Y makes our knowledge of Y certain, i.e. $H(Y|Y) = 0$
- By knowing X, we canonly decrease uncertaintyp about Y: $H(Y|X) \leq H(Y)$

Information Gain

- The **information gain** in Y due to X, or the **mutual information** of Y and X, is

$$IG(Y|X) = H(Y) - H(Y|X) \geq 0$$

- If X is completely uninformative about Y, then $IG(Y|X) = 0$
- If X is completely informative about Y, then $IG(Y|X) = H(Y)$

Constructing Decision Trees

- Information gain of a split: how much information (over the training set) about the class label Y is gained by knowing which side of a split we are on
- At each level, choose
  1. Which feature to split
  2. Where to split
- Algorithm (simple, greedy, recursive approach, builds up tree node-by-node)
  1. Pick a feature to split at a non-terminal node
  2. Split examples into groups based on feature value
  3. For each group:

    - If no examples, return majority from parent
    - Else if all examples in same class, return that class
    - Else loop to step 1
- Terminates when all leaves contain only examples in the same class or are empty

A Good Tree

- Not too small: need to handle important but possibly subtle distinctions in data
- Not too big
    - Computational efficiency
    - Avoid over-fitting training examples
    - Human interpretability
- *Occam's razer*: find the simplest hypothesis that fits the observations
- Want small trees with informative nodes near the root
- Problems
    - Exponentially less data at lower levels
    - Too big of a tree overfits data
    - Greedy algorithms don't necessarily yield the global optimum
- Handling continuous attributes: split based on a threshold, chosen to maximize information gain
- For regression, we can choose splits to minimize squared error, rather than maximize information gain

KNN vs. Decision Trees

- Advantages of decision trees
    - Simple to deal with discrete features, missing values, and poorly scaled data
    - Fast at test time
    - More interpretable
- Advantages of KNN
    - Few hyperparameters (i.e. 1)
    - Can incorporate interesting distance measures (e.g. shape contexts)

4 Bias-Variance Decomposition

Basic Setup for Classification

- p_{sample} is a *data generating distribution* (e.g. for lemons and oranges, p_{sample} characterizes heights and widths)
- Pick a fixed query point $\boldsymbol{x}$, we want to get a prediction y at $\boldsymbol{x}$
- A training set $\mathcal{D}$ consists of pairs $(\boldsymbol{x}_i, t_i)$ sampled i.i.d. from p_{sample}
- Can sample lots of training sets independently from p_{sample}
- Run learning algorithm on each training set, and compute its prediction y at the query point $\boldsymbol{x}$
- Can view y as a RV, where randomness comes from the choice of training set
- Classification accuracy determined by the distribution of y
- Repeat the following:
  1. Sample a random training dataset $\mathcal{D}$ i.i.d. from p_{sample}
  2. Run the learning algorithm on $\mathcal{D}$ to get a prediction y at $\boldsymbol{x}$
  3. Sample the (true) target t from the conditional distribution $p(t|\boldsymbol{x})$
     - Not constant, e.g. same height and width can be either lemon or orange
  4. Compute the loss $L(y, t)$
- y is independent of t
- This gives a distribution over the loss at $\boldsymbol{x}$, with expectation $\mathbb{E}[L(y, t)|\boldsymbol{x}]$
- For each query point $\boldsymbol{x}$, the expected loss is different, want to minimize this expectation w.r.t. $\boldsymbol{x} \sim p_{\text{sample}}$
- Consider the squared error loss, $L(y, t) = \frac{1}{2}(y - t)^2$

Lemma 4.1. *$y_* = \mathbb{E}[t|\boldsymbol{x}]$ is the best possible prediction*

Proof. (sketch) After derivation,

$$\mathbb{E}[(y - t)^2|\boldsymbol{x}] = (y - y_*)^2 + \text{Var}[t|\boldsymbol{x}]$$

□

- First term is nonnegative, can be made 0 by choosing $y_* = y$
- Second term is **Bayes error**, or the noise or inherent unpredictability of the target t
  - An algorithm that achieves this is **Bayes optimal**
  - This term does not depend on y
- Process of choosing a single value y_* based on $p(t|\boldsymbol{x})$ is an example of *decision theory*

Bayes Optimality

- We can decompose the expected loss as follows:

$$\mathbb{E}[(y - t)^2] = (y_* - \mathbb{E}[y])^2 + \text{Var}(y) + \text{Var}(t)$$

- The first term is the **bias**, which is how wrong the expected prediction is (corresponds to underfitting)
- The second term is the **variance**, which is the amount of variability in the predictions (corresponds to overfitting)
- The third term is the **Bayes error**, which is the inherent unpredictability of the targets
- An overly simple model might have high bias and low variance
- An overly complex model might have low bias and high variance

5 Bagging

Motivation

- Consider the following procedure:
  1. Sample m independent training sets from p_{sample}
  2. Compute the prediction y_i using each training set
  3. Compute the average prediction $y = \frac{1}{m}\sum_{i=1}^{m} y_i$
- Bias is unchanged since the averaged prediction has the same expectation
- Variance is reduced since we are averaging over independent predictions (to $1/m$ of original)
- Bayes error is unchanged since we have no control over it

Idea

- p_{sample} is expensive to sample from
- Given training set $\mathcal{D}$, use the empirical distribution $p_{\mathcal{D}}$ as a proxy for p_{sample}. This is called **bootstrap aggregation** or **bagging**
  1. Take a dataset $\mathcal{D}$ with n examples
  2. Generate m new datasets (bootstrap samples)
  3. Each dataset has n examples sampled from $\mathcal{D}$ *with replacement*
  4. Average the predictions of models trained on the m datasets
- Intuition: $|\mathcal{D}| \to \infty \implies p_{\mathcal{D}} \to p_{\text{sample}}$
- For binary classification: (majory vote)

$$y_{\text{bagged}} = \mathbb{1}\left(\frac{1}{m}\sum_{i=1}^{m} y_i \geq 0.5\right)$$

Properties

- A bagged classifier can be stronger than the average model
- If m datasets are **not** independent, we don't get the $\frac{1}{m}$ variance reduction (i.e. sampling from the same pool)
- We need to reduce correlation between datasets by introducing additional variability (e.g. average over multiple algorithms, or multiple configurations of the same algorithm)
- Weighting members equally may not be the best, leads to *weighted ensembling*

Random Forests

- We can reduce correlation between bagged decision trees
- For each node, we can choose a random subset of features to consider split upon
- Works well with no tuning

6 Linear Regression

Intro

- Task: predict scalar-valued targets
- Architecture: linear function of the inputs

Approach

1. Choose a *model* describing relationships between variables
2. Define a *loss function* quantifying how well the model fits the data
3. Choose a *regularizer* expressing preference over different models
4. Fit a model taht minimizes the loss function and satisfies the regularizer's constraint/penalty, possibly using an *optimization algorithm*

Supervised Learning Setup

- Input $\boldsymbol{x} \in \mathcal{X}$ (a vector of features)
- Target $t \in \mathcal{T}$
- Data $\mathcal{D} = \left\{(\boldsymbol{x}^{(i)}, t^{(i)}) : i = 1, \ldots, n\right\}$
- Objective: learn a function $f : \mathcal{X} \to \mathcal{T}$ based on the data such that $t \approx y = f(\boldsymbol{x})$

Model

- A *linear* function of the features $\boldsymbol{x} = (x_1, \ldots, x_D) \in \mathbb{R}^D$ to make a prediction $y \in \mathbb{R}$ of the target $t \in \mathbb{R}$

$$y = f(\boldsymbol{x}) = \sum_j w_j x_j + b = \boldsymbol{w}^\top \boldsymbol{x} + b$$

- *Parameters* are weights $\boldsymbol{w}$ and the bias/intercept b
- Want the prediction to be close to the target: $y \approx t$

Loss Function

- *Loss function* $\mathcal{L}(y, t)$ defines how badly the algorithm's prediciton y fits the target t for some example $\boldsymbol{x}$
- Squared error loss function: $\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$
  - $y - t$ is the *residual*, whose magnitude we want to minimize
  - $1/2$ makes differentiation convenient
- *Cost function*: loss function averaged over all training examples, also called *empirical loss* or *average loss*

$$\mathcal{J}(\boldsymbol{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)}\right)^2 = \frac{1}{2N} \sum_{i=1}^{N} \left(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b - t^{(i)}\right)^2$$

Vectorization

- We can express algorithms using vectors and matrices (rather than loops)
- Code is simpler and more readable

- Vectorized code is much faster
- Put training examples into a *design matrix* $\boldsymbol{X}$
  - Each row is an example
  - Each column is a feature
- Put targets into a *target vector*

$$\boldsymbol{X}\boldsymbol{w} + b\mathbf{1} = y$$

$$\begin{bmatrix} x_1^{(1)} & \cdots & x_D^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(N)} & \cdots & x_D^{(N)} \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_D \end{bmatrix} + b \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

- Squared error cost:

$$\mathcal{J} = \frac{1}{2N} \|\boldsymbol{y} - \boldsymbol{t}\|^2$$

- Can combine the bias and the weights

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}$$

where

$$\boldsymbol{X} = \begin{bmatrix} 1 & [\boldsymbol{x}^{(1)}]^\top \\ \vdots & \vdots \\ 1 & [\boldsymbol{x}^{(N)}]^\top \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \quad \text{and} \quad \boldsymbol{w} = \begin{bmatrix} b \\ \boldsymbol{w} \end{bmatrix} \in \mathbb{R}^{D+1}$$

Optimization

- Goal: minimize the cost function $\mathcal{J}(\boldsymbol{w})$
- The minimum of a smooth function occurs at a *critical point*, i.e.

$$\nabla_{\boldsymbol{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} = \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{bmatrix} = \mathbf{0}$$

- *Direct solution*: set the gradient to 0 and solve in closed form
- *Iterative solution*: repeatedly apply an update rule that gradually takes us closer to the solution

Direct Solution for Linear Regression

- Want $\boldsymbol{w}$ to minimize $\mathcal{J}(\boldsymbol{w}) = \frac{1}{2} \|\boldsymbol{y} - \boldsymbol{t}\|^2$ (i.e. sum, not average)
- Optimal weights after some derivation:

$$\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{t}$$

Iterative Solution: Gradient Descent

- Gradient descent is an *iterative algorithm*, which means we apply an update repeatedly until some criterion is met
- We *initialize* the weights to some reasonable values (e.g. all zeros) and repeated ly adjust them in the *direction of steepest descent*, which is the gradient

- Gradient descent update rule:

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

  where $\alpha > 0$ is the **learning rate** (or step size)

  - The larger α is, the faster $\boldsymbol{w}$ changes
  - If we minimize the total loss rather than the average loss, then we need a smaller learning rate $\alpha' = \alpha/N$

- Update rule in vector form:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}}$$

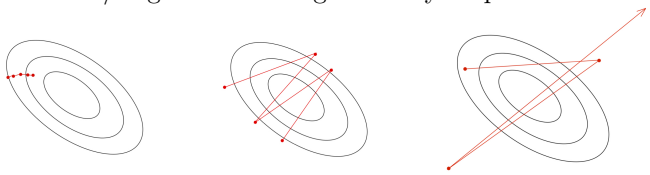
- Update rule for linear regression:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \boldsymbol{x}^{(i)}$$

- Once it converges, we get a critical point, i.e. $\frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} = \mathbf{0}$

Finding a Good Learning Rate

- Too small/large of a learning rate may be problematic

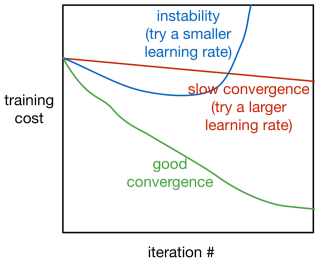


α too small: slow progress

α too large: oscillations

α much too large: instability

- Good values are typically between 0.001 and 0.1
- Do a grid search for good performance (i.e. try 0.1, 0.03, 0.01, etc.)
- Diagnose optimization problems using a training curve



Gradient Descent vs. Direct Solution

- Closed form solution requires matrix inversion, which is $\mathcal{O}(D^3)$ time
- Gradient descent update costs $\mathcal{O}(ND)$, or less with stochastic gradient descent
- Gradient descent is more efficient than direct solution for regression in high-dimensional space

Feature Mappings

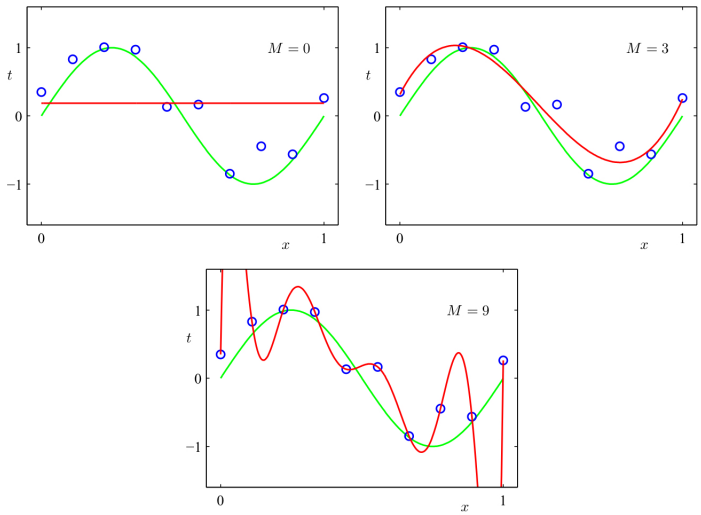
- We can model a nonlinear relationship using linear regression by mapping the input features to another space $\boldsymbol{\psi}(\boldsymbol{x}) : \mathbb{R}^D \to \mathbb{R}^d$, and treat the mapped feature as the input of a linear regression procedure

Polynomial Feature Mapping

- Fit the data using a degree-M polynomial function of the form

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{i=0}^{M} w_i x^i$$

- The feature mapping is $\boldsymbol{\psi}(x) = [1, x, x^2, \ldots, x^M]^\top$
- $y = \boldsymbol{\psi}(x)^\top \boldsymbol{w}$ is linear in $(1, x, x^2, \ldots, x^M)$
- Use linear regression to find $\boldsymbol{w}$
- When M is small, we are underfitting (high bias, low variance)
- When M is large, we are overfitting (low bias, high variance)
- If we pick a good M, we get a small test error and the model generalizes well



- As M increases, the magnitude of coefficients gets larger (i.e. $w_0, w_1, \ldots, w_M$ becomes large), so that the function exhibits large oscillations between data points

Regularization

- We can keep the model large, but *regularize* it
- **Regularizer**: a function that quantifies how much we prefer one hypothesis vs. another

L^2 (or l_2) Regularization

- Encourage the weights to be small by choosing the L^2 penality as our regularizer

$$\mathcal{R}(\boldsymbol{w}) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 = \frac{1}{2} \sum_j w_j^2$$

- The regularized cost function makes a tradeoff between the fit to the data and the norm of the weights

$$\mathcal{J}_{\text{reg}}(\boldsymbol{w}) = \mathcal{J}(\boldsymbol{w}) + \lambda \mathcal{R}(\boldsymbol{w}) = \mathcal{J}(\boldsymbol{w}) + \frac{\lambda}{2} \sum_j w_j^2$$

  - If we fit training data poorly, $\mathcal{J}$ is large
  - If the weights are large in magnitude, $\mathcal{R}$ is large
  - Large λ penalizes weight values more
  - λ is a hyperparameter

L^2 Regularized Least Squares: Ridge Regression

- For the least squares problem, we have $\mathcal{J}(\boldsymbol{w}) = \frac{1}{2N} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{t}\|^2$
- With $\lambda > 0$, regularized cost gives

$$\boldsymbol{w}_\lambda^{\text{Ridge}} = \arg\min_{\boldsymbol{w}} \mathcal{J}_{\text{reg}} = \left(\boldsymbol{X}^\top \boldsymbol{X} + \lambda N \boldsymbol{I}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{t}$$

- When $\lambda = 0$, we get the same solution as least squares
- Can also formulate to the total loss problem and get the solution of

$$\boldsymbol{w}_\lambda^{\text{Ridge}} = \left(\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1} \boldsymbol{X}^\top \boldsymbol{t}$$

Gradient Descent Under the L^2 Regularization

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial}{\partial \boldsymbol{w}} (\mathcal{J} + \lambda \mathcal{R}) = (1 - \alpha\lambda)\boldsymbol{w} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}}$$

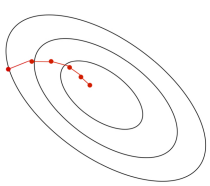
- Results in *weight decay*

Stochastic Gradient Descent

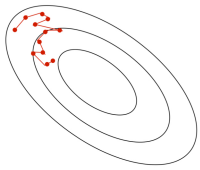
- Updates the parameters based on the gradient for *one* training example
- Algorithm
  1. Choose example i uniformly at random
  2. Perform update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$
- Cost of each update is cheap and independent of N
- Makes significant progress before seeing all the data
- Is an unbiased estimate if we sample each example uniformly at random

$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}\right] = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}$$

- High variance in the estimate (i.e. each step on their own may not be in the right direction)
- Cannot exploit efficient vectorized operations



Batch GD
moves downhill at each step.



Stochastic GD
moves in a noisy direction,
but downhill on average.

Mini-Batch Gradient Descent

- Compute each gradient on a *subset* of examples
- Not to be confused with *batch gradient descent*, which is just the ordinary gradient descent
- *Mini-batch*: a randomly chosen medium-sized subset of training examples $\mathcal{M}$
- We can permute the training set and then go through it sequentially
- Each pass over the data is called an **epoch**
- Mini-batch sizes

Large	Small
More computation time	Faster updates
Estimates accurate	Estimates noisy
Can exploit vectorization	Cannot exploit vectorization

- $|\mathcal{M}|$ is a hyperparameter
- A reasonable value is $|\mathcal{M}| = 100$

Setting Learning Rate for Stocastic GD

- Start with a large learning rate to get close to the optimum
- Gradually decrease the learning rate to reduce the fluctuations

7 Binary Linear Classification

Binary Linear Classification

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b$$
$$y = \begin{cases} 1, & \text{if } z \geq r \\ 0, & \text{if } z < r \end{cases}$$

- *Classification*: predict a discrete-valued target given a D-dimensional input $\boldsymbol{x} \in \mathbb{R}^D$
- *Binary*: predict a binary target $t \in \{0, 1\}$
  - $t = 1$ is a positive example
  - $t = 0$ is a negative example
  - Sometimes -1 is used instead of 0
- *Linear*: prediction y is a linear function of $\boldsymbol{x}$

Simplified Model

$$z = \boldsymbol{w}^\top \boldsymbol{x}$$
$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

- We can eliminate the threshold r by modifying $b \leftarrow b - r$
- We can eliminate the bias b by adding a dummy feature $x_0 = 1$. The weight $w_0 = b$ is equivalent to a bias

Modelling NOT

x_0	x_1	t
1	0	1
1	1	0

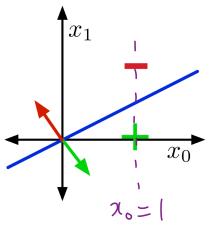
$$z = w_0 x_0 + w_1 x_1$$
$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

- Derive a set of values for w_0, w_1 that guarantee perfect classification

$$\begin{cases} w_0 \times 1 + w_1 \times 0 \geq 0 \\ w_0 \times 1 + w_1 \times 1 < 0 \end{cases}$$

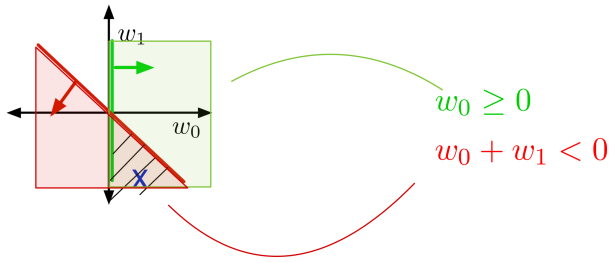
- E.g. $w_0 = 1, w_1 = -2$

Visualizing NOT in Data Space



- Each point is a *training example*
- The axes are the xs
- The line $\{\boldsymbol{x} : \boldsymbol{w}^\top \boldsymbol{x} = 0\}$ defines the decision boundary
- Data is *linearly separable* if a linear decision rule can perfectly separate the training examples

Visualizing NOT in Weight Space



- Each point is a *set of values for weights* $\boldsymbol{w}$
- Each training example $\boldsymbol{x}$ specifies a half-space that $\boldsymbol{w}$ must lie in to guarantee correct classification
- The *feasible region* satisfies all constraints
- The problem is *feasible* is the feasible region is nonempty

Learning the Weights for Linearly Separable Data

- Use linear programming
- Use the perceptron algorithm
- However, data is almost never linearly separable

8 Logistic Regression

Data Being Not Linearly Separable

- Can define a loss function, then find weights that minimize the average loss over the training examples

Model

$$z = \boldsymbol{w}^\top \boldsymbol{x}$$
$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

0-1 Loss

$$\mathcal{L}_{0-1}(y, t) = \begin{cases} 0, & \text{if } y = t \\ 1, & \text{if } y \neq t \end{cases} = \mathbb{1}[y \neq t]$$

- The cost $\mathcal{J}$ is the *misclassification rate*

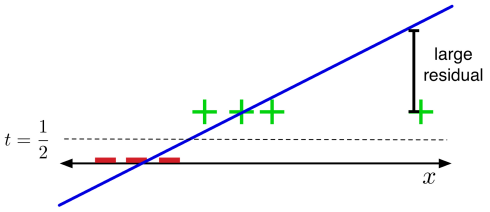
$$\mathcal{J} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[y^{(i)} \neq t^{(i)}]$$

- Gradient is 0 almost everywhere
- Discontinuous at $z = 0$, where the gradient is undefined

Squared Loss for Linear Regression

$$\mathcal{L}_{\text{SE}}(z, t) = \frac{1}{2}(z - t)^2$$

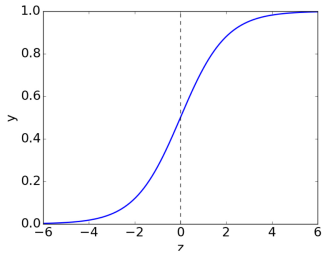
- Treat the binary targets as continuous values
- Make final predictions y by thresholding z at $1/2$
- Making a correct prediction with high confidence incurs a large loss



Logistic Activation Function

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$
$$\mathcal{L}_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2$$

- Squash predictions y into $[0, 1]$
- The **logistic function** is a **sigmoid** (S-shaped) function

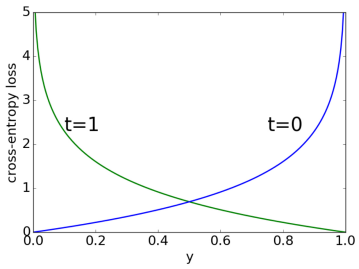


- σ is called an *activation function*
- Suppose that $t = 1$ and z is very negative (i.e. $z \ll 0$), then the prediction $y \approx 0$ is very wrong
- However, the weights appear to be at a critical point

Cross-Entropy Loss

$$\mathcal{L}_{\text{CE}}(y, t) = \begin{cases} -\log y, & \text{if } t = 1 \\ -\log(1-y), & \text{if } t = 0 \end{cases} = -t \log y - (1-t) \log(1-y)$$

- Interpret $y \in [0, 1]$ as the estimated probability that $t = 1$
- Heavily penalize the extreme misclassification cases that leads to problems with only the logistic function
- Also known as *log loss*



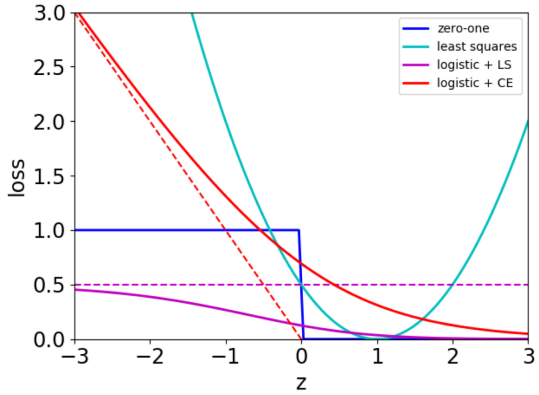
Numerical Instabilities

- If $z \ll 0$ and y is small enough, y may become numerically zero, then we are apply log to 0
- We can combine the logistic activation function and the cross-entropy loss into a single *logistic-cross-entropy* function

$$\mathcal{L}_{\text{LCE}}(z, t) = \mathcal{L}_{\text{CE}}(\sigma(z), t) = t \log(1 + e^{-z}) + (1-t) \log(1 + e^{z})$$

```
E = t * np.logaddexp(0, -z) + (1 - t) * np.logaddexp(0, z)
```

Comparison of Loss Functions for $t = 1$



Gradient Descent for Logistic Regression

- Apply chain rule and obtain

$$\frac{\partial \mathcal{L}_{\text{CE}}}{\partial w_j} = (y - t)x_j$$

- Update rule:

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} = w_j - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)}$$

  - Similar to linear regression, but now $y^{(i)} = \dfrac{1}{1 - e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}$ rather than $\boldsymbol{w}^\top \boldsymbol{x}^{(i)}$

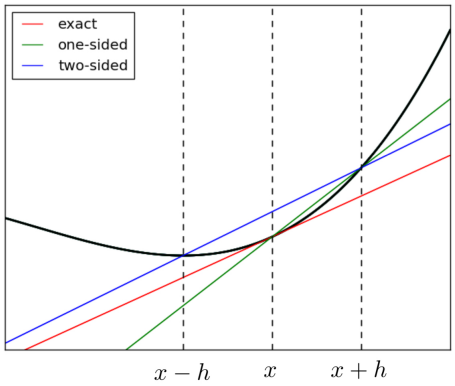
Gradient Checking

- We want to check that our implementation of the gradient is correct
- One-sided definition of partial derivative:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}$$

- Two-sided definition of partial derivative:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h}$$



- Can run gradient checks on small, randomly chosen inputs, using *double precision* floats
- Plug in a small value of h, e.g. 10^{-10}
- Compute the *relative error*

$$\frac{|a - b|}{|a| + |b|}$$

  where a is the finite differences estimate and b is the derivative computed by our implementation
- The relative error should be small, e.g. 10^{-6}
- Important since even if the math is wrong, learning algorithms might seem to work

Hypothesis Space and Inductive Bias

- A **hypothesis** is a function $f : \mathcal{X} \to \mathcal{T}$ from the input to target space
- The hypothesis space $\mathcal{H}$ is a set of hypotheses

  - In linear regression, $\mathcal{H}$ is the set of linear functions

- A machine learning algorithm aims to find a good hypothesis $f \in \mathcal{H}$
- An algorithm's **inductive bias** is the members of $\mathcal{H}$ and its preference for some hypotheses of $\mathcal{H}$ over others
- *No Free Lunch Theorem*: if datasets/problems were not naturally biased, no machine learning algorithm would be better than another
  - Learning is not possible without bias/prior knowledge

Parametric vs. Non-Parametric Algorithms

- *Parametric* algorithm: the hypothesis space $\mathcal{H}$ is defined using a finite set of parameters
  - E.g. linear regression, logistic regression
- *Non-parametric* algorithm: the hypothesis space $\mathcal{H}$ is defined in terms of the data (no parameters)
  - E.g. k-nearest neighbors, decision trees

9 Softmax Regression

Multi-Class Classification

- Task is to predict a discrete (> 2)-valued target
- Targets form a discrete set $\{1, \ldots, K\}$
- Represent targets as **one-hot vectors** or **one-of-K** encoding:

  $$\boldsymbol{t} = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^K$$

  where the entry k has value 1

Linear Function of Inputs

- Vectorized form:

  $$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}, \quad \text{or} \quad \boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} \quad \text{with dummy } x_0 = 1$$

  where
  - $\boldsymbol{W}$ is a $K \times D$ matrix
  - $\boldsymbol{x}$ is a $D \times 1$ vector
  - $\boldsymbol{b}$ is a $K \times 1$ vector
  - $\boldsymbol{z}$ is a $K \times 1$ vector
- Non-vectorized form:

  $$z_k = \sum_{j=1}^{D} w_{kj} x_j + b_k, \quad k = 1, 2, \ldots, K$$

Generating a Prediction

- Interpret z_k as how much the model prefers the kth prediction

  $$y_i = \begin{cases} 1, & \text{if } i = \arg\max_k z_k \\ 0, & \text{elsewise} \end{cases}$$

- When $K = 2$, this would be same as binary linear classification

Softmax Regression

- Soften the predictions for optimization
- Activation function: **softmax function**, which is a generalization of the logistic function

  $$y_k = \text{softmax}(z_1, \ldots, z_K)_k = \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)}$$

- Inputs z_k are called the *logits*
- Interpret outputs as probabilities
- If z_k is much larger than the others, then $\text{softmax}(\boldsymbol{z})_k \approx 1$ and it behaves like argmax

Cross Entropy as Loss Function

$$\mathcal{L}_{\mathrm{CE}}(\boldsymbol{y}, \boldsymbol{t}) = -\sum_{k=1}^{K} t_k \log y_k = -\boldsymbol{t}^\top (\log \boldsymbol{y})$$

- The log is applied element-wise
- Can combine into a *softmax-cros-entropy* function

Gradient Descent Updates for Softmax Regression

- Softmax regression:

$$\begin{aligned}
\boldsymbol{z} &= \boldsymbol{W}\boldsymbol{x} \\
\boldsymbol{y} &= \mathrm{softmax}(\boldsymbol{z}) \\
\mathcal{L}_{\mathrm{CE}}(\boldsymbol{y}, \boldsymbol{t}) &= -\boldsymbol{t}^\top (\log \boldsymbol{y})
\end{aligned}$$

- Gradient descent updates:

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\mathrm{CE}}}{\partial \boldsymbol{w}_k} &= (y_k - t_k) \cdot \boldsymbol{x} \\
\boldsymbol{w}_k &\leftarrow \boldsymbol{w}_k - \frac{\alpha}{N} \sum_{i=1}^{N} \left(y_k^{(i)} - t_k^{(i)} \right) \boldsymbol{x}^{(i)}
\end{aligned}$$

10 Convexity

Critical Points

- Local optimum
- If a function is convex, then every critical point is a global optimum

Convex Sets

- A set $\mathcal{S}$ is **convex** if any line segment connecting two points in $\mathcal{S}$ lies entirely within $\mathcal{S}$

$$x_1, x_2 \in \mathcal{S} \implies \lambda x_1 + (1-\lambda)x_2 \in \mathcal{S} \quad \forall 0 \leq \lambda \leq 1$$

- Weighted averages or convex combinations of points in $\mathcal{S}$ lie within $\mathcal{S}$

$$x_1, \ldots, x_N \in \mathcal{S} \implies \lambda_1 x_1 + \cdots + \lambda_N x_N \in \mathcal{S} \quad \forall \lambda_i > 0, \lambda_1 + \cdots + \lambda_N = 1$$

Convex Functions

- A function f is **convex** if
  - the line segment between any two points on f's graph lies above f, or
  - the set of points lying above the graph of f is convex, or
  - f is bowl-shaped, or
  - for any x_0, x_1 in the domain of f,

$$f((1-\lambda)x_0 + \lambda x_1) \leq (1-\lambda)f(x_0) + \lambda f(x_1)$$

Convex Loss Functions

- For linear models, $z = \boldsymbol{w}^\top \boldsymbol{x} + b$ is a linear function of $\boldsymbol{w}$ and b
- If the loss function is a convex function of z, then it is also a convex function of $\boldsymbol{w}$ and b

11 Tracking Model Performance

Tracking Accuracy for Binary Classification

- **Accuracy**: fraction correctly classified
  - Equivalent to the average 0-1 loss, the **error rate**
  - Useful to track though we can't optimize it
- Can break down accuracy as below:

$$Acc = \frac{TP + TN}{P + N} = \frac{TP + TN}{TP + TN + FP + FN}$$

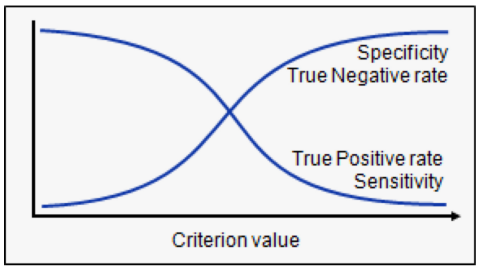
  where
  - P is the number of positives
  - N is the number of negatives
  - TP is the number of true positives
  - TN is the number of true negatives
  - FP is the number of false positives (type I error, i.e. predicted positive but actually negative)
  - FN is the number of false negatives (type II error, i.e. predicted negative but actually positive)
- Accuracy is highly sensitive to class imbalance
  - If only 0.1% of people have the disease, we can always predict negative and get 99.9% accuracy

Sensitivity and Specificity

- Useful under class imbalance

$$\text{Sensitivity} = \frac{TP}{TP + FN} = \frac{TP}{P} \quad \text{(True positive rate)}$$
$$\text{Specificity} = \frac{TN}{TN + FP} = \frac{TN}{N} \quad \text{(True negative rate)}$$

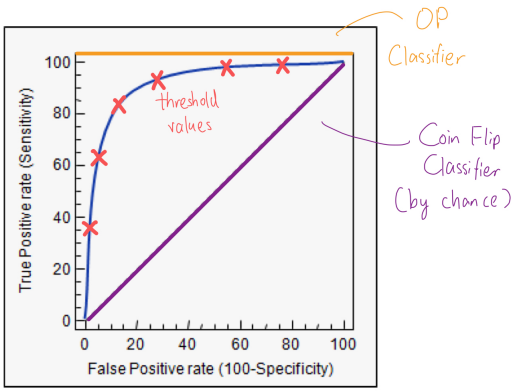
- Picking a threshold for $y = \sigma(x)$
  - A higher threshold increases specificity and decreases sensitivity
  - A lower threshold increases sensitivity and decreases specificity



Receiver Operating Characteristic (ROC) Curve

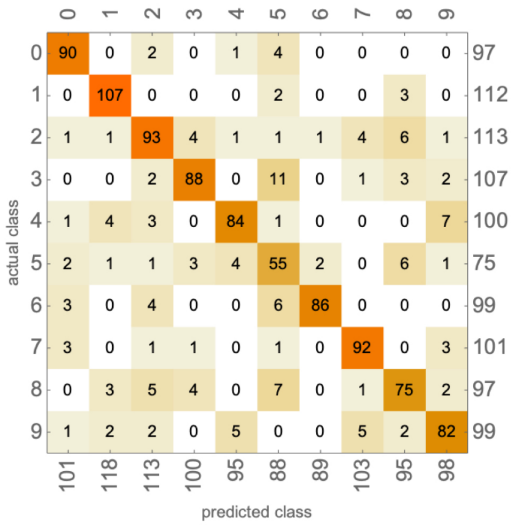
- Plots sensitivity against specificity
- Every point is a threshold value

- Area under the ROC curve (AUC) is a useful metric to track if a binary classifier achieves a good tradeoff between sensitivity and specificity



Confusion Matrix for Multi-Class Classification

- **Confusion matrix**: $K \times K$ matrix, where rows are true labels, columns are predicted labels, entries are frequencies



12 Neural Networks

Lemma 12.1. *XOR is not linearly separable*

Proof. Half-spaces are convex. If two points lie in a half-space, line segment connecting them also lie in the same half-space.

Suppose for a contradiction that there were some feasible weights (hypothesis). Then the points $(0,0)$ and $(1,1)$ would lie in the negative half-space and the points $(0,1)$ and $(1,0)$ would lie in the positive half-space. On one hand, the point $(0.5, 0.5)$ lies on the line segment connecting $(0,0)$ and $(1,1)$, therefore it belongs to the positive half-space. On the other hand, the point $(0.5, 0.5)$ lies on the line segment connecting $(0,1)$ and $(1,0)$, therefore it belongs to the negative half-space. This is a contradiction. □

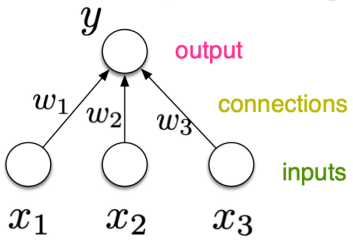
- However, we can use feature maps to classify XOR
  - E.g.

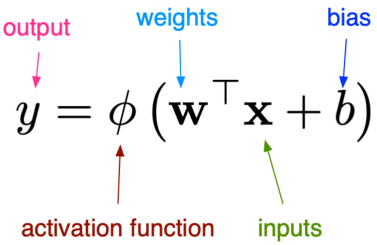
$$\boldsymbol{\psi}(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix}$$

- Designing feature maps can be hard

Neuron

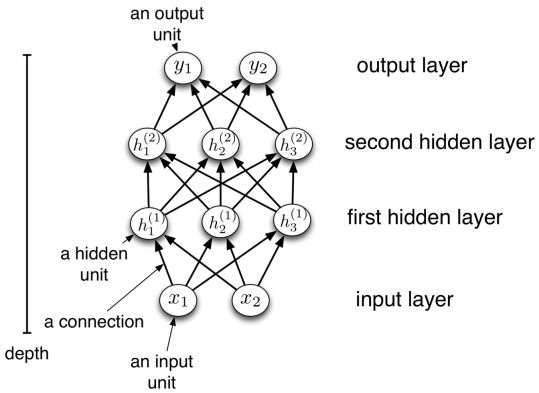
- For neural nets, we use a simple model of neuron, or **unit**





A Feed-Forward Neural Network

- A *directed acyclic graph* (DAG)
- Opposite: recurrent neural network (RNN)
- Units are grouped into *layers*



Multilayer Perceptrons

- A multi-layer network consists of fully connected layers
- In a *fully connected layer*, all input units are connected to all output units
- Each hidden layer i connects N_{i-1} input units to N_i output units; weight matrix is $N_i \times N_{i-1}$
- The outputs are a function of the input units

$$\boldsymbol{y} = f(\boldsymbol{x}) = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

  where ϕ is applied component-wise

Examples of Activation Functions

- Identity: $y = z$
- Rectified Linear Unit (ReLU): $y = \max(0, z)$
- Soft ReLU: $y = \log(1 + e^z)$
- Hard Threshold: $y = \mathbb{1}(z > 0)$
- Logistic: $y = \frac{1}{1+e^{-z}}$
- Hyperbolic Tangent: $y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

A Composition of Functions

- Each layer computes a function, so the network computes a composition of functions:

$$\begin{aligned}
\boldsymbol{h}^{(1)} &= f^{(1)}(\boldsymbol{x}) = \phi(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \\
\boldsymbol{h}^{(2)} &= f^{(2)}(\boldsymbol{h}^{(1)}) = \phi(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \\
&\vdots \\
\boldsymbol{y} &= f^{(L)}(\boldsymbol{h}^{(L-1)}) = \phi(\boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} + \boldsymbol{b}^{(L)})
\end{aligned}$$

- Equivalently

$$\boldsymbol{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\boldsymbol{x})$$

- Provides modularity, where we can implement each layer's computations as a black box
- For the last layer
  - If our task is regression, choose

$$\boldsymbol{y} = f^{(L)}\left(\boldsymbol{h}^{(L-1)}\right) = \left(\boldsymbol{w}^{(L)}\right)^\top \boldsymbol{h}^{(L-1)} + b^{(L)}$$

    * No need for activation function
  - If our task is classification, choose

$$\boldsymbol{y} = f^{(L)}\left(\boldsymbol{h}^{(L-1)}\right) = \sigma\left(\left(\boldsymbol{w}^{(L)}\right)^\top \boldsymbol{h}^{(L-1)} + b^{(L)}\right)$$

    * Need an activation

Feature Learning

- The hidden layers can be viewed as feature learning layers (feature extractors)

- Features learned by neural networks adapt to patterns in the data

Expressivity

- A hypothesis space $\mathcal{H}$ is the set of functions that can be represented by some model
- Consider 2 models A and B with hypothesis spaces $\mathcal{H}_A$, $\mathcal{H}_B$
- If $\mathcal{H}_B \subseteq \mathcal{H}_A$, then A is more *expressive* than B, i.e. A can reprsent any function f in $\mathcal{H}_B$

Expressive Power of Linear Networks

- Any sequence of *linear* layers is equivalent to a single linear layer
- Linear function composed with linear function is linear function
- Deep linear networks can only represent linear functions, which is no more regressive than linear regression

Expressive Power of Non-Linear Networks

- Multilayer feed-forward neural nets with *nonlinear* activation functions are **universal function approximators**
  - They can approximate any function arbitrarily well, i.e. for any $f : \mathcal{X} \to \mathcal{T}$, there is a sequence $f_i \in \mathcal{H}$ with $f_i \to f$ as $i \to \infty$
- For binary inputs and targets, universality can be shown by having 2^D hidden units, each of which corresopnds to one particular input configuration
  - Only requires 1 wide hidden layer
- Expressivity can be bad, where we could overfit to the training data

Regularization and Overfitting for Neural Networks

- Can apply L^2 regularization
- Can use **early stopping**, or stopping training early, because overfitting increases as training progresses

Learning Weights

- Goal: learn weights in a multi-layer neural network using gradient descent
- Weight space for a multi-layer neural net: one set of weights for each unit in every layer of the network

Univariate Chain Rule

- Let $z = f(y)$ and $y = g(x)$ be univariate functions. Then $z = f(g(x))$ implies

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Error Signal Notation

- Let $\overline{y}$ denote the derivative $\frac{d\mathcal{L}}{dy}$, called the **error signal**
- Error signals are *values* our program is computing
- E.g. for

$$z = wx + b \qquad y = \sigma(z) \qquad \mathcal{L} = \frac{1}{2}(y - t)^2$$

  we have

$$\overline{y} = y - t \qquad \overline{z} = \overline{y}\sigma'(z) \qquad \overline{w} = \overline{z}x \qquad \overline{b} = \overline{z}$$

Multivariate Chain Rule

- Suppose we have functions $f(x, y), x(t), y(t)$, then

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

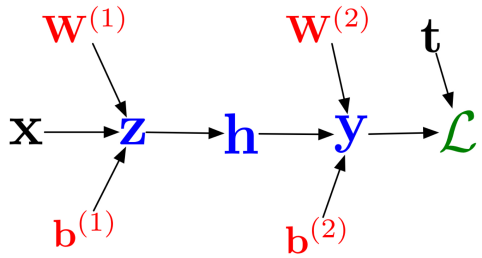
- Derivative along every path, then add the result

Full Backpropagation Algorithm

- Let $v_1, \ldots, v_N$ be a *topological ordering* of the computation graph (i.e. parents come before children), where v_i denotes the variable for which we're trying to compute gradients
- Forward pass: for $i = 1, \ldots, N$, compute v_i as a function of Parents(v_i) (i.e. from input, get output)
- Backward pass: for $i = N-1, \ldots, 1$, compute

$$\overline{v}_i = \sum_{j \in \text{Children}(v_i)} \overline{v}_j \frac{\partial v_j}{\partial v_i}$$

Backpropagation for Two-Layer Neural Network



- Forward pass:

$$\begin{aligned}
\boldsymbol{z} &= \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \\
\boldsymbol{h} &= \sigma(\boldsymbol{z}) \\
\boldsymbol{y} &= \boldsymbol{W}^{(2)}\boldsymbol{h} + \boldsymbol{b}^{(2)} \\
\mathcal{L} &= \frac{1}{2}\left\|\boldsymbol{t} - \boldsymbol{y}\right\|^2
\end{aligned}$$

- Backward pass:

$$\overline{\mathcal{L}} = 1 \qquad \overline{\boldsymbol{y}} = \overline{\mathcal{L}}(\boldsymbol{y} - \boldsymbol{t})$$

$$\overline{\boldsymbol{W}^{(2)}} = \overline{\boldsymbol{y}}\boldsymbol{h}^\top \qquad \overline{\boldsymbol{b}^{(2)}} = \overline{\boldsymbol{y}}$$

$$\overline{\boldsymbol{h}} = {\boldsymbol{W}^{(2)}}^\top \overline{\boldsymbol{y}} \qquad \overline{\boldsymbol{z}} = \overline{\boldsymbol{h}} \cdot \sigma'(\boldsymbol{z})$$

$$\overline{\boldsymbol{W}^{(1)}} = \overline{\boldsymbol{z}}\boldsymbol{x}^\top \qquad \overline{\boldsymbol{b}^{(1)}} = \overline{\boldsymbol{z}}$$

Computational Cost for Two-Layer Neural Network

- Forward pass: one add-multiply operation per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Backward pass: two add-multiply operations per weight

$$\overline{w_{ki}^{(2)}} = \overline{y_k} h_i$$
$$\overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)}$$

- 1 backward pass is as expensive as 2 forward passes
- For a multilayer perceptron, the cost is *linear* in the number of layers, and *quadratic* in the number of units per layer (since it is fully connected)

Auto-Differentiation

- **Autodifferentiation** performs backprop in a completely mechanical and automatic way
- Autodiff libraries: PyTorch, Tensorflow, Jax, etc.

13 Convolutional Networks

Motivation

- Suppose we want to train a network that takes a 200×200 RGB image as input
  - Input size $= 200 \times 200 \times 3 = 120000$
  - If we were to have 1000 hidden unit, then the number of parameters would be $120000 \times 1000 = 120000000$
  - If the object in the image shifts a little, then the network will have to relearn the weights

Convolution Layers

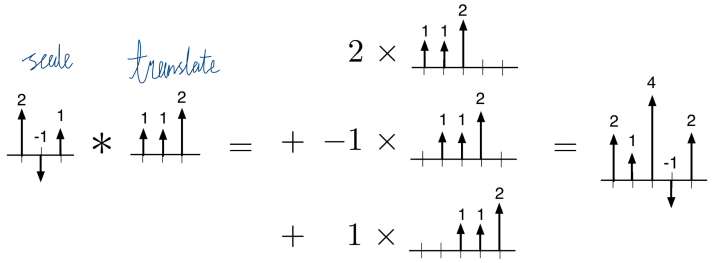
- Layers are *not* fully connected: each set of hidden units looks at a small region of the image
- Convolution layers can be stacked

1-D Convolution

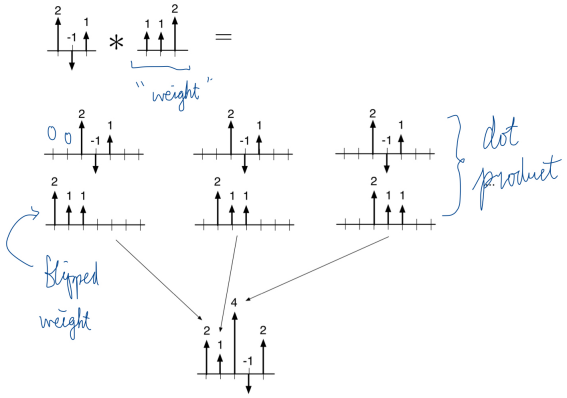
- We have two signals/arrays x and w
- x is an *input signal*
- w is a *set of k weights* (also referred to as a *kernel* or *filter*)
- Often *zero pad* x to an infinite array
- The tth value in the convolution is defined as

$$(x * w)[t] = \sum_{\tau=0}^{k-1} x[t-\tau]w[\tau]$$

- Can be thought of as *translate-and-scale*



- Can be thought of as *flip-and-filter*



Properties of Convolution

- Commutativity: $a * b = b * a$
- Linearity: $a * (\lambda_1 b + \lambda_2 c) = \lambda_1 a * b + \lambda_2 a * c$

2-D Convolution

- If x and w are 2-D arrays, then

$$(x * w)[i, j] = \sum_s \sum_t x[i - s, j - t] * w[s, t]$$

- Can be thought of as *translate-and-scale*

$$1 \times \begin{array}{|c|c|c|c|}\hline 1 & 3 & 1 & \\ \hline 0 & -1 & 1 & \\ \hline 2 & 2 & -1 & \\ \hline & & & \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array} = + 2 \times \begin{array}{|c|c|c|c|}\hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline & & & \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline 1 & 5 & 7 & 2 \\ \hline 0 & -2 & -4 & 1 \\ \hline 2 & 6 & 4 & -3 \\ \hline 0 & -2 & -2 & 1 \\ \hline \end{array}$$

$$+ -1 \times \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & 1 & 3 & 1 \\ \hline & 0 & -1 & 1 \\ \hline & 2 & 2 & -1 \\ \hline \end{array}$$

- Can be thought of as *flip-and-filter*

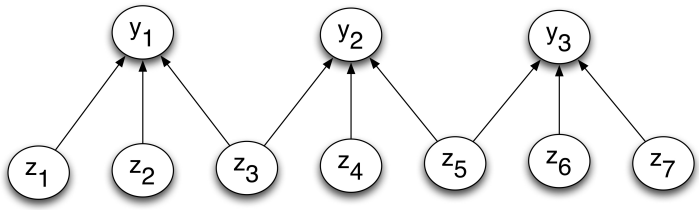
$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} * \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 0 & -1 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|}\hline 1 & 3 & 1 \\ \hline 0 & -1 & 1 \\ \hline 2 & 2 & -1 \\ \hline \end{array} \times \begin{array}{c} \text{-1 0} \\ \text{2 1} \end{array} \rightarrow \begin{array}{|c|c|c|c|}\hline 1 & 5 & 7 & 2 \\ \hline 0 & -2 & -4 & 1 \\ \hline 2 & 6 & 4 & -3 \\ \hline 0 & -2 & -2 & 1 \\ \hline \end{array}$$

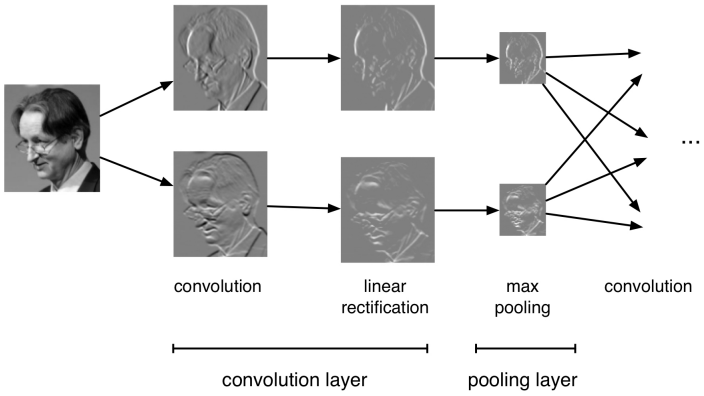
Layers

- Two types of layers: **convolution layers** (or detection layer), and **pooling layers**
- The convolution layer has a set off filters and produces a set of feature maps
- Each feature map is a result of convolving the image with a filter
- Common to apply a ReLU nonlinearity after each convolution layer
  - Since convolution is a linear operation, having 2 convolution layers would be no more powerful than 1

- The pooling layers reduce the size of the representation



- *Max-pooling* is commonly used, which computes the maximum value of the units in a pooling group
- Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in the lower layers
- Structure



Equivariance and Invariance

- Convolution layers are **equivariant**, i.e. if we translate the inputs, the outputs are translated by the same amount
- Want the network's predictions to be **invariant**, i.e. if we translate the inputs, the prediction should not change
- Pooling layers provide invariance to small translations

14 Probabilistic Models

Example of a Biased Coin

- We flip a coin $N = 100$ times and get outcomes $\{x_1, \ldots, x_N\}$, where $x_i \in \{0, 1\}$ and $x_i = 1$ if the coin lands heads H
- Suppose we had $N_H = 55$ heads and $N_T = 45$ tails
- Want to create a model to predict the outcome of the next coin flip
- The coin is likely biased, assume that one coin flip outcome x is a *Bernoulli random variable* for a currently unknown parameter $\theta \in [0, 1]$

$$\Pr(x = 1 \mid \theta) = \theta \quad \text{and} \quad \Pr(x = 0 \mid \theta) = 1 - \theta$$
$$\Pr(x \mid \theta) = \theta^x (1 - \theta)^{1-x}$$

- Assume that $\{x_1, \ldots, x_N\}$ are *independent and identically distributed* (i.i.d.)
- Thus, the joint probability of the outcome $\{x_1, \ldots, x_N\}$ is

$$\Pr(x_1, \ldots, x_N \mid \theta) = \prod_{i=1}^{N} \theta^{x_i} (1 - \theta)^{1-x_i}$$

  - θ is known
  - Can plug in a set of values for $x_1, \ldots, x_N$ to check probability

Maximum Likelihood Estimation

- The **likelihood function** is the probability of observing the data as a function of the parameters θ:

$$L(\theta) = \prod_{i=1}^{N} \theta^{x_i} (1 - \theta)^{1-x_i}$$

  - $x_1, \ldots, x_N$ are known
  - Can plug in a value for θ to check likelihood
- We usually work with log-likelihoods

$$l(\theta) = \sum_{i=1}^{N} [x_i \log \theta + (1 - x_i) \log(1 - \theta)]$$

- Good values of θ should assign high probability to the observed data
- The **maximum likelihood criterion** says that we should pick the parameter that maximize the likelihood, i.e.

$$\hat{\theta}_{ML} = \arg\max_{\theta \in [0,1]} l(\theta)$$

- Can find the optimal solution by setting derivatives to 0

$$\frac{dl}{d\theta} = \frac{d}{d\theta} \left(\sum_{i=1}^{N} x_i \log \theta + (1 - x_i) \log(1 - \theta) \right) = \frac{N_H}{\theta} - \frac{N_T}{1 - \theta}$$

- Therefore the maximum likelihood estimate is

$$\hat{\theta}_{ML} = \frac{N_H}{N_H + N_T}$$

Discriminative Classifiers

- **Discriminative** classifiers try to learn mappings directly from the space of inputs $\mathcal{X}$ to class labels $\{0, 1, \ldots, K\}$
- From features x, we get class probability $\Pr(y \mid x)$
- Estimate parameters of decision boundary/class separator directly from labelled exxamples
- Solves the problem of "how do I separate the classes?"

Generative Classfiers

- **Generative** classifiers try to build a model of "what data for a class looks like", i.e. $\Pr(\boldsymbol{x}, y)$
- If we know $\Pr(y)$, then we can compute $\Pr(\boldsymbol{x} \mid y)$
- Utilizes *Bayes' rule*, therefore also known as *Bayes classifiers*
- From probability of feature given label $\Pr(x \mid y)$, we get class label y
- Models the distribution of input characteristic of the class
- Models $\Pr(\boldsymbol{x} \mid t)$, then applies Bayes' Rule to derive $\Pr(t \mid \boldsymbol{x})$
- Solves the problem of "what does each class look like?"
- Requires a distributional assumption over inputs

Bayesian Classifier

- Given features $\boldsymbol{x} = (x_1, \ldots, x_D)$
- Compute class probabilities using Bayes' rule

$$\Pr(c \mid \boldsymbol{x}) = \frac{\Pr(\boldsymbol{x} \mid c) \Pr(c)}{\Pr(\boldsymbol{x})}$$

$$\text{Posterior for class} = \frac{\text{Probability of feature given class} \times \text{Prior for class}}{\text{Probability of feature}}$$

- We need $\Pr(\boldsymbol{x} \mid c)$ and $\Pr(c)$

Naive Bayes Independence Assumption

- We have two classes $c \in \{0, 1\}$
- We have binary features $\boldsymbol{x} = (x_1, \ldots, x_D)$ where $x_i \in \{0, 1\}$
- Define a joint distribution $\Pr(c, x_1, \ldots, x_D)$, we would need $2^{D+1} - 1$ probabilities to specify it
  - Two possibilities for each of c and x_i, and the final probability is 1 minus everyting else
- Naive assumption: the features x_i are *conditionally independent* given the class c
- We could decompose the joint distribution as follows:

$$\Pr(c, x_1, \ldots, x_D) = \Pr(c) \Pr(x_1 \mid c) \cdots \Pr(x_D \mid c)$$

- Prior probability of class: $\Pr(c = 1) = \pi$
- Conditional probability of feature given class: $\Pr(x_j = 1 \mid c) = \theta_{jc}$
- We now only need $2D + 1$ probabilities to specify this distribution
  - $\Pr(x_i = 1 \mid c = 1)$ and $\Pr(x_1 = 1 \mid c = 0)$ for each i, and $\Pr(c)$

Decomposing the Log-Likelihood

$$\begin{aligned}
l(\theta) &= \sum_{i=1}^{N} \log \Pr\left(c^{(i)}, \boldsymbol{x}^{(i)}\right) \\
&= \sum_{i=1}^{N} \log \left[\Pr\left(\boldsymbol{x}^{(i)} \mid c^{(i)}\right) \Pr\left(c^{(i)}\right)\right] \\
&= \sum_{i=1}^{N} \log \left[\Pr\left(c^{(i)}\right) \prod_{j=1}^{D} \Pr\left(x_j^{(i)} \mid c^{(i)}\right)\right] \qquad \text{By conditional independence} \\
&= \sum_{i=1}^{N} \left[\log \Pr\left(c^{(i)}\right) + \sum_{j=1}^{D} \log \Pr\left(x_j^{(i)} \mid c^{(i)}\right)\right] \\
&= \sum_{i=1}^{N} \log \Pr\left(c^{(i)}\right) + \sum_{j=1}^{D} \sum_{i=1}^{N} \log \Pr\left(x_j^{(i)} \mid c^{(i)}\right)
\end{aligned}$$

- The former term is the log-likelihood of labels
- The latter term is the sum of log-likelihoods of features

Learning the Prior Over Class

- To learn the prior, we maximize $\sum_{i=1}^{N} \log \Pr\left(c^{(i)}\right)$
- Define $\pi = \Pr\left(c^{(i)} = 1\right)$
- Probability of the ith data point: $\Pr\left(c^{(i)}\right) = \pi^{c^{(i)}} (1 - \pi)^{1 - c^{(i)}}$
- Log-likelihood of the dataset:

$$\sum_{i=1}^{N} \log \Pr\left(c^{(i)}\right) = \sum_{i=1}^{N} c^{(i)} \log \pi + \sum_{i=1}^{N} \left(1 - c^{(i)}\right) \log(1 - \pi)$$

- MLE for prior π is the fraction of positive examples in the dataset:

$$\hat{\pi} = \frac{\sum_i \mathbb{1}\left\{c^{(i)} = 1\right\}}{N}$$

Learning Probability of Feature Given Class

- To learn $\Pr\left(x_j^{(i)} = 1 \mid c\right)$, we maximize $\sum_{i=1}^{N} \log \Pr\left(x_j^{(i)} \mid c^{(i)}\right)$
- Define $\theta_{jc} = \Pr\left(x_j^{(i)} = 1 \mid c\right)$
- Probability of ith data point: $\Pr\left(x_j^{(i)} \mid c\right) = \theta_{jc}^{x_j^{(i)}} (1 - \theta_{jc})^{1 - x_j^{(i)}}$

- Log-likelihood of the dataset:

$$\sum_{i=1}^{N} \log \Pr\left(x_j^{(i)} \mid c^{(i)}\right)$$
$$= \sum_{i=1}^{N} c^{(i)} \left[x_j^{(i)} \log \theta_{j1} + \left(1 - x_j^{(i)}\right) \log\left(1 - \theta_{j1}\right)\right] + \sum_{i=1}^{N} \left(1 - c^{(i)}\right) \left[x_j^{(i)} \log \theta_{j0} + \left(1 - x_j^{(i)}\right) \log\left(1 - \theta_{j0}\right)\right]$$

- MLE for θ_{jc} is the fraction of feature j occurrences in each class in the dataset:

$$\hat{\theta}_{jc} = \frac{\sum_i \mathbb{1}\left\{x_j^{(i)} = 1 \wedge c^{(i)} = c\right\}}{\sum_i \mathbb{1}\left\{c^{(i)} = c\right\}}$$

Naive Bayes Properties

- Training time: estimate parameters using maximum likelihood
  - Compute co-occurrence counts of each feature with the labels
  - Requires only 1 pass through the data
- Test time: apply Bayes' rule
  - Cheap because of the model structure
- Analysis can be extended to probability distributions other than Bernoulli
- Less accurate in practice compared to discriminative models due to its "naive" independence assumption

Data Sparsity

- Maximum likelihood can *overfit* if there is too little data
- E.g. two coin tosses of both heads

The Posterior Distribution

- To define a Bayesian model, we need to specify two distributions:
  1. The **prior distribution** $\Pr(\boldsymbol{\theta})$ encodes our beliefs about the parameters *before* we observe the data
  2. The **likelihood** $\Pr(\mathcal{D} \mid \boldsymbol{\theta})$ encodes the likelihood of observing the data given the parameters
- When we *update* our beliefs based on the observations, we compute the **posterior distribution** using Bayes' rule:

$$\Pr(\boldsymbol{\theta} \mid \mathcal{D}) = \frac{\Pr(\boldsymbol{\theta}) \Pr(\mathcal{D} \mid \boldsymbol{\theta})}{\int \Pr(\boldsymbol{\theta}') \Pr(\mathcal{D} \mid \boldsymbol{\theta}') d\boldsymbol{\theta}'}$$

  - We rarely need to compute the denominator

Example of a Coin Flip

- The likelihood is

$$L(\theta) = \Pr(\mathcal{D} \mid \theta) = \theta^{N_H} (1 - \theta)^{N_T}$$

- We need to specify the prior $\Pr(\theta)$

- An *uninformative prior* assumes as little as possible, e.g. a uniform prior
- Experience tells us 0.5 is more likely than 0.99, therefore we could use the *beta distribution*

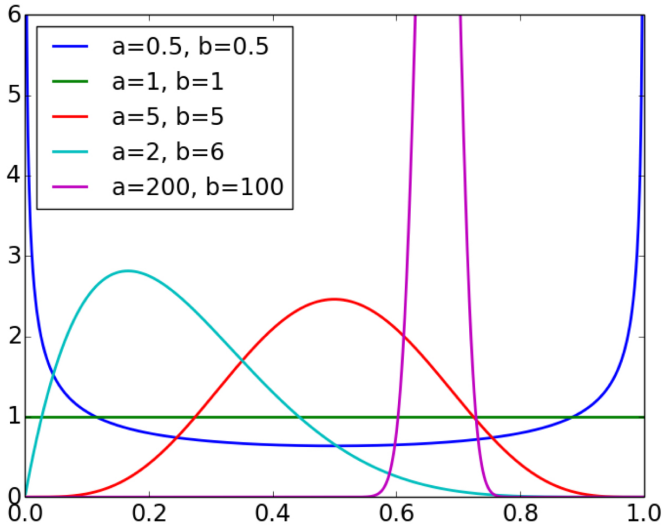
$$\Pr(\theta; a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\theta^{a-1}(1-\theta)^{b-1}$$

where we could ignore the normalization constant, and so

$$\Pr(\theta; a, b) \propto \theta^{a-1}(1-\theta)^{b-1}$$

Beta Distribution Properties

- Defined on $[0, 1]$
- Expectation is $\mathbb{E}[\theta] = a/(a+b)$
- Distribution gets more peaked when a and b are large
- When $a = b = 1$, it becomes the uniform distribution



Posterior for the Coin Flip Example

- Posterior distribution:

$$\begin{aligned}\Pr(\boldsymbol{\theta} \mid \mathcal{D}) &\propto \Pr(\boldsymbol{\theta})\Pr(\mathcal{D} \mid \boldsymbol{\theta}) \\ &\propto \left[\theta^{a-1}(1-\theta)^{b-1}\right]\left[\theta^{N_H}(1-\theta)^{N_T}\right] \\ &= \theta^{a-1+N_H}(1-\theta)^{b-1+N_T}\end{aligned}$$

- Therefore

$$\Pr(\boldsymbol{\theta} \mid \mathcal{D}) \sim \text{Beta}(N_H + a, N_T + b)$$

- The posterior expectation of θ is

$$\mathbb{E}[\theta \mid \mathcal{D}] = \frac{N_H + a}{N_H + N_T + a + b}$$

- a and b can be thought of as *pseudo-counts*, where we already have $a-1$ heads and $b-1$ tails
- The prior and likelihood have the same functional form, called *conjugate priors*

- When we have enough observations, the data could overwhelm the prior

Maximum A-Posteriori Estimation

$$\begin{aligned}\hat{\boldsymbol{\theta}}_{MAP} &= \arg\max_{\boldsymbol{\theta}} \Pr(\boldsymbol{\theta} \mid \mathcal{D}) \\ &= \arg\max_{\boldsymbol{\theta}} \Pr(\boldsymbol{\theta}) \Pr(\mathcal{D} \mid \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \left[\log \Pr(\boldsymbol{\theta}) + \log \Pr(\mathcal{D} \mid \boldsymbol{\theta})\right]\end{aligned}$$

- If the prior is uniform, then $\Pr(\boldsymbol{\theta})$ is a constant, therefore MAP is the same as maximum likelihood
- Converts the Bayesian parameter estimation problem into a maximization problem
- For the coin flip example:

$$\hat{\theta}_{MAP} = \frac{N_H + a - 1}{N_H + N_T + a + b - 2}$$

  - Can be understood as $a - 1 + N_H$ heads and $b - 1 + N_T$ tails

15 Multivariate Gaussians

Eigenvectors and Eigenvalues

- Let $\boldsymbol{B}$ be a square matrix
- An **eigenvector** of $\boldsymbol{B}$ is a vector $\boldsymbol{v}$ such that

$$\boldsymbol{B}\boldsymbol{v} = \lambda \boldsymbol{v}$$

  for a scalar λ, which is called an **eigenvalue**
- A matrix of size $D \times D$ has *at most D distinct eigenvalues*

Spectral Theorem

- For a symmetric $D \times D$ matrix:
  - All of the eigenvalues are *real-valued*
  - There is a full set of D linearly independent eigenvectors, and these eigenvectors form a basis for $\mathbb{R}^D$
  - These eigenvectors can be chosen to be *real-values*
  - These eigenvectors can be chosen to be *orthogonal*, i.e. perpendicular unit vectors

Spectral Decomposition

- Factorize a symmetric matrix $\boldsymbol{A}$ with the **Spectral decomposition**:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$$

  - $\boldsymbol{Q}$ is an orthogonal matrix, and the columns $\boldsymbol{q}_i$ of $\boldsymbol{Q}$ are eigenvectors
  - $\boldsymbol{\Lambda}$ is a diagonal matrix, and the diagonal entries λ_i are the corresponding eigenvalues
- Because $\boldsymbol{A}$ has a full set of orthonormal eigenvectors $\{\boldsymbol{q}_i\}$, we can use these as an orthonormal basis for $\boldsymbol{R}^D$
  - A vector $\boldsymbol{x}$ can be written in an alternate coordinate system:

$$\boldsymbol{x} = \tilde{x}_1 \boldsymbol{q}_1 + \cdots + \tilde{x}_D \boldsymbol{q}_D$$

  - Converting between the two coordinate systems:

$$\tilde{\boldsymbol{x}} = \boldsymbol{Q}^\top \boldsymbol{x} \qquad \boldsymbol{x} = \boldsymbol{Q}\tilde{\boldsymbol{x}}$$

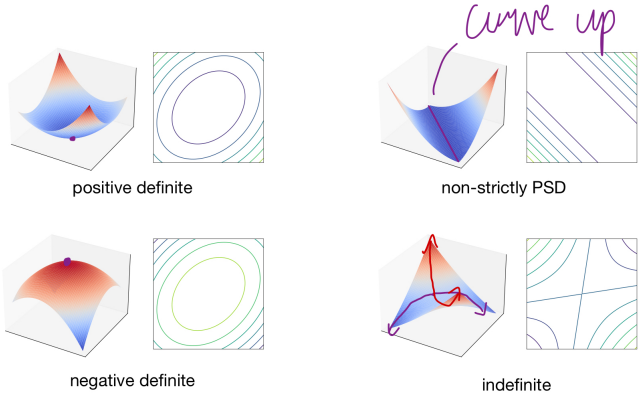
    * For an orthonormal matrix, its inverse is its transpose
- In the alternate coordinate system, $\boldsymbol{A}$ acts by rescaling the individual coordinates:

$$\boldsymbol{A}\boldsymbol{x} = \tilde{x}_1 \boldsymbol{A}\boldsymbol{q}_1 + \cdots + \tilde{x}_D \boldsymbol{A}\boldsymbol{q}_D = \lambda_1 \tilde{x}_1 \boldsymbol{q}_1 + \cdots + \lambda_D \tilde{x}_D \boldsymbol{q}_D$$

Quadratic Forms

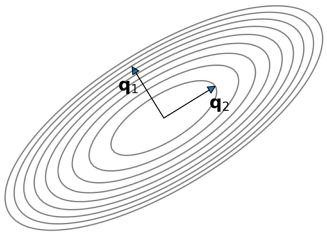
- Symmetric matrices represent **quadratic forms**, $f(\boldsymbol{v}) = \boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v}$ (terms are quadratic)
- If $\boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v} > 0$ for all $\boldsymbol{v} \neq \boldsymbol{0}$, then $\boldsymbol{A}$ is **positive definite**, denoted $\boldsymbol{A} \succ 0$
- If $\boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v} \geq 0$ for all $\boldsymbol{v}$, then $\boldsymbol{A}$ is **positive semi-definite**, denoted $\boldsymbol{A} \succeq 0$
- If $\boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v} < 0$ for all $\boldsymbol{v} \neq \boldsymbol{0}$, then $\boldsymbol{A}$ is **negative definite**, denoted $\boldsymbol{A} \prec 0$

- If $\boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v}$ can be positive or negative, then $\boldsymbol{A}$ is **indefinite**

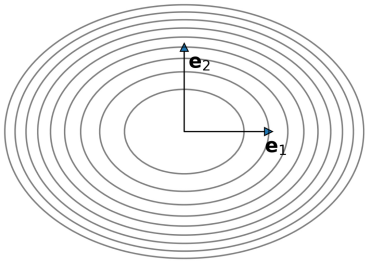


PSD Matrices

- Non-negative linear combinations of PSD matrices are PSD
- If $\boldsymbol{A}$ is a random matrix which is always PSD, then $\mathbb{E}[\boldsymbol{A}]$ is PSD
- For any matrix $\boldsymbol{B}$, the matrix $\boldsymbol{B}\boldsymbol{B}^\top$ is PSD
- For a random vector $\boldsymbol{x}$, the *covariance matrix* $\text{Cov}(\boldsymbol{x}) = \mathbb{E}[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^\top]$ is a PSD matrix
- $\boldsymbol{A}$ is positive definite (resp. PSD) iff all of its eigenvalues are positive (resp. non-negative)
- For general positive definite $\boldsymbol{A}$, the contours of the quadratic form are elliptical, and the principle axes of the ellipses are aligned with the eigenvectors



- If $\boldsymbol{A}$ is both diagonal and positive definite (i.e. its diagonal entries are positive), then the ellipses are axis-aligned



Matrix Powers

- Applying the Spectral Decomposition, we can take the kth power of the matrix $\boldsymbol{A}$:

$$\boldsymbol{A}^k = \boldsymbol{Q}\boldsymbol{\Lambda}^k\boldsymbol{Q}^\top$$

- If $\boldsymbol{A}$ is invertible, we can take its inverse:

$$\boldsymbol{A}^{-1} = \left(\boldsymbol{Q}^\top\right)^{-1} \boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^{-1} = \boldsymbol{Q}\boldsymbol{\Lambda}^{-1}\boldsymbol{Q}^\top$$

- If $\boldsymbol{A}$ is PSD, then we can define the **matrix square root**:

$$\boldsymbol{A}^{1/2} = \boldsymbol{Q}\boldsymbol{\Lambda}^{1/2}\boldsymbol{Q}^\top$$

  - The resulting matrix is PSD and squaring it gives $\boldsymbol{A}$

Determinant Properties

- The determinant of a symmetric matrix equals the product of its eigenvalues

$$|\boldsymbol{A}| = |\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top| = |\boldsymbol{Q}||\boldsymbol{\Lambda}||\boldsymbol{Q}^\top| = |\boldsymbol{\Lambda}| = \sum_i \lambda_i$$

- The determinant of a PSD matrix is non-negative, and the determinant of a positive definite matrix is positive
- Properties of determinant:
  1. $|\boldsymbol{B}\boldsymbol{C}| = |\boldsymbol{B}| \cdot |\boldsymbol{C}|$
  2. $|\boldsymbol{B}| = 0$ iff $\boldsymbol{B}$ is singular (i.e. not invertible)
  3. $\left|\boldsymbol{B}^{-1}\right| = |\boldsymbol{B}|^{-1}$ if $\boldsymbol{B}$ is invertible (nonsingular)
  4. $|\boldsymbol{B}^\top| = |\boldsymbol{B}|$
  5. If $\boldsymbol{Q}$ is orthogonal, then $|\boldsymbol{Q}| = \pm 1$ (since orthogonal transformations preserve volume)
  6. If $\boldsymbol{\Lambda}$ is diagonal with entries $\{\lambda_i\}$, then $|\Lambda| = \prod_i \lambda_i$

Univariate Gaussian Distribution

- The **Gaussian** or **normal** distribution:

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

- Parameterized by mean μ and variance σ^2
- The Central Limit Theorem says that sums of lots of independent random variables are approximately Gaussian
- Gaussians make calculations easy in machine learning

Multivariate Mean and Covariance

$$\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{x}] = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_D \end{bmatrix}$$

$$\boldsymbol{\Sigma} = \mathrm{Cov}(\boldsymbol{x}) = \mathbb{E}[(\boldsymbol{x}-\boldsymbol{\mu})(\boldsymbol{x}-\boldsymbol{\mu})^\top] = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1D} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2D} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{D1} & \sigma_{D2} & \cdots & \sigma_D^2 \end{bmatrix}$$

- $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ uniquely define a **multivariate Gaussian** or **normal** distribution, denoted $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ or $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$
- The covariance tells us how different dimensions interact

Multivariate Gaussian Distribution

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right]$$

- For univariate Gaussian distributions, we can obtain $\mathcal{N}(\mu, \sigma^2)$ by starting with $\mathcal{N}(0, 1)$, shifting by μ, and stretching by $\sigma = \sqrt{\sigma^2}$
- The standard multivariate normal has $\boldsymbol{\mu} = \mathbf{0}$ and $\boldsymbol{\Sigma} = \boldsymbol{I}$
- Shifting is by vector $\boldsymbol{\mu}$
- Scaling is by the matrix square root $\boldsymbol{\Sigma}^{1/2}$
  - $\boldsymbol{\Sigma}^{1/2} = \boldsymbol{Q}\boldsymbol{\Lambda}^{1/2}\boldsymbol{Q}^\top$
  - For each eigenvector $\boldsymbol{q}_i$ with eigenvalue λ_i, we stretch by a factor of $\sqrt{\lambda_i}$ in the direction $\boldsymbol{q}_i$

Examples of Bivariate Gaussian

$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ $\boldsymbol{\Sigma} = 0.5\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ $\boldsymbol{\Sigma} = 2\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

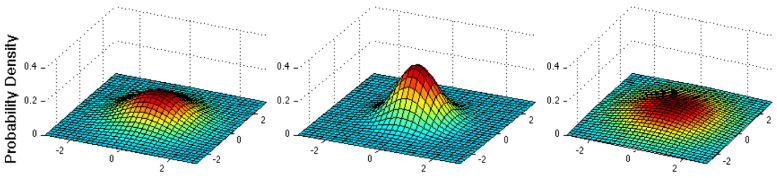


Figure: Probability density function

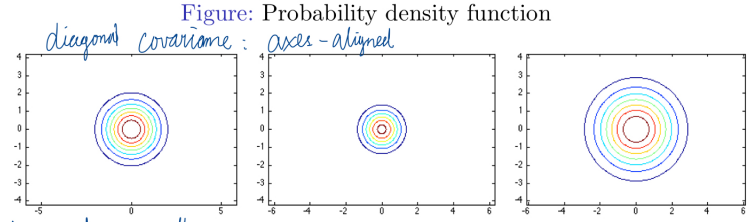


Figure: Contour plot of the pdf

$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ $\boldsymbol{\Sigma} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ $\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$

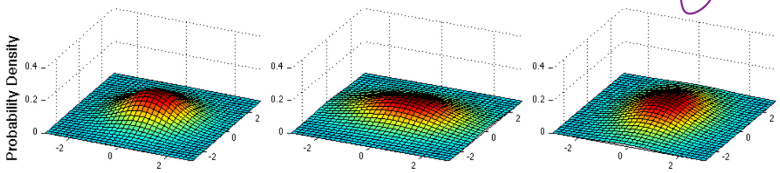


Figure: Probability density function

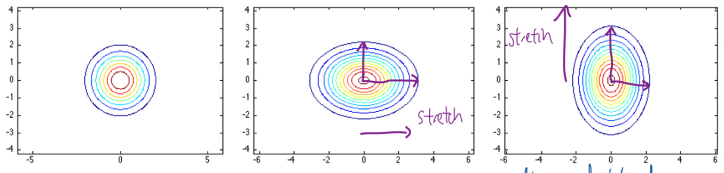


Figure: Contour plot of the pdf things don't change equally

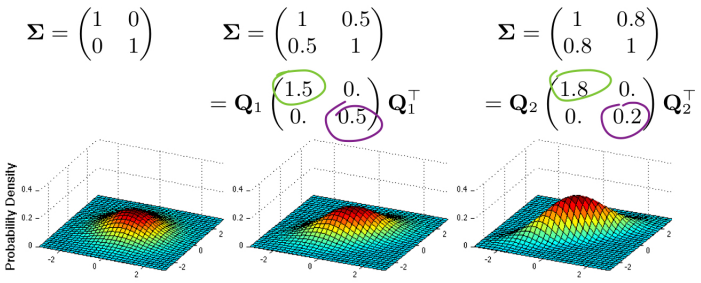


Figure: Probability density function

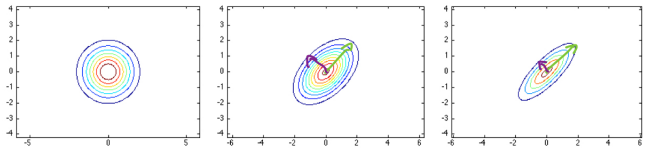


Figure: Contour plot of the pdf

Maximum Likelihood for Multivariate Gaussian

- Assuming data are drawn from a Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, we want to estimate $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ using data
- For univariate Gaussian:

$$\hat{\mu}_{ML} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{x}^{(i)}$$

$$\hat{\sigma}_{ML} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{x}^{(i)} - \mu\right)^2}$$

- Log-likelihood for multivariate Gaussian:

$$l(\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^{N} -\log(2\pi)^{d/2} - \log|\boldsymbol{\Sigma}|^{1/2} - \frac{1}{2} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}\right)^\top \boldsymbol{\Sigma}^{-1} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}\right)$$

- MLE for $\boldsymbol{\mu}$:

$$\hat{\boldsymbol{\mu}} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{x}^{(i)}$$

  which is the *empirical mean*

- MLE for $\boldsymbol{\Sigma}$:

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N} (\boldsymbol{X} - \mathbf{1}\boldsymbol{\mu}^\top)^\top (\boldsymbol{X} - \mathbf{1}\boldsymbol{\mu}^\top)$$

  which is the *empirical covariance*

  - $\mathbf{1}$ is an N-dimensional vector of 1s

Linear Regression as Maximum Likelihood

- Assume a Gaussian noise model

$$t \mid \boldsymbol{x} \sim \mathcal{N}(\boldsymbol{w}^\top \boldsymbol{x}, \sigma^2)$$

- Linear regression is maximum likelihood under this model:

$$\frac{1}{N}\sum_{i=1}^{N}\log\Pr\left(t^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{w}, b\right) = \text{const} - \frac{1}{2N\sigma^2}\sum_{i=1}^{N}\left(t^{(i)} - \boldsymbol{w}^\top \boldsymbol{x}\right)^2$$

Regularization as MAP Inference

- We can view an L_2 regularizer as MAP inference with a Gaussian prior
- MAP inference:

$$\arg\max_{\boldsymbol{w}} \log\Pr(\boldsymbol{w} \mid \mathcal{D}) = \arg\max_{\boldsymbol{w}} \left[\log\Pr(\boldsymbol{w}) + \log\Pr(\mathcal{D} \mid \boldsymbol{w})\right]$$

- Assuming a Gaussian prior

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{m}, \boldsymbol{S})$$

  we have

$$\log\Pr(\boldsymbol{w}) = -\frac{1}{2}(\boldsymbol{w} - \boldsymbol{m})^\top \boldsymbol{S}^{-1}(\boldsymbol{w} - \boldsymbol{m}) + \text{const}$$

- Having $\boldsymbol{m} = \boldsymbol{0}$ and $\boldsymbol{S} = \eta\boldsymbol{I}$, we have

$$\log\Pr(\boldsymbol{w}) = -\frac{1}{2\eta}\left\|\boldsymbol{w}\right\|^2 + \text{const}$$

  which is L_2 regularization

Gaussian Discriminant Analysis

- **Gaussian Discriminant Analysis (GDA)** assumes that $\Pr(\boldsymbol{x} \mid t)$ is distributed according to a multivariate Gaussian distribution

$$\Pr(\boldsymbol{x} \mid t = k) = \frac{1}{(2\pi)^{D/2}\left|\boldsymbol{\Sigma}_k\right|^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_k)\right]$$

- Each class k has associated mean vector $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$
  - Each $\boldsymbol{\mu}_k$ has D parameters, for a total of DK parameters
  - Each $\boldsymbol{\Sigma}_k$ has $\mathcal{O}(D^2)$ parameters, for a total of $\mathcal{O}(D^2K)$ parameters
- GDA learns the parameters for each class using maximum likelihood
- E.g. binary classification

$$\Pr(t \mid \phi) = \phi^t(1-\phi)^{1-t}$$

  - Can compute the ML estimates in closed form

$$\hat{\phi} = \frac{1}{N}\sum_{i=1}^{N} r_1^{(i)} \qquad \text{Fraction of examples in class 1}$$

$$\hat{\boldsymbol{\mu}}_k = \frac{\sum_{i=1}^{N} r_k^{(i)} \cdot \boldsymbol{x}^{(i)}}{\sum_{i=1}^{N} r_k^{(i)}} \qquad \text{Empirical mean}$$

$$\hat{\boldsymbol{\Sigma}}_k = \frac{1}{\sum_{i=1}^{N} r_k^{(i)}}\sum_{i=1}^{N} r_k^{(i)}\left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_k\right)\left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_k\right)^\top \qquad \text{Empirical covariance}$$

$$\hat{r}_k^{(i)} = \mathbb{1}\left[t^{(i)} = k\right] \qquad \text{Does example } i \text{ belong to class } k?$$

- ϕ is the probability for an example to belong in class 1
- $\boldsymbol{\mu}_k$ is the mean vector for class k
- $\boldsymbol{\Sigma}_k$ is the covariance matrix for class k

GDA Decision Boundary

- For Bayes Classifiers, we compute the decision boundary with Bayes' Rule:

$$\Pr(t \mid \boldsymbol{x}) = \frac{\Pr(t)\Pr(\boldsymbol{x} \mid t)}{\sum_{t'} \Pr(t')\Pr(\boldsymbol{x} \mid t')}$$

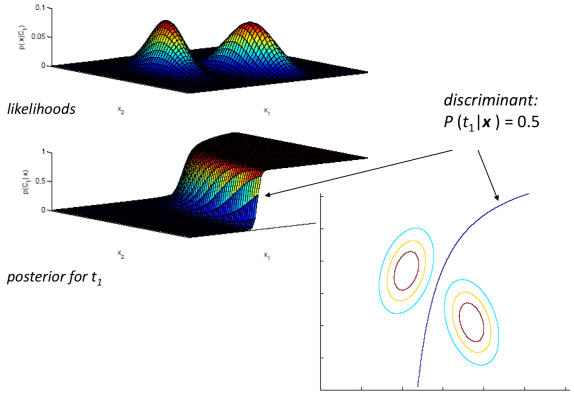
- By plugging in the Gaussian for $\Pr(\boldsymbol{x} \mid t)$:

$$\begin{aligned}\log \Pr(t_k \mid \boldsymbol{x}) &= \log \Pr(\boldsymbol{x} \mid t_k) + \log \Pr(t_k) - \log \Pr(\boldsymbol{x}) \\ &= -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k) + \log \Pr(t_k) - \log \Pr(\boldsymbol{x})\end{aligned}$$

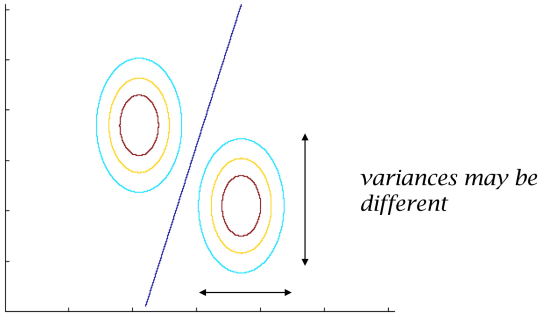
- The decision boundary, by setting $\log \Pr(t_k \mid \boldsymbol{x}) = \log \Pr(t_l \mid \boldsymbol{x})$, is

$$(\boldsymbol{x}-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k) = (\boldsymbol{x}-\boldsymbol{\mu}_l)^\top \boldsymbol{\Sigma}_l^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_l) + \text{const}$$

- Since we have a quadratic function in $\boldsymbol{x}$, the decision boundary is a conic section



- If all classes share the same covariance $\boldsymbol{\Sigma}$, then we get a *linear* decision boundary



GDA vs. Logistic Regression

- Binary classification: if we examine $\Pr(t = 1 \mid \boldsymbol{x})$ under GDA and assume $\boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}$, then we get

$$\Pr(t \mid \boldsymbol{x}, \phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) = \frac{1}{1 + \exp\left(-\boldsymbol{w}^\top \boldsymbol{x} - b\right)}$$

  where $(\boldsymbol{w}, b)$ are chosen based on $\phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}$

- This is the same model as *logistic regression*
- Decision boundary is linear
- GDA makes a stronger modelling assumption, which assumes that class-conditional data is multivariate Gaussian
  - If this is true, GDA is asymptotically effficient
  - If this is not true, the quality of the predictions might suffer
- Many class-conditional distribution distributions lead to logistic classifier
  - When these distributions are non-Gaussian, logistic regression usually beats GDA
- GDA can easily handle missing features (which is not the case for logistic regression)

Gaussian Naive Bayes

- If $\boldsymbol{x}$ is high-dimensional, then D would be large and that $\boldsymbol{\Sigma}_k$ would have $\mathcal{O}(D^2K)$ parameters
- **Naive Bayes**: assumption that features aree independent given the class

$$\Pr(\boldsymbol{x} \mid t = k) = \prod_{j=1}^{D} \Pr(x_j \mid t = k)$$

- Assuming likelihoods are Gaussian, then the Naive Bayes classifier requires D parameters (since $\boldsymbol{\Sigma}$ is diagonal)
- **Gaussian Naive Bayes** classifier assumes that the likelihoods are Gaussian:

$$\Pr(x_j \mid t = k) = \frac{1}{\sqrt{2\pi}\sigma_{jk}} \exp\left[-\frac{(x_j - \mu_{jk})^2}{2\sigma_{jk}^2}\right]$$

  (1-dimensional Gaussian, one for each input dimension)
- Models the same as GDA with diagonal covariance matrix
- Maximum likelihood estimate of parameters:

$$\mu_{jk} = \frac{\sum_{i=1}^{N} r_k^{(i)} x_j^{(i)}}{\sum_{i=1}^{N} r_k^{(i)}}$$

$$\sigma_{jk}^2 = \frac{\sum_{i=1}^{N} r_k^{(i)} \left(x_j^{(i)} - \mu_{jk}\right)^2}{\sum_{i=1}^{N} r_k^{(i)}}$$

$$r_k^{(i)} = \mathbb{1}\left[t^{(i)} = k\right]$$

Isotropic Decision Boundary

- We could assume that the covariances are **spherical**, or **isotropic** (i.e. all variances are the same)
- In this case $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$ (only 1 parameter)

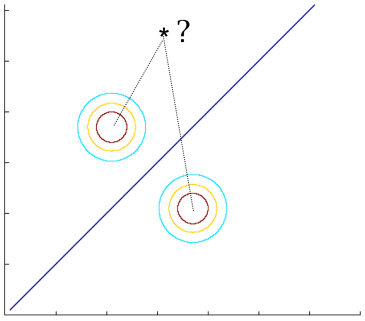
- Class posterior for GDA:

$$\begin{aligned}\log \Pr(t_k \mid \boldsymbol{x}) &= \log \Pr(\boldsymbol{x} \mid t_k) + \log \Pr(t_k) - \log \Pr(\boldsymbol{x}) \\ &= -\frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}_k| - \frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k) + \log \Pr(t_k) - \log \Pr(\boldsymbol{x})\end{aligned}$$

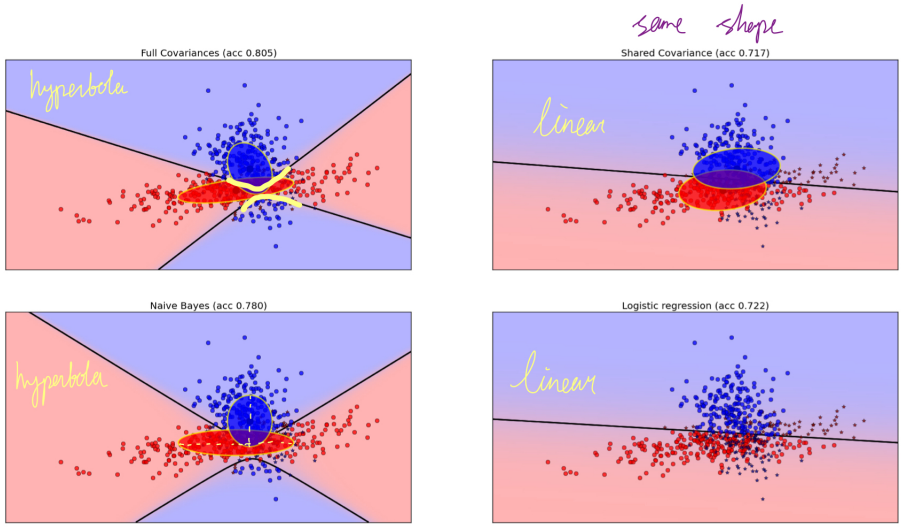
- Plugging in $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$:

$$\begin{aligned}\log \Pr(t_k \mid \boldsymbol{x}) - \log \Pr(t_l \mid \boldsymbol{x}) &= -\frac{1}{2\sigma^2}\left[(\boldsymbol{x}-\boldsymbol{\mu}_k)^\top(\boldsymbol{x}-\boldsymbol{\mu}_k) - (\boldsymbol{x}-\boldsymbol{\mu}_l)^\top(\boldsymbol{x}-\boldsymbol{\mu}_l)\right] \\ &= -\frac{1}{2\sigma^2}\left[\|\boldsymbol{x}-\boldsymbol{\mu}_k\|^2 - \|\boldsymbol{x}-\boldsymbol{\mu}_l\|^2\right]\end{aligned}$$

- The decision boundary bisects the class means



Decision Boundaries



16 Principal Component Analysis

Overview

- Principal Component Analysis (PCA) is an *unsupervised learning algorithm*, and an example of *linear dimensionality reduction*
- **Dimensionality reduction**: maps data to a lower dimensional space
  - Save computation/memory
  - Reduce overfitting, achieve better generalization
  - Visualize in 2 dimensions
- Since PCA is a linear model, this mapping will be a *projection*

Euclidean Projection

- Projection of $\boldsymbol{x}$ on $\mathcal{S}$ denoted by $\text{Proj}_{\mathcal{S}}(\boldsymbol{x})$
- $\boldsymbol{x}^\top \boldsymbol{u} = \|\boldsymbol{x}\| \, \|\boldsymbol{u}\| \cos(\theta) = \|\boldsymbol{x}\| \cos(\theta) = \|\tilde{\boldsymbol{x}}\|$
- $\text{Proj}_{\mathcal{S}}(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{u} \cdot \boldsymbol{u} = \|\tilde{\boldsymbol{x}}\| \, \boldsymbol{u}$
  - $\boldsymbol{x}^\top \boldsymbol{u}$ is the length of the projection
  - $\boldsymbol{u}$ is the direction of projection

Projection onto a Subspace

- To project onto a K-dimensional subspace, we first choose an orthonormal basis $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_K\}$ for $\mathcal{S}$, then project onto each unit vector individually and sum together the projections

$$\text{Proj}_{\mathcal{S}}(\boldsymbol{x}) = \sum_{i=1}^{K} z_i \boldsymbol{u}_i \qquad \text{where } z_i = \boldsymbol{x}^\top \boldsymbol{u}_i$$
$$\text{Proj}_{\mathcal{S}}(\boldsymbol{x}) = \boldsymbol{U}\boldsymbol{z} \qquad \text{where } \boldsymbol{z} = \boldsymbol{U}^\top \boldsymbol{x}$$

- We want to be able to project onto *affine spaces*, which could have arbitrary origin $\hat{\boldsymbol{\mu}}$, so that $\boldsymbol{z} = \boldsymbol{U}^\top (\boldsymbol{x} - \hat{\boldsymbol{\mu}})$
- In machine learning, $\tilde{\boldsymbol{x}}$ is called the **reconstruction** of $\boldsymbol{x}$
- $\boldsymbol{z}$ is called the **representation**, or **code**
- If we have a K-dimensional subspace in a D-dimensional input space, then $\boldsymbol{x} \in \mathbb{R}^D$ and $\boldsymbol{z} \in \mathbb{R}^K$
- If the data points $\boldsymbol{x}$ all lie close to their reconstructions, then we can approximate distances in terms of these same operations on the code vectors $\boldsymbol{z}$
- If $K \ll D$, then it's much cheaper to work with $\boldsymbol{z}$ than $\boldsymbol{x}$
- A mapping to a space that's easier to manipulate or visualize is called a **representation**, and learning such a mapping is **representation learning**
- Dimensionality reduction

Leraning a Subspace

- Choosing a good subspace $\mathcal{S}$:
  - Origin $\hat{\boldsymbol{\mu}}$ is the *empirical mean* of the data

- Need to choose a $D \times K$ matrix $\boldsymbol{U}$ with *orthonormal columns*

- Two criteria

  1. Minimize the *reconstruction error* (i.e. average squared error)

$$\min_{\boldsymbol{U}} \frac{1}{N} \sum_{i=1}^{N} \left\| \boldsymbol{x}^{(i)} - \tilde{\boldsymbol{x}}^{(i)} \right\|^2$$

  2. Maximize the *variance of reconstructions* (i.e. find a subspace where data has the *most variability*)

$$\max_{\boldsymbol{U}} \frac{1}{N} \sum_{i=1}^{N} \left\| \tilde{\boldsymbol{x}}^{(i)} - \hat{\boldsymbol{\mu}} \right\|^2$$

- The two criteria are equivalent

Principal Component Analysis

- Choosing a subspace to maximize the projected variance (minimize the reconstruction error) is called **principal component analysis (PCA)**

- Consider the *empirical covariance matrix*

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{\mu}} \right) \left(\boldsymbol{x}^{(i)} - \hat{\boldsymbol{\mu}} \right)^{\top}$$

- $\hat{\boldsymbol{\Sigma}}$ is symmetric and positive semidefinite

- The *optimal PCA subspace* is spanned by the *top K eigenvectors* (i.e. those with the highest eigenvalues) of $\hat{\boldsymbol{\Sigma}}$

- Choose the first K of any orthonormal eigenbasis for $\hat{\boldsymbol{\Sigma}}$

- These eigenvectors are called *principal components*

Decorrelation

- The dimensions of $\boldsymbol{z}$ are decorrelated

- The covariance matrix, $\text{Cov}(\boldsymbol{z})$ is diagonal (i.e. the top left $K \times K$ block of $\boldsymbol{\Lambda}$), meaning that the features are uncorrelated

Matrix Factorization

- For PCA, each input vector $\boldsymbol{x}^{(i)} \in \mathbb{R}^D$ is approximated as $\hat{\boldsymbol{\mu}} + \boldsymbol{U}\boldsymbol{z}^{(i)}$:

$$\boldsymbol{x}^{(i)} \approx \tilde{\boldsymbol{x}}^{(i)} = \hat{\boldsymbol{\mu}} + \boldsymbol{U}\boldsymbol{z}^{(i)}$$

  where

  - $\hat{\boldsymbol{\mu}} = \frac{1}{n} \sum_i \boldsymbol{x}^{(i)}$ is the data mean
  - $\boldsymbol{U} \in \mathbb{R}^{D \times K}$ is the orthogonal basis for the principal subspace
  - $\boldsymbol{z}^{(i)} \in \mathbb{R}^K$ is the code vector
  - $\tilde{\boldsymbol{x}}^{(i)} \in \mathbb{R}^D$ is $\boldsymbol{x}^{(i)}$'s reconstruction or approximation

- Assume for simplicity that the data is centered, i.e. $\hat{\boldsymbol{\mu}} = \mathbf{0}$. Then the approximation looks like

$$\boldsymbol{x}^{(i)} \approx \tilde{\boldsymbol{x}}^{(i)} = \boldsymbol{U}\boldsymbol{z}^{(i)}$$

- In matrix form, we have $\boldsymbol{X} \approx \boldsymbol{Z}\boldsymbol{U}^\top$ where $\boldsymbol{X}$ and $\boldsymbol{Z}$ are matrices with one *row* per data point

$$\boldsymbol{X} = \begin{bmatrix} \left[\boldsymbol{x}^{(1)}\right]^\top \\ \left[\boldsymbol{x}^{(2)}\right]^\top \\ \vdots \\ \left[\boldsymbol{x}^{(N)}\right]^\top \end{bmatrix} \in \mathbb{R}^{N \times D} \quad \text{and} \quad \boldsymbol{Z} = \begin{bmatrix} \left[\boldsymbol{z}^{(1)}\right]^\top \\ \left[\boldsymbol{z}^{(2)}\right]^\top \\ \vdots \\ \left[\boldsymbol{z}^{(N)}\right]^\top \end{bmatrix} \in \mathbb{R}^{N \times K}$$

- Can write the squared reconstruction error as

$$\sum_{i=1}^{N} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{U}\boldsymbol{z}^{(i)} \right\|^2 = \left\| \boldsymbol{X} - \boldsymbol{Z}\boldsymbol{U}^\top \right\|_F^2$$

  where $\|\cdot\|_F$ denotes the *Frobenius norm*:

$$\|\boldsymbol{Y}\|_F^2 = \left\| \boldsymbol{Y}^\top \right\|_F^2 = \sum_{i,j} y_{ij}^2 = \sum_i \left\| y^{(i)} \right\|^2$$

- PCA is approximating $\boldsymbol{X} \approx \boldsymbol{Z}\boldsymbol{U}^\top$, or equivalently $\boldsymbol{X}^\top \approx \boldsymbol{U}\boldsymbol{Z}^\top$
- Based on the sizes of the matrices, this is a *rank-K approximation*
- Since $\boldsymbol{U}$ was chosen to minimize reconstruction error, this is the *optimal rank-K approximation*, in terms of error $\left\| \boldsymbol{X}^\top - \boldsymbol{U}\boldsymbol{Z}^\top \right\|_F^2$

Singular Value Decomposition (SVD)

- Matrix factorization has a close relationship to the **singular value decomposition (SVD)** of $\boldsymbol{X}$, which is a matrix factorization technique
- Consider an $N \times D$ matrix $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ with SVD

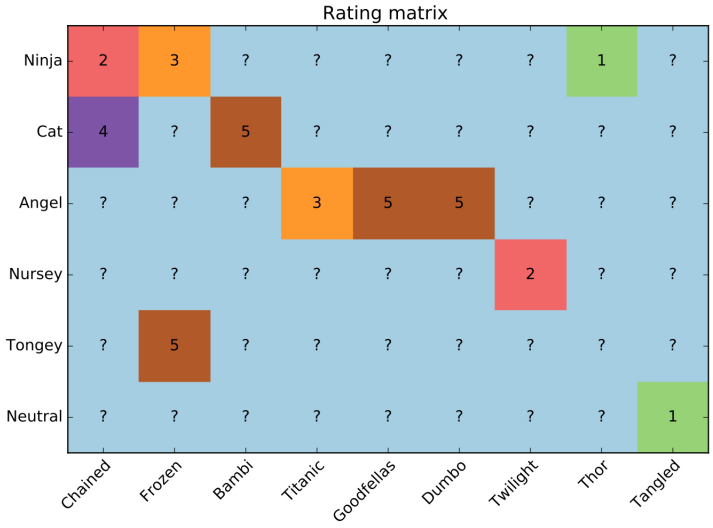
$$\boldsymbol{X} = \boldsymbol{Q}\boldsymbol{S}\boldsymbol{U}^\top$$

  - $\boldsymbol{Q}$, $\boldsymbol{S}$, and $\boldsymbol{U}^\top$ provide a real-valued matrix factorization of $\boldsymbol{X}$
  - $\boldsymbol{Q}$ is a $N \times D$ matrix with *orthonormal columns*, i.e. $\boldsymbol{Q}^\top\boldsymbol{Q} = \boldsymbol{I}_D$, where $\boldsymbol{I}_D$ is the $D \times D$ identity matrix
  - $\boldsymbol{U}$ is an orthonormal $D \times D$ matrix, i.e. $\boldsymbol{U}^\top = \boldsymbol{U}^{-1}$
  - $\boldsymbol{S}$ is a $D \times D$ diagonal matrix, with non-negative singular values $s_1, s_2, \ldots, s_D$ on the diagonal, where the singular values are conventionally ordered from largest to smallest

Matrix Completion

- PCA gives the optimal low-rank matrix factorization to a matrix $\boldsymbol{X}$
- We want to generalize this to the case where $\boldsymbol{X}$ is only *partially observed*
- E.g. movie recommendation problem: users watch movies and rate them. Because users only rate a few items, one would like to infer their preference for unrated items

- **Matrix completion problem**: transform the table into a N users by M movies matrix $\boldsymbol{R}$



  - Data: users rate some movies, $\boldsymbol{R}_{\text{user, movie}}$, very sparse
  - Task: predict missing entries, i.e. how a user would rate a movie they haven't previously rated
  - Evaluation metric: squared error

- *Latent factor models* attempt to explain the ratings by characterizing both movies and users on a number of factors K inferred from the ratings patterns

- We seek representations for movies and users as vectors in $\mathbb{R}^K$ that can ultimately be translated to ratings

- Let the representation of user i in the K-dimensional space be $\boldsymbol{u}_i$, and the representation of movie j be $\boldsymbol{z}_j$

  - Intuition: maybe the each entry of $\boldsymbol{u}_i$ is the user's preference on a particular type of movie, and each entry of $\boldsymbol{z}_j$ is how much does the movie belongs to a particular type

- Assume the rating user i gives to movie j is given by a dot product: $\boldsymbol{R}_{ij} = \boldsymbol{u}_i^\top \boldsymbol{z}_j$

- In matrix form, if

$$\boldsymbol{U} = \begin{bmatrix} - & \boldsymbol{u}_1^\top & - \\ & \vdots & \\ - & \boldsymbol{u}_N^\top & - \end{bmatrix} \quad \text{and} \quad \boldsymbol{Z}^\top = \begin{bmatrix} | & & | \\ \boldsymbol{z}_1 & \cdots & \boldsymbol{z}_M \\ | & & | \end{bmatrix}$$

  then $\boldsymbol{R} \approx \boldsymbol{U}\boldsymbol{Z}^\top$

- This becomes a matrix factorization problem

- Since most entries are missing, we only want to count the error for the observed entries

  - Let $O = \{(n, m) : \text{entry } (n, m) \text{ of matrix } \boldsymbol{R} \text{ is observed}\}$
  - Using the squared error loss, matrix completion requires solving

$$\min_{\boldsymbol{U},\boldsymbol{Z}} \frac{1}{2} \sum_{(i,j)\in O} \left(\boldsymbol{R}_{ij} - \boldsymbol{u}_i^\top \boldsymbol{z}_j\right)^2$$

  - The objective is non-convex in $\boldsymbol{U}$ and $\boldsymbol{Z}$ jointly, and it is NP-hard to minimize the above cost function exactly

- As a function of either $\boldsymbol{U}$ or $\boldsymbol{Z}$ individually, the problem is convex and easy to optimize
- **Alternating least squares (ALS)**: fix $\boldsymbol{Z}$ and optimize $\boldsymbol{U}$, followed by fixing $\boldsymbol{U}$ and optimizing $\boldsymbol{Z}$, and so on until convergence

Alternating Least Squares

- Want to minimize the squared error cost w.r.t. the factor $\boldsymbol{U}$ (the case of $\boldsymbol{Z}$ is exactly symmetric)
- Can decompose the cost into a sum of independent terms

$$\sum_{(i,j)\in O} \left(\boldsymbol{R}_{ij} - \boldsymbol{u}_i^\top \boldsymbol{z}_j\right)^2 = \sum_i \sum_{j:(i,j)\in O} \left(\boldsymbol{R}_{ij} - \boldsymbol{u}_i^\top \boldsymbol{z}_j\right)^2$$

  which could be minimized independently for each $\boldsymbol{u}_i$

- This is a *linear regression problem*, whose optimal solution is

$$\boldsymbol{u}_i = \left(\sum_{j:(i,j)\in O} \boldsymbol{z}_j \boldsymbol{z}_j^\top\right)^{-1} \sum_{j:(i,j)\in O} \boldsymbol{R}_{ij} \boldsymbol{z}_j$$

- Similarly for $\boldsymbol{Z}$:

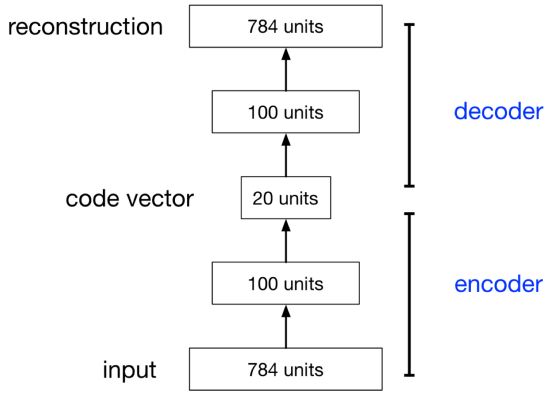
$$\boldsymbol{z}_j = \left(\sum_{i:(i,j)\in O} \boldsymbol{u}_i \boldsymbol{u}_i^\top\right)^{-1} \sum_{i:(i,j)\in O} \boldsymbol{R}_{ij} \boldsymbol{u}_i$$

```
ALS:
    initialize U and Z randomly
    while not reaching convergence do
        for i <- 1, ..., N do
            compute u_i as above
        for j <- 1, ..., M do
            compute z_j as above
```

17 Autoencoder

Overview

- An **autoencoder** is a *feed-forward neural net* whose job is to take an input $\boldsymbol{x}$ and predict $\boldsymbol{x}$
- To make this nontrivial, we need to add a *bottleneck layer* whose dimension is much smaller than the input

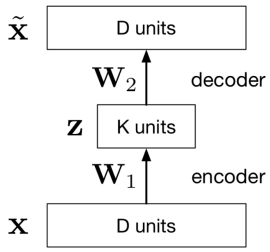


- Can map high-dimensional data to 2 dimensions for visualization
- Can learn *abstract features* in an *unsupervised* way so we could apply them to a supervised task (since unlabelled data is much more abundant than labelled data)

Linear Autoencoders

- The simplest kind of autoencoder has 1 hidden layer, linear activations, and squared error loss

$$\mathcal{L}(\boldsymbol{x}, \tilde{\boldsymbol{x}}) = \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|^2$$



- The network computes $\tilde{\boldsymbol{x}} = \boldsymbol{W}_2\boldsymbol{W}_1\boldsymbol{x}$, which is a linear function
- If $K \geq D$, then we can choose $\boldsymbol{W}_2$ and $\boldsymbol{W}_1$ such that $\boldsymbol{W}_2\boldsymbol{W}_1 = \boldsymbol{I}$
- Suppose $K < D$, then $\boldsymbol{W}_1$ maps $\boldsymbol{x}$ to a K-dimensional space, so it's doing *dimensionality reduction*
- The output of the autoencoder must lie in a K-dimensional subspace spanned by the columns of $\boldsymbol{W}_2$ because $\tilde{\boldsymbol{x}} = \boldsymbol{W}_2\boldsymbol{z}$
- The best possible K dimensional linear subspace in terms of reconstruction error is the PCA subspace
- The autoencoder can achieve this by setting $\boldsymbol{W}_1 = \boldsymbol{U}^\top$ and $\boldsymbol{W}_2 = \boldsymbol{U}$
- The optimal weights for a linear autoencoder are the principle components

Nonlinear Autoencoders

- Deep nonlinear autoencoders learn to project the data onto a *nonlinear manifold*, which is the image of the decoder
- This is a kind of *nonlinear dimensionality reduction*
- Can learn more powerful codes for a given dimensionality, compared with linear autoencoders (e.g. PCA)

18 K-Means

Clustering

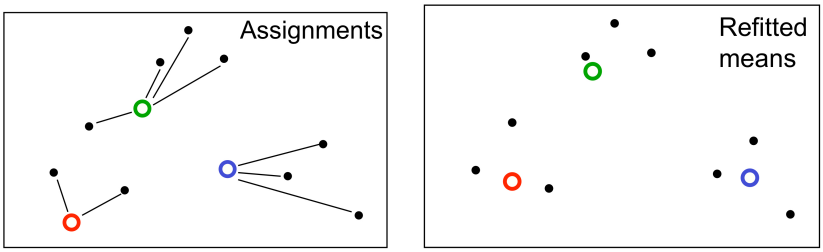
- Sometimes the data form clusters, where samples within a cluster are similar to each other, and samples in different clusters are dissimilar
- Such a distribution is *multimodal*, since it has multiple *modes* (i.e. regions of high probablity mass)
- **Clustering**: grouping data points into clusters, with *no observed labels*
- Unsupervised learning

K-Means Intuition

- There are k clusters, and each point is close to its cluster center (i.e. mean)
- Given the cluster assignments, we could compute the cluster centers
- Given the cluster centers, we could compute the cluster assignments
- We can start randomly and alternate between the two

K-Means Algorithm

- Randomly initialize cluster centers
- Alternate between two steps:
  1. **Assignment step**: assign each data point to the closest cluster
  2. **Refitting step**: move each cluster center to the mean of its members



K-Means Objective

- Find cluster centers $\boldsymbol{m}$ and assignments $\boldsymbol{r}$ to minimize the sum of squared distances of data points $\left\{\boldsymbol{x}^{(n)}\right\}$ to their assigned cluster centers

$$\min_{\{\boldsymbol{m}\},\{\boldsymbol{r}\}} J(\{\boldsymbol{m}\},\{\boldsymbol{r}\}) = \min_{\{\boldsymbol{m}\},\{\boldsymbol{r}\}} \sum_{n=1}^{N}\sum_{k=1}^{K} r_k^{(n)} \left\|\boldsymbol{m}_k - \boldsymbol{x}^{(n)}\right\|^2$$

  such that $\sum_k r_k^{(n)} = 1$ for all n, where $r_k^{(n)} \in \{0,1\}$ for all k and n

  - $r_k^{(n)} = 1$ if $\boldsymbol{x}^{(n)}$ is assigned to cluster k
  - We sum over the loss of each data point to its assigned cluster center
- Finding the exact optimum is NP-hard
- K-means can be seen as block coordinate descent on this objective

  - Analogous to ALS for matrix completion
  - Assignment step: minimize w.r.t. $\left\{r_k^{(n)}\right\}$
  - Refitting step: minimize w.r.t. $\{\boldsymbol{m}_k\}$
- If we fit the centers $\{\boldsymbol{m}_k\}$, then we can find the optimal assignments $\left\{\boldsymbol{r}^{(n)}\right\}$ for each sample n

$$\min_{\boldsymbol{r}^{(n)}} \sum_{k=1}^{K} r_k^{(n)} \left\|\boldsymbol{m}_k - \boldsymbol{x}^{(n)}\right\|^2$$

- Assign each point to the cluster with the nearest center

$$r_k^{(n)} = \begin{cases} 1, & \text{if } k = \arg\min_j \left\|\boldsymbol{m}_j - \boldsymbol{x}^{(n)}\right\|^2 \\ 0, & \text{elsewise} \end{cases}$$

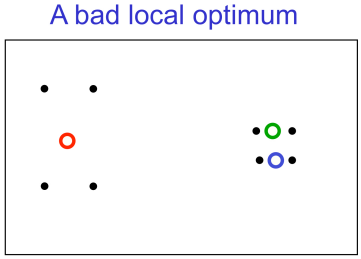
- If we fix the assignments $\left\{\boldsymbol{r}^{(n)}\right\}$, then we can find the optimal centers $\{\boldsymbol{m}_k\}$

$$\frac{\partial}{\partial \boldsymbol{m}_l} \sum_{n=1}^{N} \sum_{k=1}^{K} r_k^{(n)} \left\|\boldsymbol{m}_k - \boldsymbol{x}^{(n)}\right\|^2 = 0$$

$$\boldsymbol{m}_l = \frac{\sum_n r_l^{(n)} \boldsymbol{x}^{(n)}}{\sum_n r_l^{(n)}}$$

Convergence of K-Means

- K-means algorithm reduces the cost at each iteration
  - Whenever an assignment is changed, the sum of squared distances J of data points from their assigned cluster centers is reduced
  - Whenever a cluster center is moved, J is reduced
- Test for convergence: if the assignments do not change in the assignment step, we have converged (to at least a local minimum)
- This convergence will always happen after a finite number of iterations, since the number of possible cluster assignments is finite
- The objective J is non-convex, so that coordinate descent on J is not guaranteed to converge to the global minimum, e.g.



  - We could try many random starting points

Soft K-Means

- We could make *soft assignments* of data points to clusters, e.g. one cluster may have a responsibility of 0.7 for a data point and another may have a responsibility of 0.3
  - Allows a cluster to use more information about the data in the refitting step
  - We could change the 1-of-K encoding to softmax assignments

Soft K-Means Algorithm

- Initialization: set K means $\{\boldsymbol{m}_k\}$ to random values
- Repeat ountil convergence (measured by how much J changes):
  - Assignment: each data point n given soft "degree of assignment" to each cluster mean k, based on responsibilities

$$r_k^{(n)} = \frac{\exp\left[-\beta \left\|\boldsymbol{m}_k - \boldsymbol{x}^{(n)}\right\|^2\right]}{\sum_j \exp\left[-\beta \left\|\boldsymbol{m}_j - \boldsymbol{x}^{(n)}\right\|^2\right]} \implies \boldsymbol{r}^{(n)} = \text{softmax}\left(-\beta \left\{\left\|\boldsymbol{m}_k - \boldsymbol{x}^{(n)}\right\|^2\right\}_{k=1}^{K}\right)$$

  - Refitting: model parameters, means, are adjusted to match sample means of datapoints they are responsible for

$$\boldsymbol{m}_k = \frac{\sum_n r_k^{(n)} \boldsymbol{x}^{(n)}}{\sum_n r_k^{(n)}}$$

- Issues
  - How to set β?
  - How to handle clusters with unequal weight and width?
- As $\beta \to \infty$, soft K-means becomes K-means

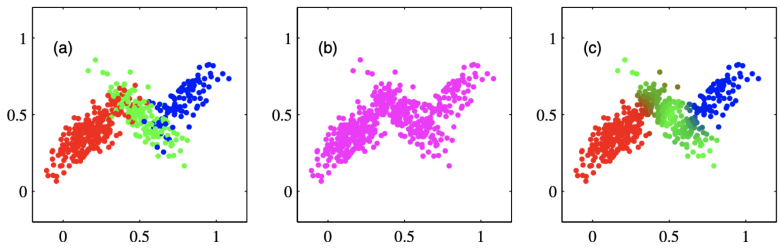
19 Gaussian Mixture Models

Generative Models

- For generative (Bayes) classifiers
$$\Pr(\boldsymbol{x}, t) = \Pr(\boldsymbol{x} \mid t) \Pr(t)$$
we fit $\Pr(t)$ and $\Pr(\boldsymbol{x} \mid t)$ using labelled data
- If t is never observed, we call it a *latent variable*, or *hidden variable*, and generally denote it with z instead
- The things that we can observe (i.e. $\boldsymbol{x}$) are called *observables*
- By marginalizing out z, we get a density over the observables:
$$\Pr(\boldsymbol{x}) = \sum_z p(\boldsymbol{x}, z) = \sum_z \Pr(\boldsymbol{x} \mid z) \Pr(z)$$
  - This is called a *latent variable model*
- If $\Pr(z)$ is a categorical distribution, this is a *mixture model*, and different values of z correspond to different *components*

Gaussian Mixture Model

- Most common mixture model: Gaussian mixture model (GMM)
- A GMM represents a distribution as
$$\Pr(\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$
with π_k the **mixing coefficients**, where
$$\sum_{k=1}^{K} \pi_k = 1 \quad \text{and} \quad \pi_k \geq 0 \quad \text{for all } k$$
- This defines a density over $\boldsymbol{x}$, so we can fit the parameters using maximum likelihood
- We are trying to match the data density of $\boldsymbol{x}$ as closely as possible
- GMMs are *universal approximators of densities* (i.e. they can approximate any density)
  - Even diagonal GMMs are universal approximators
- Writing the model as a *generative process*: for $i = 1, \ldots, N$,
$$z^{(i)} \sim \text{Categorical}(\boldsymbol{\pi})$$
$$\boldsymbol{x}^{(i)} \mid z^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}_{z^{(i)}}, \boldsymbol{\Sigma}_{z^{(i)}})$$



a) Samples from $p(\mathbf{x} \mid z)$ b) Samples from the marginal $p(\mathbf{x})$ c) Responsibilities $p(z \mid \mathbf{x})$

Maximum Likelihood with Latent Variables

- We need to choose the parameters $\{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$
- Maximum likelihood principle: choose parameters to maximize likelihood of *observed data*
- We don't observe the cluster assignments z, we only see the data $\boldsymbol{x}$
- Given data $\mathcal{D} = \left\{\boldsymbol{x}^{(n)}\right\}_{n=1}^N$, choose parameters to maximize

$$\log \Pr(\mathcal{D}) = \sum_{n=1}^N \log \Pr\left(\boldsymbol{x}^{(n)}\right)$$

- Can find $\Pr(\boldsymbol{x})$ by marginalizing out z:

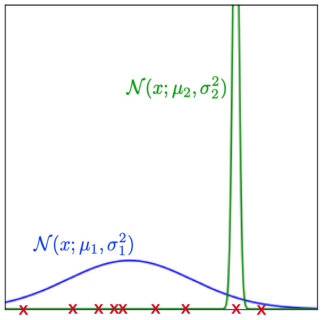
$$\Pr(\boldsymbol{x}) = \sum_{k=1}^K \Pr(z = k, \boldsymbol{x}) = \sum_{k=1}^K \Pr(z = k) \Pr(\boldsymbol{x} \mid z = k)$$

Using Maximum Likelihood to Fit GMMs

- Shorthand notations:
  - Let $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$ denote the full set of model parameters
  - Let $\boldsymbol{X} = \left\{\boldsymbol{x}^{(i)}\right\}$
  - Let $\boldsymbol{Z} = \left\{\boldsymbol{z}^{(i)}\right\}$
- Maximum likelihood objective:

$$\log \Pr(\boldsymbol{X}; \boldsymbol{\theta}) = \sum_{i=1}^N \log \left(\sum_{k=1}^K \pi_k \mathcal{N}(\boldsymbol{x}^{(i)}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

- No closed-form solution
- Not *identifiable*: solution is invariant to permutations
- Challenges in optimizing this using gradient descent:
  - Non-convex (due to permutation symmetry)
  - Need to enforce non-negativity constraint on π_k and PSD constraint on $\boldsymbol{\Sigma}_k$
  - Derivatives w.r.t. $\boldsymbol{\Sigma}_k$ are expensive/complicated
- However we don't want the global maximum, since we could achieve arbitrarily high training likelihood by placing a small-variance Gaussian component on a training example
  - Known as *singularity*



Latent Variable Models

- Inferences
  - If we knew the parameters $\boldsymbol{\theta} = \{\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}$, we could infer which component a data point $\boldsymbol{x}^{(i)}$ probably belongs to by inferring its latent variable $z^{(i)}$
  - We could do this posterior inference using Bayes' Rule

$$\Pr\left(z^{(i)} = k \mid \boldsymbol{x}^{(i)}\right) = \frac{\Pr(z = k)\Pr(\boldsymbol{x} \mid z = k)}{\sum_l \Pr(z = l)\Pr(\boldsymbol{x} \mid z = l)}$$

- Learning
  - If we knew the latent variables for every data point, we could maximize the joint log-likelihood

$$\begin{aligned}\log \Pr(\boldsymbol{X}, \boldsymbol{Z}; \boldsymbol{\theta}) &= \sum_{i=1}^{N} \log \Pr\left(\boldsymbol{x}^{(i)}, z^{(i)}; \boldsymbol{\theta}\right) \\ &= \sum_{i=1}^{N} \left[\log \Pr\left(z^{(i)}\right) + \log \Pr\left(\boldsymbol{x}^{(i)} \mid z^{(i)}\right)\right]\end{aligned}$$

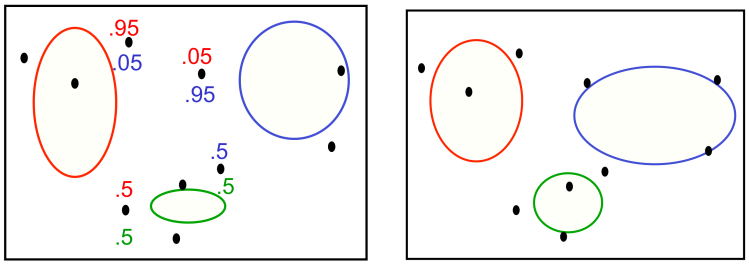
- Problem: we don't know the $z^{(i)}$, so we need to marginalize them out:

$$\begin{aligned}\log \Pr(\boldsymbol{X}; \boldsymbol{\theta}) &= \sum_{i=1}^{N} \log \Pr\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{\theta}\right) \\ &= \sum_{i=1}^{N} \log \left(\sum_{z^{(i)}=1}^{K} \Pr\left(\boldsymbol{x}^{(i)} \mid z^{(i)}; \{\boldsymbol{\mu}_k\}, \{\boldsymbol{\Sigma}_k\}\right) \Pr\left(z^{(i)} \mid \boldsymbol{\pi}\right)\right)\end{aligned}$$

  - The log is outside the sum, so things don't simplify
- Given $\boldsymbol{\theta}$, inferring the $z^{(i)}$ is easy
- Given the $z^{(i)}$, learning $\boldsymbol{\theta}$ (with maximum likelihood) is easy

Expectation Maximization Algorithm

- Alternates between 2 steps
  1. **Expectation step (E-step)**: compute the posterior probability over z given our current model, i.e. how much do we think each Gaussian generates each datapoint
  2. **Maximization step (M-step)**: assuming that the data really was generated this way, change the parameters of each Gaussian to maximize the probability that it would generate the data it is currently responsible for



Expectation Maximization for GMM

1. **E-step**: assign the *responsibility* $r_k^{(i)}$ of component k for data point i using the posterior probability

$$r_k^{(i)} = \Pr\left(z^{(i)} = k \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)$$

2. **M-step**: apply the maximum likelihood updates, where each component is fit with a weighted dataset. The weights are proportional to the responsibilities

$$\pi_k = \frac{1}{N} \sum_{i=1}^{N} r_k^{(i)} \qquad \text{Weights}$$

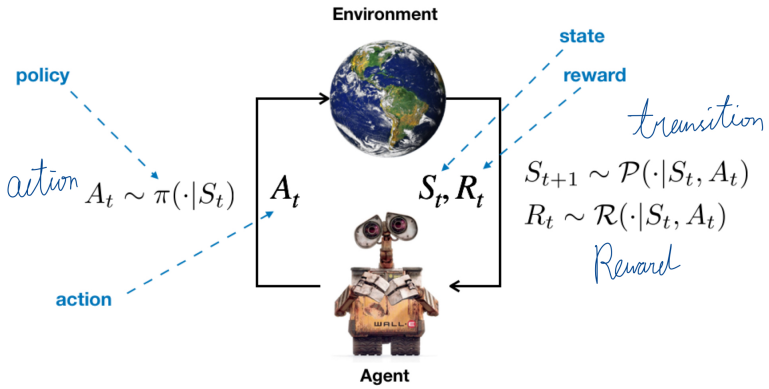
$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} r_k^{(i)} \cdot \boldsymbol{x}^{(i)}}{\sum_{i=1}^{N} r_k^{(i)}} \qquad \text{Center}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} r_k^{(i)} \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_k\right) \left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_k\right)^T}{\sum_{i=1}^{N} r_k^{(i)}} \qquad \text{Shape}$$

20 Reinforcement Learning

Markov Decision Process

- Most RL is done in the Markov Decision Process framework



- States and actions
  - The **state** is a description of the environment in sufficient detail to determine its evolution
    - Analogous to Newtonian physics
    - *Markov assumption*: the state at time $t+1$ depends directly on the state and action at time t, and not on past states or actions
  - To describe the *dynamics*, we need to specify the *transition probabilities* $\mathcal{P}(S_{t+1} \mid S_t, A_t)$
  - We assume the state is *fully observable*
- Policies
  - **Policy**: the way the agent chooses the action in each step
  - Two types of policies:
    1. **Deterministic policy**: $A_t = \pi(S_t)$ for some function $\pi : \mathcal{S} \to \mathcal{A}$
    2. **Stochastic policy**: $A_t \sim \pi(\cdot \mid S_t)$ for some function $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, where $\mathcal{P}(\mathcal{A})$ is the set of distributions over actions
  - With stochastic policies, the distribution over *rollouts*, or *trajectories*, factorizes

$$\Pr(s_1, a_1, \ldots, s_T, a_T) = \Pr(s_1)\pi(a_1 \mid s_1)\mathcal{P}(s_2 \mid s_1, a_1)\pi(a_2 \mid s_2) \cdots \mathcal{P}(s_T \mid s_{T-1}, a_{T-1})\pi(a_T \mid s_T)$$

  - Policies only need to consider the current state because of the Markov assumption and full observability
    - If the environment is partially observable, then the policy needs to depend on the history of observations
- Rewards
  - In each time step, the agent receives a reward from a distribution that depends on the current state and action

$$R_t \sim \mathcal{R}(\cdot \mid S_t, A_t)$$

  - Assume rewards are deterministic, i.e.

$$R_t = r(S_t, A_t)$$

  - The **return** determines how good was the outcome of an episode

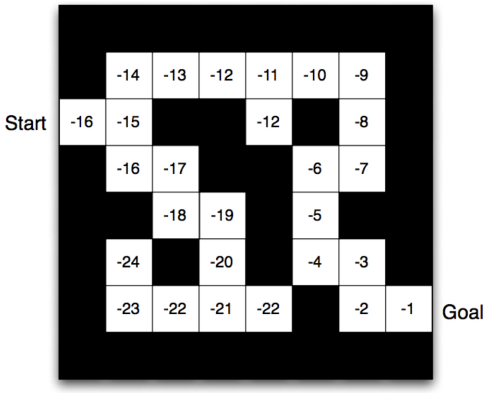
    - *Undiscounted*: $G = R_0 + R_1 + R_2 + \cdots$ (sum of all rewards)
    - *Discounted*: $G = R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots$
  - The goal is to maximize the expected return, $\mathbb{E}[G]$
  - γ is a hyperparameter called the **discount factor**, which determines how much we care about rewards now vs. rewards later
- A **Markov Decision Process (MDP)** is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$
  - $\mathcal{S}$ is the state space, could be discrete or continuous
  - $\mathcal{A}$ is the action space, which we consider finite, i.e. $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$
  - $\mathcal{P}$ is the transition probability
  - $\mathcal{R}$ is the immediate reward distribution
  - $\gamma \in [0, 1)$ is the discount factor
- Want to find a policy that achieves a high return
- Two situations:
  1. *Planning*: given a fully specified MDP
  2. *Learning*: agent interacts with an environment with unknown dynamics, i.e. the environment is a black box that takes in actions and outputs states and rewards

Value Function

- The **value function** V^π for a policy π measures the expected return if we start in state s and follow policy π

$$V^\pi(s) \triangleq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid S_t = s\right]$$

- Measures the desirability of state s
- E.g. numbers represent value $V^\pi(s)$ of each state s



Bellman Equations

- Value functions satisfy a recursive relationship, called the **Bellman equation**

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \qquad \text{Value function} \\ &= \mathbb{E}[R_t + \gamma G_{t+1} \mid S_t = s] \qquad \text{Expand} \\ &= \sum_a \pi(a \mid s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \right] \\ &= \sum_a \pi(a \mid s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) V^\pi(s') \right] \end{aligned}$$

- Viewing V^π as a vector (where entries corresopnd to states) define the **Bellman backup operator** T^π

$$(T^\pi V)(s) \triangleq \sum_a \pi(a \mid s) \left[r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) V(s') \right]$$

- The Bellman equation can be seen as a *fixed point* of the Bellman operator

$$T^\pi V^\pi = V^\pi$$

State-Action Value Function

- The **state-action value function**, or **Q-function**, Q^π for policy π, is defined as

$$Q^\pi(s,a) \triangleq \mathbb{E}_\pi \left[\sum_{k \geq 0} \gamma^k R_{t+k} \mid S_t = s, A_t = a \right]$$

- If we knew Q^π, then we could obtain V^π:

$$V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s,a)$$

- If we knew V^π, then we could obtain Q^π:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) V^\pi(s')$$

  - However this requires knowing the dynamics

- Q^π satisfies a Bellman equation very similar to V^π

$$Q^\pi(s,a) = \underbrace{r(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid a, s) \sum_{a'} \pi(a' \mid s') Q^\pi(s', a')}_{\triangleq (T^\pi Q^\pi)(s,a)}$$

Optimal State-Action Value Function

- Suppose we're in state s and want to pick an action a, and then follow policy π from then on, we would pick

$$\arg\max_a Q^\pi(s,a)$$

  where Q^π is the expected reward from that state

- If a deterministic policy π is optimal, then for any state s:

$$\pi(s) = \arg\max_a Q^\pi(s, a)$$

  otherwise we could improve the policy by changing $\pi(s)$

- Bellman equation for optimal policy π^*:

$$\begin{aligned} Q^{\pi^*}(s, a) &= r(s, a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s, a) Q^{\pi^*}(s', \pi^*(s')) \\ &= r(s, a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s, a) \max_{a'} Q^{\pi^*}(s', a') \qquad \text{Since we're using the optimal policy} \end{aligned}$$

- Now $Q^* = Q^{\pi^*}$ is the **optimal state-action value function**, and we can rewrite the *optimal Bellman equation* without mentioning π^*:

$$Q^*(s, a) = \underbrace{r(s, a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s, a) \max_{a'} Q^*(s', a')}_{\triangleq (T^* Q^*)(s, a)}$$

- This is *sufficient* to characterize the optimal policy, so we would need to solve the fixed point equation $T^* Q^* = Q^*$ and then choose $\pi^*(s) = \arg\max_a Q^*(s, a)$

Bellman Fixed Points

- We know that some problems could be reduced to finding fixed points of Bellman backup operators:
  - Evaluating a fixed policy π:
$$T^\pi Q^\pi = Q^\pi$$
  - Finding the optimal policy:
$$T^* Q^* = Q^*$$

- We could keep iterating the backup operator

$$\begin{aligned} Q &\leftarrow T^\pi Q \qquad \text{Policy evaluation} \\ Q &\leftarrow T^* Q \qquad \text{Find optimal policy} \end{aligned}$$

  - We are treating Q^π or Q^* as vector with $|\mathcal{S}| \cdot |\mathcal{A}|$ entries
  - This type of algorithm is an instance of *dynamic programming*

- An operator f is a **contraction map** if

$$\|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)\| \leq \alpha \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$$

  for some scalar $0 \leq \alpha < 1$ and vector norm $\|\cdot\|$

- Let $f^{(k)}$ denote f iterated k times, it can be shown from induction that

$$\left\| f^{(k)}(\boldsymbol{x}_1) - f^{(k)}(\boldsymbol{x}_2) \right\| \leq \alpha^k \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$$

- Let $\boldsymbol{x}^*$ be a fixed point of f. Then for any $\boldsymbol{x}$:

$$\left\| f^{(k)}(\boldsymbol{x}) - \boldsymbol{x}^* \right\| \leq \alpha^k \|\boldsymbol{x} - \boldsymbol{x}^*\|$$

- Iterated application of f, starting from any $\boldsymbol{x}$, converges exponentially to a unique fixed point

Value Iteration

- **Value iteration**: start from an initial function Q_1. For each $k = 1, 2, \ldots$, apply

$$Q_{k+1} \leftarrow T^* Q_k$$

- Writing out the update in full:

$$Q_{k+1}(s, a) \leftarrow r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s, a) \max_{a' \in \mathcal{A}} Q_k(s', a')$$

- The value iteration update is a contraction map:

$$\left\| T^* Q_1 - T^* Q_2 \right\|_\infty \leq \gamma \left\| Q_1 - Q_2 \right\|_\infty$$

  where $\|\cdot\|_\infty$ is the L^∞ norm, defined as

$$\|\boldsymbol{x}\|_\infty = \max_i |x_i|$$

- This means that value iteration converges exponentially to the unique fixed point
- The exponential decay factor is λ (i.e. the discount factor), which means that longer term planning is harder
- Limitations
  - This assumes known dynamics
  - Requires explicitly representing Q^* as a vector
    - $|\mathcal{S}|$ can be extremely large, or infinite
    - $|\mathcal{A}|$ can be infinite (e.g. continuous)

Monte Carlo Estimation

- Optimal Bellman equation:

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{\mathcal{P}(s' \mid s, a)} \left[\max_{a'} Q^*(s', a') \right]$$

- We need to know the dynamics to evaluate the expectation
- **Monte Carlo estimation** of an expectation $\mu = \mathbb{E}[X]$ is to repeatedly sample X and update, i.e.

$$\mu \leftarrow \mu + \alpha(X - \mu)$$

- We could apply Monte Carlo estimation to the Bellman equation by sampling $S' \sim \mathcal{P}(\cdot \mid s, a)$ and updating:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[\underbrace{r(s, a) + \gamma \max_{a'} Q(S', a') - Q(s, a)}_{\text{Bellman error}} \right]$$

- This is an example of *temporal difference learning*, i.e. updating our predictions to match our later predictions (once we have more information)
- A problem is that every iteration of value iteration requires updating Q for every state

  - There could be lots of states
  - We only observe transitions for states that are visited
- We could only update Q for the states that are actually visited
- Another problem would be that we might never visit certain states if they don't look promising, so we'll never learn about them
- We could address that problem by having the agent to sometimes take random actions so that it eventually visits every state
  - *ε-greedy policy*: a policy which picks $\arg\max_a Q(s,a)$ with probability $1-\varepsilon$ and a random action with probability ε
  - A reasonble value for ε is 0.05
- Doing everything all at once is called **Q-learning**

Q-Learning with ε-Greedy Policy

- Parameters: learning rate α and exploration parameter ε
- Initialize $Q(s,a)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$
- The agent starts at state S_0
- For time step $t = 0, 1, \ldots$:
  - Choose A_t according to the *ε-greedy policy*, i.e.

$$A_t \leftarrow \begin{cases} \arg\max_{a \in \mathcal{A}} Q(S_t, a) & \text{with probability } 1-\varepsilon \\ \text{Uniformly random action in } \mathcal{A} & \text{with probability } \varepsilon \end{cases}$$

  - Take action A_t in the environment
  - The state changes from S_t to $S_{t+1} \sim \mathcal{P}(\cdot \mid S_t, A_t)$
  - Observe S_{t+1} and R_t (could be $r(S_t, A_t)$, or stochastic)
  - Update the action-value function at state-action (S_t, A_t):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_t + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t) \right]$$

Exploration vs. Exploitation

- The ε-greedy is a simple mechanism for managing the **exploration-exploitation tradeoff**

$$\pi_\varepsilon(S; Q) = \begin{cases} \arg\max_{a \in \mathcal{A}} Q(S, a) & \text{with probability } 1-\varepsilon \\ \text{Uniformly random action in } \mathcal{A} & \text{with probability } \varepsilon \end{cases}$$

- This policy ensures that most of the time, the agent exploits its incomplete knowledge of the world by choosing the best action, but occassionally it explores other actions
- Without exploration, the agent may never find some good actions
- Example of exploration-exploitation: go to the favourite restaurant vs. try a new restaurant

Off-Policy Learning

- The Q-learning update does not mention the policy anywhere, the only thing the policy is used for is to determine which states are visited
- This means that we could follow whatever policy we want (e.g. ε-greedy), and it still converges to the optimal Q-function
- Algorithms like this are called **off-policy algorithms**
- An example of an **on-policy algorithm** is policy graidient, where it is hard to encourage exploration

Function Approximation

- So far, we've been assuming a *tabular representation* of Q (i.e. matrix; one entry for every state/action pair)
- This is impractical since matrices can get very large
- We could approximate Q using a parameterized function, e.g.
  - A linear function approximation: $Q(\boldsymbol{s}, \boldsymbol{a}) = \boldsymbol{w}^\top \psi(\boldsymbol{s}, \boldsymbol{a})$
  - Compute Q with a neural net
- We could update Q using backprop:

$$t \leftarrow r(\boldsymbol{s}_t, \boldsymbol{a}_t) + \gamma \max_{\boldsymbol{a}} Q(\boldsymbol{s}_{t+1}, \boldsymbol{a})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(t - Q(\boldsymbol{s}, \boldsymbol{a}))\nabla_{\boldsymbol{\theta}} Q(\boldsymbol{s}_t, \boldsymbol{a}_t)$$