MITx 6.86x - Machine Learning with Python: from Linear Models to Deep Learning

https://www.edx.org/course/machine-learning-with-python-from-linear-models-to

Lecturers: Regina Barzilay, Tommi Jaakkola, Karene Chu

Notes assembled by: Antonello Lobianco

Student's notes (2020 run)

Disclaimer: The following notes are a mesh of my own notes, selected transcripts, some useful forum threads and various course material. I do not claim any authorship of these notes, but at the same time any error could well be arising from my own interpretation of the course material.

Contributions are really welcome. If you spot an error , want to specify something in a better way (English is not my primary language), add material or just have comments, you can clone, make your edits and make a pull request (preferred) or just open an issue.

(This PDF versions may be outdated ! Please refer to the GitHub repository source files for latest version.)

Table of contents

- Unit 00 - Course Overview, Homework 0, Project 0
  - Recitation 1 - Brief Review of Vectors, Planes, and Optimization
    - Points and Vectors
    - Vector projections
    - Planes
  - Loss Function, Gradient Descent, and Chain Rule
  - Geometric progressions and series
- Unit 01 - Linear Classifiers and Generalizations
  - Course Introduction
  - Lecture 1. Introduction to Machine Learning
    - 1.1. Unit 1 Overview
    - 1.2. Objectives
    - 1.3. What is Machine Learning?
      - Prediction
    - 1.4. Introduction to Supervised Learning
    - 1.5. A Concrete Example of a Supervised Learning Task
    - 1.6. Introduction to Classifiers: Let's bring in some geometry!
    - 1.7. Different Kinds of Supervised Learning: classification vs regression
  - Lecture 2. Linear Classifier and Perceptron Algorithm
    - 2.1. Objectives
    - 2.2. Review of Basic Concepts
      - What is this lecture going to be about ?
    - 2.3. Linear Classifiers Mathematically Revisited
      - Through the origin classifiers
      - General linear classifiers
    - 2.4. Linear Separation
    - 2.5. The Perceptron Algorithm
      - Through the origin classifier
      - Generalised linear classifier
  - Lecture 3 Hinge loss, Margin boundaries and Regularization
    - 3.1. Objective
    - 3.2. Introduction

        - 3.3. Margin Boundary
        - 3.4. Hinge Loss and Objective Function
            - A bit of terminology
    - Homework 1
        - 1. Perceptron Mistakes
            - 1. (e) Perceptron Mistake Bounds
    - Lecture 4. Linear Classification and Generalization
        - 4.1. Objectives
        - 4.2. Review and the Lambda parameter
            - The role of the lambda parameter
        - 4.3. Regularization and Generalization
        - 4.4. Gradient Descent
            - Multi-dimensional case
        - 4.5. Stochastic Gradient Descent
            - An other point of view on SGD compared to a deterministic approach:
        - 4.6. The Realizable Case - Quadratic program
    - Homework 2
    - Project 1: Automatic Review Analyzer
    - Recitation 1: Tuning the Regularization Hyperparameter by Cross Validation and a Demonstration
        - Outline
        - Supervised Learning
        - Support Vector Machine
            - Loss term
            - Regularisation term
            - Objective function
        - Maximum margin
        - Testing and Training Error as Regularization Increases
        - Cross Validation
        - Tumor Diagnosis Demo
- Unit 02 - Nonlinear Classification, Linear regression, Collaborative Filtering
    - Lecture 5. Linear Regression
        - 5.1. Unit 2 Overview
        - 5.2. Objectives
        - 5.3. Introduction
        - 5.4. Empirical Risk
        - 5.5. Gradient Based Approach
        - 5.6. Closed Form Solution
        - 5.7. Generalization and Regularization
        - 5.8. Regularization
            - Gradient based approach with regularisation
            - The closed form approach with regularisation
        - 5. 9. Closing Comment
    - Lecture 6. Nonlinear Classification
        - 6.1. Objectives
        - 6.2. Higher Order Feature Vectors
        - 6.3. Introduction to Non-linear Classification
        - 6.4. Motivation for Kernels: Computational Efficiency
        - 6.5. The Kernel Perceptron Algorithm
        - 6.6. Kernel Composition Rules
        - 6.7. The Radial Basis Kernel
            - Other non-linear classifiers
            - Summary
    - Lecture 7. Recommender Systems
        - 7.1. Objectives
        - 7.2. Introduction
        - Problem definition
        - 7.3. K-Nearest Neighbor Method
        - 7.4. Collaborative Filtering: the Naive Approach
        - 7.5. Collaborative Filtering with Matrix Factorization
        - 7.6. Alternating Minimization
            - Numerical example

    - Project 2: Digit recognition (Part 1)
        - The softmax function
- Unit 03 - Neural networks
    - Lecture 8. Introduction to Feedforward Neural Networks
        - 8.1. Unit 3 Overview
        - 8.2. Objectives
        - 8.3. Motivation
            - Neural networks vs the non-linear classification methods that we saw already
        - 8.4. Neural Network Units
        - 8.5. Introduction to Deep Neural Networks
            - Overall architecture
            - Subject areas
            - Deep learning … why now?
        - 8.6. Hidden Layer Models
            - 2-D Example
            - Summary
    - Lecture 9. Feedforward Neural Networks, Back Propagation, and Stochastic Gradient Descent (SGD)
        - 9.1. Objectives
        - 9.2. Back-propagation Algorithm
        - 9.3. Training Models with 1 Hidden Layer, Overcapacity, and Convergence Guarantees
            - Summary
    - Lecture 10. Recurrent Neural Networks 1
        - 10.1. Objective
        - 10.2. Introduction to Recurrent Neural Networks
            - Exchange rate example
            - Language completion example
            - Limitations of these approaches
        - 10.3. Why we need RNNs
        - 10.4. Encoding with RNN
        - 10.5. Gating and LSTM
            - Learning RNNs
            - Simple gated RNN
            - Long Short Term Memory neural networks
            - Key things
    - Lecture 11. Recurrent Neural Networks 2
        - 11.1. Objective
        - 11.2. Markov Models
            - Outline
            - Markov models
            - Markov textual bigram model exemple
            - Outline detailed
        - 11.3. Markov Models to Feedforward Neural Nets
            - Advantages of neural network representation
        - 11.4. RNN Deeper Dive
        - 11.5. RNN Decoding
            - Key summary
    - Homework 4
    - Lecture 12. Convolutional Neural Networks (CNNs)
    - 12.1. Objectives
        - 12.2. Convolutional Neural Networks
        - 12.3. CNN - Continued
            - Take home
        - Recitation: Convolution/Cross Correlation:
            - 1-D Discrete version
            - 2-D Discrete version
    - Project 3: Digit recognition (Part 2)
- Unit 04 - Unsupervised Learning
    - Lecture 13. Clustering 1
        - 13.1. Unit 4: Unsupervised Learning
        - 13.2. Objectives
        - 13.3. Introduction to Clustering

        - 13.4. Another Clustering Example: Image Quantization
        - 13.5. Clustering Definition
            - Partitioning
            - Representativeness
        - 13.6. Similarity Measures-Cost functions
        - 13.7. The K-Means Algorithm: The Big Picture
        - 13.8. The K-Means Algorithm: The Specifics
            - Finding the best representatives
            - Impact of initialisation
            - Other drawbacks of K-M algorithm
    - Lecture 14. Clustering 2
        - 14.1. Clustering Lecture 2
        - 14.2. Limitations of the K Means Algorithm
        - 14.3. Introduction to the K-Medoids Algorithm
        - 14.4. Computational Complexity of K-Means and K-Medoids
            - The Big-O notation
            - The computational complexity of the two algorithms
        - 14.5. Determining the Number of Clusters
    - Lecture 15. Generative Models
        - 15.1. Objectives
        - 15.2. Generative vs Discriminative models
        - 15.3. Simple Multinomial Generative model
        - 15.4. Likelihood Function
        - 15.5. Maximum Likelihood Estimate
            - Max likelihood estimate for the cat/dog example
        - 15.6. MLE for Multinomial Distribution
            - The method of Lagangian multipliers for constrained optimisation
            - The constrained multinomial log-likelihood
        - 15.7. Prediction
        - 15.8. Prior, Posterior and Likelihood
        - 15.9. Gaussian Generative models
        - 15.10. MLE for Gaussian Distribution
    - Lecture 16. Mixture Models; EM algorithm
        - 16.1. Mixture Models and the Expectation Maximization (EM) Algorithm
        - 16.2. Recap of Maximum Likelihood Estimation for Multinomial and Gaussian Models
        - 16.3. Introduction to Mixture Models
            - Likelihood of Gaussian Mixture Model
        - 16.4. Mixture Model - Observed Case
        - 16.5. Mixture Model - Unobserved Case: EM Algorithm
            - Summary of Likelihood and EM algorithm
    - Homework 5
    - Project 4: Collaborative Filtering via Gaussian Mixtures
        - P4.3. Expectation–maximization algorithm
            - Data Generation Models
- Unit 05 - Reinforcement Learning
    - Lecture 17. Reinforcement Learning 1
        - 17.1. Unit 5 Overview
    - 17.2. Learning to Control: Introduction to Reinforcement Learning
        - 17.3. RL Terminology
            - Reward vs cost difference
        - 17.4. Utility Function
    - 17.5. Policy and Value Functions
        - Setting the timing in discrete time modelling
        - 17.6. Bellman Equations
    - 17.7. Value Iteration
        - Convergence
        - 17.8. Q-value Iteration
    - Lecture 18. Reinforcement Learning 2
        - 18.1. Revisiting MDP Fundamentals
        - 18.2. Estimating Inputs for RL algorithm
        - 18.3. Q value iteration by sampling
            - Exponential running average

        - Sample-based approach for Q-learning
    - 18.4. Exploration vs Exploitation
        - Epsilon-greedy
- Lecture 19: Applications: Natural Language Processing
    - 19.1. Objectives
    - 19.2. Natural Language Processing (NLP)
        - History
        - Status
    - 19.3. NLP - Parsing
    - 19.4. Why is NLP so hard
    - 19.5. NLP - Symbolic vs Statistical Approaches
    - 18.6. Word Embeddings
- Some other resources on reinforcement learning:
- Homework 6
- Project 5: Text-Based Game
    - P5.
    - P5.7. Introduction to Q-Learning with Linear Approximation
        - Intuition for the approximation of the state/action spaces
        - Q-Learning with Linear Approximation
        - Q-Learning with non-linear Approximation

[MITx 6.86x Notes Index]

Unit 00 - Course Overview, Homework 0, Project 0

Recitation 1 - Brief Review of Vectors, Planes, and Optimization

Points and Vectors

Norm of a vector

Norm: Answer the question how big is a vector

- $||X||_l \equiv l - NORM := (\sum_{i=1}^{size(X)} x_i^l)^{(1/l)}$
- Julia: `norm(x)`
- NumPy: `numpy.linalg.norm(x)`

If l is not specified, it is assumed to be 2, i.e. the Euclidian distance. It is also known as “length”, “2-norm”, “l2 norm”, …

Dot product of vector

Aka “scalar product” or “inner product”.

It has a relationship on how vectors are arranged relative to each other

- Algebraic definition: $x \cdot y \equiv x'y := \sum_{i=1}^{n} 2x_i * y_i$
- Geometric definition: $x \cdot y := ||x|| * ||y|| * cos(\theta)$ (where θ is the angle between the two vectors and $||x||$ is the 2-norm)
- Julia: `dot(x,y)`
- Numpy: `np.dot(x,y)`

Note that using the two definitions and the `arccos`, the inverse function for the cosine, you can retrieve the angle between two functions as `angle_x_y = arccos(dot(x,y)/(norm(x)*norm(y)))`.

- Julia: `angle_x_y = acos(dot(x,y)/(norm(x)*norm(y)))`

Geometric interpretation of a vector

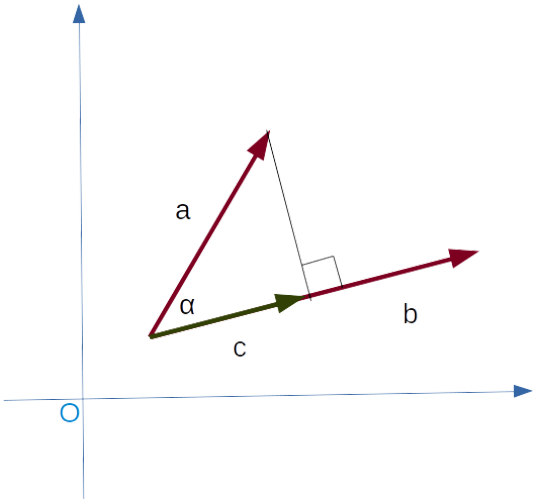
Geometrically, the elements of a vector can be seen as the coordinates of the position of arrival *compared to the position of departing*. They represent hence the *shift* from the departing point.

For example the vector [-2,0] could refer to the vector from the point (4,2) to the point (2,2) but could also represent the vector going from (6,4) to (4,4).

Vectors whose starting point is the origin are called *position vectors* and they define the coordinates in the n-space of the points where they arrive to.

Vector projections

Let’s be *a* and *b* two (not necessary unit) vectors. We want to compute the vector *c* being the projection of *a* on *b* and its l-2 norm (or length):



Let's start from the length. We know from a well-known trigonometric equation that

$||c|| = ||a|| * cos(\alpha)$, where α is the angle between the two vectors *a* and *b*:

But we also know that the dot product $a \cdot b$ is equal to $||a|| * ||b|| * cos(\alpha)$.

By substitution we find that $||c|| = \frac{a \cdot b}{||b||}$. This quantity is also called the *component* of *a* in the direction of *b*.

To find the vector *c* we now simply multiply $||c||$ by the unit vector in the direction of *b*, $\frac{b}{||b||}$, obtaining $c = \frac{a \cdot b}{||b||^2} * b$.

If *b* is already a unit vector, the above equations reduce to:

$||c|| = a \cdot b$ and $c = (a \cdot b) * b$

In Julia:

```
using LinearAlgebra
a = [4,1]
b = [2,3]
normC = dot(a,b)/norm(b)
c = (dot(a,b)/norm(b)^2) * b
```

In Python:

```
import numpy as np
a = np.array([4,1])
b = np.array([2,3])
normC = np.dot(a,b)/np.linalg.norm(b)
c = (np.dot(a,b)/np.linalg.norm(b)**2) * b
```

Planes

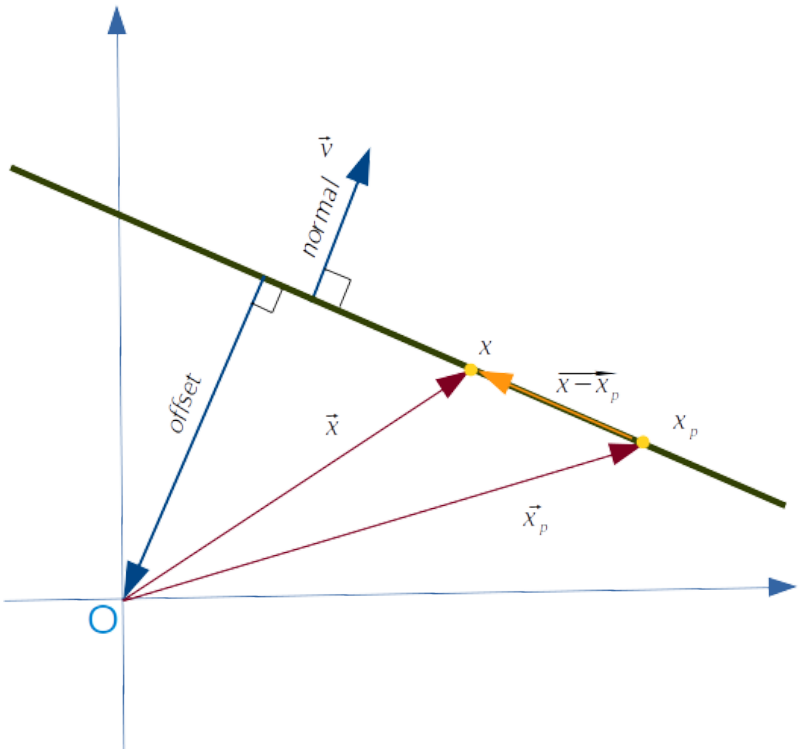
An (hyper)plane in n dimensions is any n−1 dimensional subspace defined by a linear relation. For example, in 3 dimensions, hyperplanes span 2 dimensions (and they are just called "planes") and can be defined by the vector formed by the coefficients {A,B,C,D} in the equation $Ax + By + Cz + D = 0$. Note that while the plane is unique, the vector defining it is not: in relation to the A-D coefficients, the equation is homogeneous, i.e. if we multiply all the A-D coefficients by the same number, the equation remains valid.

As hyperplanes separate the space into two sides, we can use (hyper)planes to set boundaries in classification problems, i.e. to discriminate all points on one side of the plane vs all the point on the other side.

Besides to this analytical definition, a plane can be uniquely identified also in a geometrical way starting from a point x_p on the plane and a vector $\vec{v}$ normal to the plane (not necessarily departing from the point or even from the plane):. Let's define:

- *Normal* of a plane: any n-dimensional vector perpendicular to the plane.

- *Offset of the plane with the origin*: the distance of the plan with the origin, that is the specific normal between the origin and the plane



Given a point x_p known to sit on the plane and $\overrightarrow{x_p}$ its positional vector, a generic point x and corresponding positional vector $\vec{x}$, the point x is part of the plane if and only if ("iff") the vector connecting the two points, that is $x - \overrightarrow{x_p}$, lies on the plane. In turn this is true iff such vector is orthogonal to the normal of the plane $\vec{v}$, that we can check using the dot product.

To sum up, we can define the plane as the set of all points x x such that $(x - \overrightarrow{x_p}) \cdot \vec{v} = 0$.

As from the coefficients A-D in the equation, while x_p and $\vec{v}$ unambiguously identify the plane, the converse is not true: any plane has indeed infinite points and normal vectors.

For example, let's define a plane in two dimensions passing by the point $x_p = (3, 1)$ and with norm $\vec{v} = (2, 2)$, and let's check if point $a = (1, 3)$ is on the plane. Using the above equation we find $(\vec{a} - \overrightarrow{x_p}) \cdot \vec{v} = \left(\begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 3 \\ 1 \end{bmatrix} \right) \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 0$, *a* is part of the plane.

Let's consider instead the point b = (1,4). As $(\vec{b} - \overrightarrow{x_p}) \cdot \vec{v} = \left(\begin{bmatrix} 1 \\ 4 \end{bmatrix} - \begin{bmatrix} 3 \\ 1 \end{bmatrix} \right) \cdot \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 2$, *b* is not part of the plane.

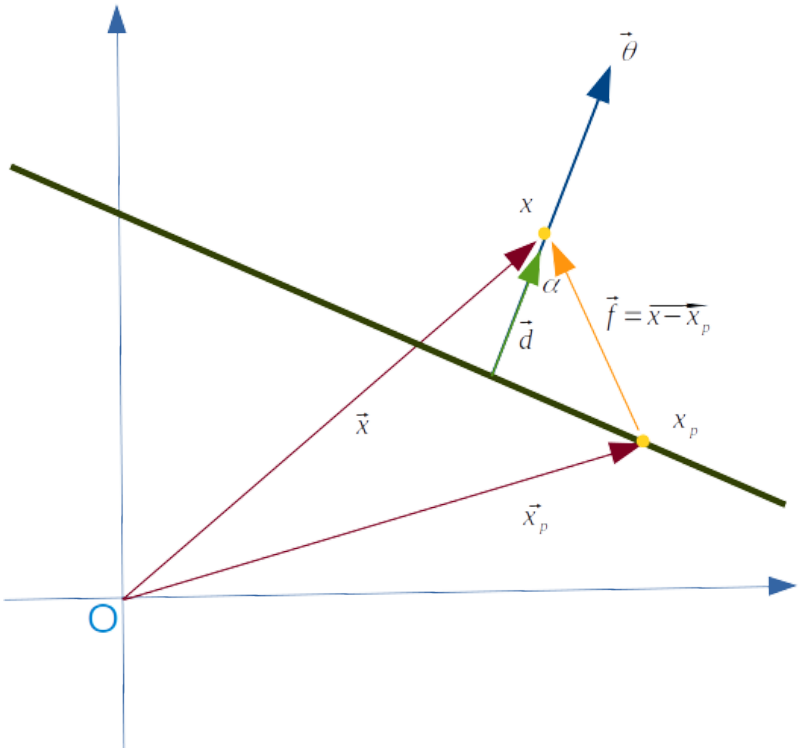
Starting from the information on the point and the normal we can retrieve the algebraic equation of the plane rewriting the equation $(x - \overrightarrow{x_p}) \cdot \vec{v} = 0$ as $\vec{x} \cdot \vec{v} - \overrightarrow{x_p} \cdot \vec{v} = 0$: the first dot product gives us the polynomial terms in the n-dimensions (often named θ), the last (negatively signed) term gives the associated offset (often named θ_0). Seen from the other side, this also means that the coefficients of the dimensions of the algebraic equation represent the normal of the plane. We can hence define our plane with just the normal and the corresponding offset (always a scalar).

Note that when θ is a unit vector (a vector whose 2-norm is equal to 1) the offset θ_0 is equal to the offset of the plane with the origin.

Using the above example ($x_p = [3, 1]$, $\vec{v} = [2, 2]$), we find that the algebraic equation of the plane is $2x_1 + 2x_2 - (3 * 2 + 1 * 2) = 0$, or, equivalently, $\frac{x_1}{\sqrt{2}} + \frac{x_2}{\sqrt{2}} - \frac{4}{\sqrt{2}} = 0$.

Distance of a point to a plane

Given a plan defined by its norm θ and the relative offset θ_0, which is the distance of a generic point x from such plane ? Let's start by calling $\vec{f}$ the vector between any point on the plan x_p and the point x, that is $\vec{f} = \vec{x} - \overrightarrow{x_p}$. The distance $\|\vec{d}\|$ of the point with the plane is then $\|\vec{d}\| = \|\vec{f}\| * cos(\alpha)$, where α is the angle between the vector $\vec{f}$ and $\vec{d}$.



But we know also that $\vec{f} \cdot \vec{\theta} = ||\vec{f}|| * ||\vec{\theta}|| * cos(\alpha)$ from the definition of the dot product.

By substitution, we find that $||\vec{d}|| = ||\vec{f}|| * \frac{\vec{f} \cdot \vec{\theta}}{||\vec{f}|| * ||\vec{\theta}||} = \frac{\vec{f} \cdot \vec{\theta}}{||\vec{\theta}||}$ (we could have arrived to the same conclusion by considering that $||\vec{d}||$ is the component of $\vec{f}$ in direction of $\vec{\theta}$).

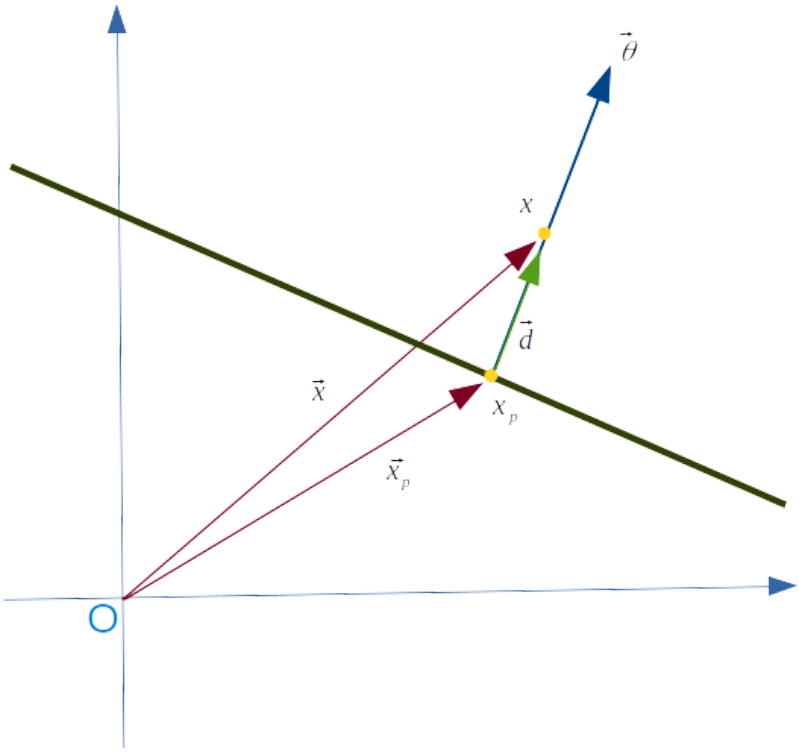
By expanding $\vec{f}$ we find that $||\vec{d}|| = \frac{(\vec{x} - \vec{x_p}) \cdot \vec{\theta}}{||\vec{\theta}||} = \frac{\vec{x} \cdot \vec{\theta} + \theta_0}{||\vec{\theta}||}$

The distance is positive when x is on the same side of the plane as $\vec{\theta}$ points and negative when x is on the opposite side.

For example the distance between the point $x = (6, 2)$ and the plane as define earlier with $\theta = (1, 1)$ and $\theta_0 = -4$ is $\frac{\vec{x} \cdot \vec{\theta} + \theta_0}{||\vec{\theta}||} = \frac{\begin{bmatrix} 6 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 4}{\| \begin{bmatrix} 1 \\ 1 \end{bmatrix} \|} = \frac{8 - 4}{\sqrt{2}} = 2 * \sqrt{2}$

Projection of a point on a plane

We can easily find the projection of a point on a plane by summing to the positional vector of the point, the vector of the distance from the point to the plan, in turn obtained multiplying the distance (as found earlier) by the *negative* of the unit vector of the normal to the plane.



Algebraically: $\overrightarrow{x_p} = \vec{x} - \frac{\vec{x}\cdot\vec{\theta}+\theta_0}{\|\vec{\theta}\|} * \frac{\vec{\theta}}{\|\vec{\theta}\|} = \vec{x} - \vec{\theta} * \frac{\vec{x}\cdot\vec{\theta}+\theta_0}{\|\vec{\theta}\|^2}$

For the example before, the point x_p is given by

$$\begin{bmatrix}6\\2\end{bmatrix} - \begin{bmatrix}1\\1\end{bmatrix} * \frac{\begin{bmatrix}6\\2\end{bmatrix}\cdot\begin{bmatrix}1\\1\end{bmatrix}-4}{2} = \begin{bmatrix}6\\2\end{bmatrix} - \begin{bmatrix}2\\2\end{bmatrix} = \begin{bmatrix}4\\0\end{bmatrix}$$

On this subject see also:

- https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/defining-a-plane-in-r3-with-a-point-and-normal-vector
- https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/point-distance-to-plane

Loss Function, Gradient Descent, and Chain Rule

Loss Function

This argument in details: segments 3.4 (binary linear classification), 5.4 (Linear regression)

The loss function, aka the *cost function* or the *race function*, is some way for us to value how far is our model from the data that we have.

We first define an “error” or “Loss”. For example in Linear Regression the “error” is the Euclidean distance between the predicted and the observed value:

$$L(x, y; \Theta) = \sum_{i=1}^{n} |\hat{y} - y| = \sum_{i=1}^{n} |\theta_1 x + \theta_2 - y|$$

The objective is to minimise the loss function by changing the parameter theta. How?

Gradient Descent

This argument in details: segments 4.4 (gradient descent in binary linear classification), 4.5 (stochastic gradient descent) and 5.5 (SGD in linear regression)

The most common iterative algorithm to find the minimum of a function is the gradient descent.

We compute the loss function with a set of initial parameter(s), we compute the gradient of the function (the derivative concerning the various parameters), and we move our parameter(s) of a small delta against the direction of the gradient at each step:

$$\hat{\theta}_{s+1} = \hat{\theta}_{s+1} - \gamma \nabla L(x, y; \theta)$$

The γ parameter is known as the *learning rate*.

- too small learning rate: we may converge very slowly or end up trapped in small local minima;
- too high learning rate: we may diverge instead of converge to the minimum

Chain rule

How to compute the gradient for complex functions.

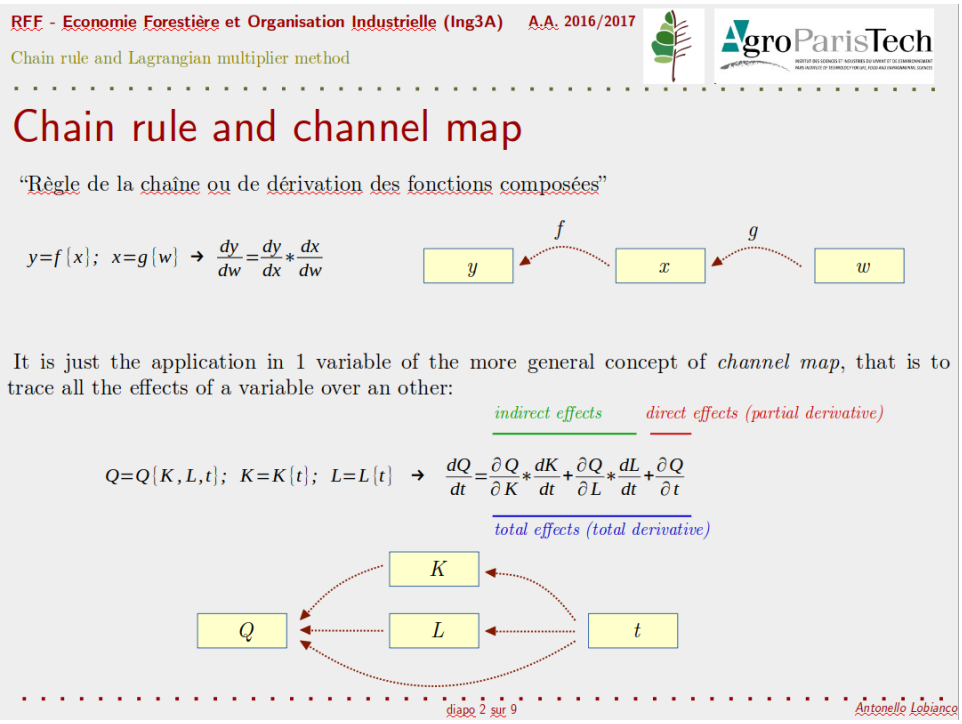
$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} * \frac{\partial z}{\partial x}$

e.g. $\hat{y} = \frac{1}{1+e^{-(\theta_1 x+\theta_2)}} = (1+e^{-(\theta_1 x+\theta_2)})^{-1}$, $\frac{\partial \hat{y}}{\partial \theta_1} = -\frac{1}{(1+e^{-(\theta_1 x+\theta_2)})^2} * e^{-(\theta_1 x+\theta_2)} * -x$

For computing derivatives one can use SymPy, a library for symbolic computation. In this case the derivative can be computed with the following script:

```
from sympy import *
x, p1, p2 = symbols('x p1 p2')
y = 1/(1+exp( - (p1*x + p2)))
dy_dp1 = diff(y,p1)
print(dy_dp1)
```

It may be useful to recognise the chain rule as an application of the *chain map*, that is tracking all the effect from one variable to the other:



Geometric progressions and series

Geometric progressions are sequence of numbers where each term after the first is found by multiplying the previous one by a fixed, non-zero number called the *common ratio*.

It results that geometric series can be wrote as $a, ar, ar^2, ar^3, ar^4, \ldots$ where r is the common ratio and the first value a is called the *scale factor*.

Geometric series are the sum of the values in a geometric progression.

Closed formula exist for both finite geometric series and, provided $|r| < 1$, for infinite ones:

- $\sum_{k=m}^{n} ar^k = \frac{a(r^m - r^{n+1}}{1-r}$ with $r \neq 1$ and $m < n$
- $\sum_{k=m}^{\infty} ar^k = \frac{ar^m}{1-r}$ with $|r| < 1$ and $m < n$.

Where m is the first element of the series that you want to consider for the summation. Typically $m = 0$, i.e. the summation considers the whole series from the scale factor onward.

For many more details on geometric progressions and series consult the relative excellent Wikipedia entry.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 01 - Linear Classifiers and Generalizations

Course Introduction

There are lots and lots of applications out there, but what's so interesting about it is that, in terms of algorithms, there is a relatively small toolkit of algorithms you need to learn to understand how these different applications can work so nicely and be integrated in our daily life.

The course covers these 3 topics

Supervised Learning

- Make predictions based on a set of data for which correct examples are given
- E.g. Attach the label "dog" to an image, based on many couple (image/dog label) already provided
- The correct answers of the examples provided to the algorithm are already given
- What to produce is given explicitly as a target
- First 3 units in increasing complexity from simple linear classification methods to more complex neural network algorithms

Unsupervised Learning

- A generalisation of supervised learning
- The target is no longer given
- "Learning" to produce output (still at the end, "make predictions"), but now just observing existing outputs, like images, or molecules
- Algorithms to learn how to generate things
- We'll talk about probabilistic models and how to generate samples, mix and match (???)

Reinforced Learning

- Suggest how to act in the world given a certain goal, how to achieve an objective optimally
- It involves making predictions, aspects of SL and UL, but also it has a goal
- Learning from the failures, how to do things better
- E.g. a robot arm trying to grasp an object
- Objective is to control a system

Interesting stuff (solving real problems) happens at the mix of these 3 categories.

Lecture 1. Introduction to Machine Learning

Slides

1.1. Unit 1 Overview

Unit 1 will cover Linear classification

Exercise: predicting whether an Amazon product review, looking at only the text, will be positive or negative.

1.2. Objectives

Introduction to Machine Learning

At the end of this lecture, you will be able to:

- understand the goal of machine learning from a movie recommender example
- understand elements of supervised learning, and the difference between the training set and the test set
- understand the difference of classification and regression - two representative kinds of supervised learning

1.3. What is Machine Learning?

Many examples: Interpretation of web search queries, movie reccomendations, digital assistants, automatic translations, image analysis… playing the go game

Machine learning as a discipline aims to design, understand, and apply computer programs that learn from experience (i.e. data) for the purpose of **modelling**, **prediction**, and **control**.

There are many ways to learn or many reasons to learn:

- You can try to model, understand how things work;
- You can try to predict about, say, future outcomes;
- Or you can try to control towards a desired output configuration.

Prediction

We will start with prediction as a core machine learning task. There are many types of predictions that we can make.:

- We can predict outcomes of *events that occur in the future* such as the market, weather tomorrow, the next word a text message user will type, or anticipate pedestrian behavior in self driving vehicles, and so on.
- We can also try to predict *properties that we do not yet know*. For example, properties of materials such as whether a chemical is soluble in water, what the object is in an image, what an English sentence translates to in Hindi, whether a product review carries positive sentiment, and so on.

1.4. Introduction to Supervised Learning

Common to all these “prediction problems" mentioned previously is that they would be very hard to solve in a traditional engineering way, where we specify rules or solutions directly to the problem. It is far easier to provide examples of correct behavior. For example, how would you encode rules for translation, or image classification? It is much easier to provide large numbers of translated sentences, or examples of what the objects are on a large set of images. I don’t have to specify the solution method to be able to illustrate the task implicitly through examples. The ability to learn the solution from examples is what has made machine learning so popular and pervasive.

We will start with supervised learning in this course. In supervised learning, we are given an example (e.g. an image) along with a target (e.g. what object is in the image), and the goal of the machine learning algorithm is to find out how to produce the target from the example.

More specifically, in supervised learning, we hypothesize a collection of functions (or mappings) parametrized by a parameter (actually a large set of parameters), from the examples (e.g. the images) to the targets (e.g. the objects in the images). The machine learning algorithm then automates the process of finding the parameter of the function that fits with the example-target pairs the best.

We will automate this process of finding these parameters, and also specifying what the mappings are.

So this is what the machine learning algorithms do for you. You can then apply the same approach in different contexts.

1.5. A Concrete Example of a Supervised Learning Task

Let’s consider a movie recommender problem. I have a set of movies I’ve already seen (the **training set**), and I was to use the experience from those movies to make recommendations of whether I would like to see tens of thousands of other movies (the **test set**).

We will compile a **feature vector** (a binary list of its characteristics… genre, director, year…) for each movie in the training set as well for those I want to test (we will denote these feature vectors as x).

I also give my recommendation (again binary) if I want to see again or not the films in the training set.

Now I have the training set (feature vectors with associated labels), but I really wish to solve the task over the test set. It is this discrepancy that makes the learning task interesting. I wish to generalize what I extract from the training set and apply that information to the test set.

More specifically, I wish to learn the mapping from x to labels (plus minus one) on the basis of the training set, and hope and guarantee that that mapping, if applied now in the same way to the test

examples, it would work well.

I want to predict the films in the test set based on their characteristics and my previous recommendation on the training set.

1.6. Introduction to Classifiers: Let's bring in some geometry!

On the basis of the training set, pairs of feature vectors associated with labels, I need to learn to map each each x, each future vector, to a corresponding label, which is a plus or minus 1.

What we need is a **classifier** (here denoted with h) that maps from points to corresponding labels.

So what the classifier does is divide the space into essentially two halves – one that's labeled 1, and the other one that's labeled minus 1. –> a "linear classifier" is a classifier that perform this division of the full space in the two half linearly

We need to now, somehow, evaluate how good the classifier is in relation to the training examples that we have. To this end, we define something called **training error** (here denoted with ϵ) It has a subscript n that refers to the number of training examples that I have. And I apply it to a particular classifier. I evaluate a particular classifier, in relation to the training examples that I have.

$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$

(The double brackets denotes a function that takes a comparison expression inside and returns 1 if the expression is true, 0 otherwise. It is basically an indicator function.)

In the second example of classifier given in the lesson, the training error is 0. So, according to just the training examples, this seems like a good classifier.

The fact that the training error is zero however doesn't mean that the classifier is perfect!

Let's now consider a non-linear classifier.

We can build an "extreme" classifier that accept as +1 only a very narrow area around the +1 training set points.

As a result, this classifier would still have training error equal to zero, but it would classify all the points in the test set, no matter how much close to the +1 training set points they are, as -1 !

what is going on here is an issue called **generalization** --how well the classifier that we train on the training set generalizes or applies correctly, similarly to the test examples, as well? This is at the heart of machine-learning problems, the ability to generalize from the training set to the test set.

The problem here is that, in allowing these kind of classifiers that wrap themselves just around the examples (in the sense of allowing any kind of non-linear classifier), we are making the **hypothesis class**, the set of possible classifiers that we are considering, too large. We improve generalization, we generalize well, when we only have a very limited set of classifiers. And, out of those, we can find one that works well on the training set.

The more complex set of classifiers we consider, the less well we are likely to generalize (this is the same concept as overfitting in a statistical context).

We would wish to, in general, solve these problems by finding a small set of possibilities that work well on the training set, so as to generalize well on the test set.

Training data can be graphically depicted on a (hyper)plane. Classifiers are mappings that take feature vectors as input and produce labels as output. A common kind of classifier is the linear classifier, which linearly divides space (the hyperplane where training data lies) into two. Given a point x in the space, the classifier h outputs $h(x) = 1$ or $h(x) = -1$, depending on where the point x exists in among the two linearly divided spaces.

Each classifier represents a possible "hypothesis" about the data; thus, the set of possible classifiers can be seen as the space of possible hypothesis

1.7. Different Kinds of Supervised Learning: classification vs regression

A supervised learning task is one where you have specified the correct behaviour. Examples are then feature vector and what you want it to be associated with. So you give "supervision" of what the correct behaviour is (from which the term).

Types of supervised learning:

- Multi-way classification : $h : \mathrm{X} \rightarrow$ finite set
- Regression: $h : \mathrm{X} \rightarrow R$
- Structured prediction: $h : \mathrm{X} \rightarrow$ structured object, like a language sentence

Types of machine learning:

- Supervised learning
- Unsupervised learning: You can observe, but the task itself is not well-defined. The problem there is to model, to find the irregularities in how those examples vary.
- Semi-supervised learning
- Active learning (the algorithm asks for useful additional examples)
- Transfer learning
- Reinforcement learning

The training set is illustration of the task, the test set is what we wish to really do well on. But those are examples that we don't have available at the time we need to select the mapping.

Classification maps feature vectors to categories. The number of categories need not be two - they can be as many as needed. Regression maps feature vectors to real numbers. There are other kinds of supervised learning as well.

Fully labelled training and test examples corresponds to supervised learning. Limited annotation is semi-supervised learning, and no annotation is unsupervised learning. Using knowledge from one task on another task means you're "transferring" information. Learning how to navigate a robot means learning to act and optimize your actions, or reinforcement learning. Deciding which examples are needed to learn is the definition of active learning.

Lecture 2. Linear Classifier and Perceptron Algorithm

Slides

2.1. Objectives

At the end of this lecture, you will be able to

- understand the concepts of Feature vectors and labels, Training set and Test set, Classifier, Training error, Test error, and the Set of classifiers
- derive the mathematical presentation of linear classifiers
- understand the intuitive and formal definition of linear separation
- use the perceptron algorithm with and without offset

2.2. Review of Basic Concepts

A supervised learning task is when you are given the input and the corresponding output that you want, and you're supposed to learn irregularity between the two in order to make predictions for future examples, or inputs, or feature vectors x.

Test error is defined exactly similarly over the test examples. So defined similarly to the training error, but over a disjoint set of examples, those future examples that you actually wish to do well.

We typically drop the n on it, assuming that the test set is relatively large,

Much of machine learning, really, the theory part is in relating how a classifier that might do well on the training set would also do well on the test set. That's the problem called Generalization, as we've already seen. We can effect generalization by limiting the choices that we have at the time of considering minimizing the training error.

So our classifier here belongs to a **set of classifiers** (here denoted with H), that's not the set of all mappings, but it's a limited set of options that we constrain ourselves to.

The trick here is to somehow guide the selection of the classifier based on the training example, such that it would do well on the examples that we have not yet seen.

In order for this to be possible at all, you have to have some relationship between the training samples and the test examples. Typically, it is assumed that both sets are samples from some large collection of examples as a random subset. So you get a random subset as a training set.

What is this lecture going to be about ?

This lecture we will consider only linear classifiers, such that we keep the problem well generalised (we will return to the question of generalization more formally later on in this course).

We're going to formally define the set of linear classifiers, the set H restricted set of classifiers. We need to introduce parameters that index classifiers in this set so that we can search over the possible classifiers in the set.

We'll see what's the limitation of using linear classifiers.

We'll next need to define the learning algorithm that takes in the training set and the set of classifiers and tries to find a classifier, in that set, that somehow best fits the training set.

We will consider initially the **perceptron algorithm**, which is a very simple online mistake driven algorithm that is still useful as it can be generalized to high dimensional problems. So perceptron algorithm finds a classifier $\hat{h}$, where hat denotes an estimate from the data. It's an algorithm that takes, as an input, the training set and the set of classifiers and then returns that estimated classifier,

2.3. Linear Classifiers Mathematically Revisited

The dividing line here is also called **decision boundary**:

- If x was a one-dimensional quantity, the decision boundary would be a point.
- In 2D, that decision boundary is a line.
- In 3D, it would be a plane.
- And in higher dimensions, it's called hyperplane that divides the space into two halves.

So now we need to parameterize the linear classifiers, so that we can effectively search for the right one given the training set.

Through the origin classifiers

Definition of a decision boundary through the origin:

All points x such that $x : g_1 * x_1 + g_2 * x_2 = 0$ or, calling θ the vector of coefficients and x the vector of the points, $x : \theta \cdot x = 0$.

Note that this means that the vector θ is ortoghonal to the positional vectors of the points in the boundary.

The classifier is now parametrised by theta: $h(x;\theta)$ So each choice of theta defines one classifier. You change theta, you get a different classifier. It's oriented differently. But it also goes through origin.

Our classifier become: $h(x;\theta) = sign(\theta \cdot x)$

Note that this association between the classifier and the parameter vector theta is not unique. There are multiple parameter vectors theta that defined exactly the same classifier. The norm of the parameter vector of theta is not relevant in terms of the decision boundary.

General linear classifiers

Decision boundary: $x : \theta \cdot x + \theta_0 = 0$.

Our classifier becomes: $h(x;\theta,\theta_0) = sign(\theta \cdot x + \theta_0)$

2.4. Linear Separation

A training set is said to be **linearly separable** if it exists a linear classifier that correctly classifies all the training examples (i.e. if parameter $\hat{\theta}$ and $\hat{\theta}_0$ exists such that $y^i * (\hat{\theta} \cdot x^i + \hat{\theta}_0) > 0 \ \forall i = 1, \ldots, n$).

Given θ and θ_0, a linear classifier $h : X \to \{-1, 0, +1\}$ is a function that outputs $+1$ if $\theta \cdot x + \theta_0$ is positive, 0 if it is zero, and -1 if it is negative. In other words, $h(x) = sign(\theta \cdot x + \theta_0)$.

2.5. The Perceptron Algorithm

When the output of a classifier is exactly 0, if the example lies exactly on the linear boundary, we count that as an error, since we don't know which way we should really classify that point.

Training error for a linear classifier:

$\epsilon_n(h) = \sum_{i=1}^{n} \frac{[\![h(x^i) \neq y^i]\!]}{n}$

becomes:

$\epsilon_n(\theta, \theta_0) = \sum_{i=1}^{n} \frac{[\![y^i * (\theta \cdot x^i + \theta_0) \leq 0]\!]}{n}$

We can now turn to the problem of actually finding a linear classifier that agrees with the training examples to the extent possible.

Through the origin classifier

- We start with a $\theta = 0$ (vector) parameter
- We check if, with this parameter, the classifier make an error
- If so, we progressively update the classifier

The update function is $\theta^i = \theta^{i-1} + y^i * x^i$.

As we start with $\theta^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, the first attempt is always leading to an error and to a first "update" that will be $\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + y^1 * x^1$.

For example, given the pair $(x^1 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, y^1 = -1)$, θ_1 becomes $\theta^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + -1 * \begin{bmatrix} 2 \\ 4 \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \end{bmatrix}$

Its error is then $\epsilon_n(\theta^1) = [\![y^i * (\theta^1 \cdot x^i)) \leq 0]\!] = [\![(y^i)^2 * ||x^i||^2) \leq 0]\!] = 0$

Now, what are we going to do here over the whole training set, is we start with the 0 parameter vector and then go over all the training examples. And if the i-th example is a mistake, then we perform that update that we just discussed.

So we'd nudge the parameters in the right direction, based on an individual update. Now, since the different training examples might update the parameters in different directions, it is possible that the later updates actually make the earlier undo some of the earlier updates, and some of the earlier examples are no longer correctly classified. In other words, there may be cases where the perceptron algorithm needs to go over the training set multiple times before a separable solution is found.

So we have to go through the training set here multiple times, here capital T times go through the training set, either in order or select at random, look at whether it's a mistake, and perform a simple update.

- function perceptron $\left(\left\{ (x^{(i)}, y^{(i)}), i = 1, \ldots, n \right\}, T \right)$:
  - initialize $\theta = 0$ (vector);
  - for $t = 1, \ldots, T$ do
    - for $i = 1, \ldots, n$ do
      - if $y^{(i)}(\theta \cdot x^{(i)}) \leq 0$ then
        - update $\theta = \theta + y^{(i)} x^{(i)}$
  - return θ

So the perceptron algorithm takes two parameters: the training set of data (pairs feature vectors => label) and the T parameter that tells you how many times you try to go over the training set.

Now, this classifier for a sufficiently large T, if there exists a linear classifier through origin that correctly classifies the training samples, this simple algorithm actually will find a solution to that problem. There are many solutions typically, but this will find one. And note that the one found is not, generally, some "optimal" one, where the points are "best" separated, just one where the points *are* separated.

Generalised linear classifier

We can generalize the perceptron algorithm to run with the general class of linear classifiers with the offset parameter. The only difference here is that now we initialize the parameter back to 0, as well as

the scalar to 0, that we consider also θ_0 in the error check, that we update θ_0 as well and that we return it together with θ:

- function perceptron $\left(\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}, T\right)$:
  - initialize $\theta = 0$ (vector), $\theta_0 = 0$ (scalar);
  - for $t = 1, \ldots, T$ do
    - for $i = 1, \ldots, n$ do
      - if $y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0$ then
        - update $\theta = \theta + y^{(i)} x^{(i)}$
        - update $\theta_0 = \theta_0 + y^{(i)}$
  - return θ, θ_0

The update of θ_0 can be seen as the update of a linear model trough the origin, where the offset is a further dimension of the data:

The model $\theta \cdot x + \theta_0$ can be then see equivalently as $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$.

The update function $\theta = \theta + x^i * y^i$ becomes then $\begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} = \begin{bmatrix} \theta \\ \theta_0 \end{bmatrix} + y^i * \begin{bmatrix} x \\ 1 \end{bmatrix}$ from which, going back to our original model, we obtain the given update functions.

So now we have a general learning algorithm. The simplest one, but it can be generalized to be quite powerful and therefore, hence it is a useful algorithm to understand.

For example, we saw here a binary classification, but it can be easily be extended to a multiclass classification employing a “one vs all” strategy, as explained in this SO answer or on wikipedia.

Code implementation: functions `perceptron_single_step_update()` , `perceptron()` , `average_perceptron()` , `pegasos_single_step_update()` *and* `pegasos()` *in project1.*

Lecture 3 Hinge loss, Margin boundaries and Regularization

Slides

3.1. Objective

Hinge loss, Margin boundaries, and Regularization

At the end of this lecture, you will be able to

- understand the need for maximizing the margin
- pose linear classification as an optimization problem
- understand hinge loss, margin boundaries and regularization

3.2. Introduction

Today, we will talk about how to turn machine learning problems into optimization problems. That is, we are going to turn the problem of finding a linear classifier on the basis of the training set into an optimization problem that can be solved in many ways. Today’s lesson we will talk about what linear large margin classification is and introduce notions such as margin loss and regularization.

Why optimisation ?

The perceptron algorithm choose a correct classifier, but there are many possible correct classifiers. Between them, we would favor the solution that somehow is drawn between the two sets of training examples, leaving lots of space on both sides before hitting the training examples. This is known as a **large margin classifier**.

A large margin linear classifier still correctly classifies all of the test examples, even if these are noisy samples of unknown true values. A large margin classifier in this sense is more robust against noises in the examples. In general, large margin means good generalization performance on test data.

How to find such large margin classifier? --> optimisation problem

We consider the parallel plane to the classifier passing by the closest positive and negative point as respectively positive and negative **margin boundaries**, and we want them to be equidistant from the

classifier.

The goal here is now to use these margin boundaries essentially to define a fat decision boundary that we will still try to fit with the training examples. So we will push these margin boundaries apart that will force us to reorient there and move that system boundary into a place that carves out this large empty space between the two sets of examples.

We will minimise an objective function made of two terms, the **regularisation term**, that push to set the boundaries as much apart as possible, but also a **loss function term**, that gives us the penalty from points that now, as the boundaries extend, got trapped inside the margin or even more to the other part (becoming misclassified)

3.3. Margin Boundary

The first thing that we must do is to define what exactly the margin boundaries are and how we can control them, how far they are from the decision boundary. Remember that they are equidistant from the decision boundary.

Given the linear equation $\theta \cdot x + \theta_0 = k$, the decision boundary is the set of points where $k = 0$, the positive margin boundary is the set of points where k is some positive constant and the negative margin boundary is where k is some negative constant.

The equation of the decision boundary can be wrote in normalised version by dividing it by the norm of the θ vector:

$$\frac{\theta}{||\theta||} \cdot x + \frac{\theta_0}{||\theta||} = \frac{k}{||\theta||}$$

3.4. Hinge Loss and Objective Function

The objective is to maximise the distance of the decision boundary from the marginal boundaries, $k/||\theta||$ for the choosen k (plus/minus 1 in the lesson). This is equivalent to minimise $||\theta||$, in turn equivalent to minimize $\frac{||\theta||^2}{2}$

We can define the **Hinge loss** function as a function of the amount of *agreement* (here denoted with z) between the classifier score and the data given by an equation we already saw: $y^i * (\theta \cdot x^i + \theta_0)$.

Previously, in the perceptron algorithm, we used the indicator function of such agreement in the definition of the error ϵ.

Now we use its value, as the loss function is defined as:

$loss(z) = 0$ if $z \geq 1$ (the point is outside the boundary, on the right direction) and $loss(z) = 1 - z$ if $z < 1$ (either the point is well classified, but inside the boundary - $0 \leq z < 1$ - or it is even misclassified - $z \leq 0$).

Note that while for a point on the decision boundary, being exactly on the decision boundary implies a misclassification, being on the (right) margin boundary implies a zero loss.

A bit of terminology

- **Support vectors** are the data points that lie closest to the decision surface (or hyperplane). They are the data points most difficult to classify but they have direct bearing on the optimum location of the decision surface.
- **Support-vector machines** (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (SVMs can perform also non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces).
- In mathematical optimization and decision theory, a **Loss function** or cost function is a function that maps an event or values of one or more variables onto a real number intuitively representing some “cost” associated with the event. An optimization problem seeks to minimize a loss function. An objective function is either a loss function or its negative (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case it is to be maximized.

- The **Hinge loss** is a specific loss function used for training classifiers. The hinge loss is used for "maximum-margin" classification, most notably for support vector machines (SVMs). For an intended output $y = \pm 1$ and a classifier score s, the hinge loss of the prediction s is defined as $\ell(s) = \max(0, 1 - y \cdot s)$. Note that s should be the "raw" output of the classifier's decision function, not the predicted class label. For instance, in linear SVMs, $s = \theta \cdot \mathbf{x} + \theta_0$, where $\mathbf{x}$ is the input variable(s). When y and s have the same sign (meaning s predicts the right class) and $|s| \geq 1$, the hinge loss $\ell(s) = 0$. When they have opposite signs, $\ell(s)$ increases linearly with s, and similarly if $|s| < 1$, even if it has the same sign (correct prediction, but not by enough margin).

We are now ready to define the objective function (to minimise) as:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$$

Now, our objective function how we wish to guide the solution is a balance between the two. And we set the balance by defining a new parameter called **regularization parameter** (in the previous equation denoted by λ) that simply weighs how these two terms should affect our solution. Regularization parameter here is always greater than 0.

Greater λ : we will favor large margin solutions but potentially at a cost of incurring some further loss as the margin boundaries push past the examples.

Optimal value of θ and θ_0 is obtained by minimizing this objective function. So we have turned the learning problem into an optimization problem.

Code implementation: functions `hinge_loss_single()` , `hinge_loss_full()` *in project1.*

Homework 1

1. Perceptron Mistakes

1. (e) Perceptron Mistake Bounds

Novikoff Theorem (1962): https://arxiv.org/pdf/1305.0208.pdf

Assumptions:

- There exists an optimal θ^* such that $\frac{y^{(i)}(\theta^* x^{(i)})}{\|\theta^*\|} \geq \gamma \forall i = 1, \ldots, n$ and some $\gamma > 0$ (the data is separable by a linear classifier through the origin)
- All training data set examples are bounded by being $\|x^{(i)}\| \leq R, i = 1, \cdots, n$

Predicate:

Then the number of k updates of the perceptron algorithm is bounded by $k < \frac{R^2}{\gamma^2}$

Lecture 4. Linear Classification and Generalization

Slides

4.1. Objectives

At the end of this lecture, you will be able to:

- understanding optimization view of learning
- apply optimization algorithms such as gradient descent, stochastic gradient descent, and quadratic program

4.2. Review and the Lambda parameter

Last time (lecture 3), we talked about how to formulate maximum margin linear classification as an optimization problem. Today, we're going to try to understand the solutions to that optimization problem and how to find those solutions.

The objective function to minimise is still

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2$$

where the first term is the average loss and the second one is the regularisation.

The average loss is what we are trying to minimize.

Minimizing the regularization term will try instead to push the margin boundaries further and further apart.

And the balance between these two are controlled by the regularization parameter lambda.

Minimising the regularisation term $\frac{\lambda}{2} \mid\mid \theta \mid\mid^2$ we maximise the distance of the margin boundary $\frac{1}{||\theta||}$.

And as we do that, we start hitting the actual training examples. The boundary needs to orient itself, and we may start incurring losses.

The role of the lambda parameter

As we change the regularization parameter, lambda, we change the balance between these two terms. The larger the value lambda is, the more we try to push the margin boundaries apart; the smaller it is, the more emphasis we put on minimizing the average laws on the training example, reducing the distance of the margin boundaries up to the extreme $\lambda = 0$ where the margin boundaries themselves collapse towards the actual decision boundary.

The further and further the margin boundaries are posed, the more the solution will be guided by the bulk of the points, rather than just the points that are right close to the decision boundary. So the solution changes as we change the regularization parameter.

In other terms:

- $\lambda = 0$: if we consider a minimisation of only the loss function, without the margin boundaries, (admitting a separable problem) we would have infinite solutions within the right classifiers (similarly to the perceptron algorithm);
- $\lambda = \epsilon^+$: With a very small (but positive) λ, the margin boundaries would be immediately pushed to the first point(s) at the minimal distance to the classifier (aka the “support vectors”, from which the name of this learning method), and only this point(s) would drive the “optimal” classifier within all the possible ones (we overfit relatively to these boundary points);
- $\lambda = \text{large}^+$: As we increase λ, we are starting to give weight also to the points that are further apart, and the optimal classifier may be changing as a consequence, eventually even misclassifying some of the closest points, if there are enough far points pushing for such classifier.

4.3. Regularization and Generalization

We minimise a new function $J/\lambda = \frac{1}{\lambda} * \frac{1}{n} \sum_{i=1}^{n} \text{Loss}_h(\cdot) + \frac{1}{2} \mid\mid \theta \mid\mid^2$ (as λ is positive minimising the new function is equal to minimising the old one).

We can think the objective function as function of the $1/\lambda$ ratio (that we denote as “c”):

- $1/\lambda$ small -> wide margins, high losses
- $1/\lambda$ big -> narrow margins, small/zero losses

Such function, once minimised, will result in optimal classifiers $h(\theta^*, \theta_0^*, c)$ from which I can compute the average loss. Such average loss would be a convex, monotonically decreasing function of c, although it would not reach zero losses even for very large values of c if the problem is not linearly separable.

If we could measure also the test loss (from the test set) obtained from such optimal classifiers, we would find a typical u-shape where test loss first decreases with c, but then would growth up again. Also, the test loss (or “error”) would be always higher than the training loss, as we are fitting from the training set, not the test set.

We call c^* the “optimal” c value that minimise the test error. If we could find it, we would have found the value of λ that best generalise the method.

Now, we cannot do that, but we will talk about how to approximately do that.

With c below c^* I am underfitting, with c above c^* I am overfitting (around the point(s) closest to the decision margin). How to find c^* ? By dividing my training set in a “new” training set, and a (smaller) **validation set**, our new “pretending” test set.

Using this “new” training set we can find the optimal classifier as function of c, and at that point use $h(\theta^*, \theta_0^*, c)$ to evaluate the **validation loss** (or “validation error”) to numerically find an approximate value of c^* .

4.4. Gradient Descent

Objective: So far we have seen how to qualitatively understand the type of solutions that we get when we vary the regularization parameter and optimize with respect to theta and theta naught. Now, we are going to talk about, actually, algorithms for finding those solutions.

Gradient descent algorithm to find the minimum of a function $J(\theta)$ with respect to a parameter $\theta \in \mathbb{R}$ (for simplicity).

- I start with a given θ_{start}
- I evaluate the derivative $\partial J / \partial \theta$ at θ_{start}, i.e. the slope of the function in θ_{start}. If this is positive, increasing θ will increase the J function and the opposite if it is negative. In both cases, if I move θ in the opposite direction of the slope, I move toward a lower value of the function.
- My new, updated value of θ is then $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$ where η is known as the **step size** or **learning rate**, and the whole equation as **gradient descent update rule**.
- We stop when the variation in θ goes below a certain threshold.

If η is too small: we may converge very slowly or end up trapped in small local minima;

If η is too small: we may diverge instead of converge to the minimum (going to the opposite direction respect to the minimum).

If the function is strictly convex, then there would be a constant learning rate small enough that I am guaranteed quickly to get to the minimum of that function.

Multi-dimensional case

The multi-dimensional case is similar, we update each parameter with respect to the corresponding partial derivative (i.e. the corresponding entry in the gradient):

$\theta_j = \theta_j - \eta \frac{\partial J}{\partial \theta_j}$ or, in vector form, $\theta = \theta - \eta \nabla J(\theta)$

(where $\nabla J(\theta_\mathbf{j})$ is the gradient of J, i.e. the column vector concatenating the partial derivatives with respect to the various parameters)

4.5. Stochastic Gradient Descent

Let firs note that we can write $\frac{1}{n} \sum_{i=1}^{n} (a_i) + b$ as $\frac{1}{n} \sum_{i=1}^{n} (a_i + b)$.

Our original objective function can hence be written as

$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2 \right]$

Let’s also, for now, simplify the calculation omitting the offset parameter:

$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{Loss}_h(y^{(i)} * (\theta \cdot x^{(i)})) + \frac{\lambda}{2} \mid\mid \theta \mid\mid^2 \right]$

We can now see the above objective function as an *expectation* of the function $J(\theta)$: $E[J(\theta)] = \frac{1}{n} \sum_{i=1}^{n} J[\theta]$ where t

We can then use a stochastic approach (SGD, standing for **Stochastic Gradient Descen**), where we sample at random from the training set a single data pair i, we compute the gradient of the objective function with respect to that single parameter $\frac{\partial J_i(\theta)}{\partial \theta}$, we compute an update of theta following the same rule as before ($\theta = \theta - \eta_t \nabla J_i(\theta)$), we sample an other data pair j, we re-compute the gradient with respect to this new pair, we update again ($\theta = \theta - \eta_t \nabla J_j(\theta)$), and so on.

This has the advantage to be more efficient than taking all the data, compute the objective function and the gradients with respect to all the data points (could be milions!), and perform the gradient descent with respect to such function.

By sampling we introduce however some stochasticity and noises, and hence we need to put care on the learning parameter that need to be reduced to avoid to diverge.

The new learning rate η needs to:

- Be lower than the one we would employ for the deterministic gradient descent algorithm in order to reduce the variance from the stochasticity that we're introducing:
  - reduce at each update t with the limit as the steps go to infinite to go to zero $\lim_{t\to\infty}\eta_t = 0$
  - be such that are "square summable", i.e. the sum of the square values must be finite, $\sum_{t=1}^{\infty}\eta_t^2 < \infty$
- But at the same time retain enough weight that we can reach the minimum wherever it is:
  - if we sum at the infinite the learning rates we must be able to reach infinity, i.e. $\sum_{t=1}^{\infty}\eta_t = \infty$

One possible learning rate that satisfies all the above constraints is $\eta_t = \frac{1}{1+t}$.

Which is the gradient of the objective function with respect to data pair i ?

$$\nabla J_i(\theta) = \begin{cases} 0 & \text{when loss=0} \\ -y^i\mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda\theta$$

The update rule for the i data pair is hence:

$$\theta = \theta - \eta_t * \left(\begin{cases} 0 & \text{when loss=0} \\ -y^i\mathbf{x}^{(i)} & \text{when loss >0} \end{cases} + \lambda\theta \right)$$

There are three differences between the update in the stochastic gradient descent method and the update in our earlier perception algorithm ($\theta = \theta + y^{(i)}\mathbf{x}^{(i)}$):

- First, is that we are actually using a decreasing learning rate due to the stochasticity.
- The second difference is that we are actually performing the update, even if we are correctly classifying the example, because the regularization term will always yield an update, regardless of what the loss is. And the role of that regularization term is to nudge the parameters a little bit backwards, so decrease the norm of the parameter vector at every stamp, which corresponds to trying to maximize the margin.
- To counterbalance that, we will get a non-zero derivative from the last terms, if the loss is non-zero. And that update looks like the perceptor update, but it is actually made even if we correctly classify the example. If the example is within the margin boundaries, you would get a non-zero loss.

An other point of view on SGD compared to a deterministic approach:

The original concern for using SGD is due to computational constraint: it is inefficient to compute the gradients over all the samples and update the model, think about you have more than a million training samples, and you can only make progress after computing all gradients. Thus, instead of compute the exact value of the gradient over the entire training data set, we randomly sample a fraction of them to estimate the true gradient. You see, there is a trade-off controlled by the batch size (number of samples used in each SGD updates), large batch size is more accurate but slower, and it need to be tuned in practice.

Nowadays, researchers start to understand SGD from a different perspective. The noise caused by SGD could make the trained more robust, (high level idea, noisy updates will make you converge to a flatter minima), which results in a better generalization performance. Thus, SGD could also be viewed as a way of regularization, in that sense, we may say it is better compared to gradient descent.

4.6. The Realizable Case - Quadratic program

Let's consider a simple case where (a) the problem is separable and (b) we do not allow any error, i.e. we want the loss part of the objective function to be zero (that's the meaning of "realisable case": we set as constraint that the loss function must return zero).

The problem becomes:

$$\min_{(\theta,\theta_0)} \frac{1}{2}||\theta||^2$$

Subject to:

$$y^{(i)} * (\theta \cdot \mathbf{x}^{(i)} + \theta_0) \geq 1 \ \ \forall i = 1, \ldots, n$$

We want hence extend the margins as much as possible while keeping zero losses, that is until the first data pair(s) is reach (we can't go over them otherwise we would start encoring a loss). Note that

this is exactly equivalent to the minimisation of our original $J(\theta)$ function when λ is very small but not zero. There the zero loss output (in case of a separable problem) is the result of the minimisation of the loss function, here is it imposed as a constraint.

This is a problem quadratic in the objective and with linear constraints, and it can be solved with so-called quadratic solvers.

Relaxing the loss constraint and incorporate the loss in the objective would still leave the problem as a quadratic one.

This case is also called **maximum margin separator** or **hard margin SVM** to distinguish it from the **soft margin SVM** we saw earlier where, even in cases of separable problems, we allow to trade some misclassifications for a better regularisation.

Homework 2

Project 1: Automatic Review Analyzer

Recitation 1: Tuning the Regularization Hyperparameter by Cross Validation and a Demonstration

Outline

Supervised learning: Importance to have a model Objective: classification or regression

Cross validation: using the validation set to find the optimal parameters of our learning algorithm.

Supervised Learning

obj: derive a function f_{est} that approximate the real function f that link an input x to an output y.

We started by sampling from X and corresponding y --> training dataset We call the relation between this sampled pairs f_{data}

Our first obj is then to find a f_{est}, parametrised by some parameters (θ, θ_0), that approximate *this* f_{data}.

In order to find (θ, θ_0), we use an optimisation approach that minimise the function $J()$ made of a "loss" term $L(\theta, \theta_0; x, y)$ that describes the difference between our f_{est} and f_{data}.

But $f_{data} \neq f$. Because of that we add to our loss function a regularisation term $R(\theta)$ that constraints f_{est} not to be too similar to f_{data} that then is no longer able to generalise to f.

A **hyperparameter** α then balance these two terms in our function to minimise. And we remain with the task to choose this parameter, as it is not determined by the minimisation of the function as it is for (θ, θ_0). We'll see that to find α we will use indeed the cross validation using training data only and how choosing alpha will affect the performance of the model.

We'll go trough cross the method of cross validation which this is actually how we find α in practice using training data only, not the test data.

Support Vector Machine

What are the Loss and the regularisation functions for Support Vector Machines (SVM)

SVM obj: maximise the margins of the decision boundary

Distance from point i to the decision boundary:

$\gamma = \frac{y^i * (\theta \cdot x^i + \theta_0)}{||\theta||}$ (note that this is a distance with positive sign if the point side match the label side, negative otherwise)

The margin d is the minimal distance of any point with the decision boundary:

$d = \min_i \gamma(x^i, y^i, \theta, \theta_0)$

Large margin --> more ability of our model to generalise

Loss term

For SVM the loss term is the hinge loss L_h:

$$L_h = f\left(\frac{\gamma}{\gamma_{ref}}\right) = \begin{cases} 1 - \frac{\gamma}{\gamma_{ref}} & \text{when} \gamma < \gamma_{ref} \\ 0 & \text{otherwise} \end{cases}$$

As a function of γ, the hinge loss function for each point is above 1 for negative gammas, is 1 for $\gamma = 1$, continue to linearly decrease up to reaching 0 when $\gamma = \gamma_{ref}$ and then remains zero, where γ_{ref} is indeed the margin d as previously defined.

Regularisation term

The goal of the regularisation term is to make the model more generalisable.

For that, we want to maximise the margin, hence γ_{ref}. As we have a minimisation problem, that is equivalent to minimise $1/\gamma_{ref}$ or (as did in practice) $1/\gamma_{ref}^2$.

Objective function

$J = \frac{1}{n}\sum_{i=1}^{n} L_h(\frac{\gamma_i}{\gamma_{ref}}) + \alpha * \frac{1}{\gamma_{ref}^2}$

We now are left to define what γ_{ref} is in terms of (θ, θ_0).

Maximum margin

Note that we can scale θ by any constant (i.e. change it's magnitude) without changing the decision boundary (obviously also θ_0 will be rescaled).

In particular, we can scale θ such that $y^m(\theta \cdot x^m + \theta_z ero) = 1$, where m denotes the point at the minimum distance with the decision boundary.

In such case θ_{ref} simply becomes $\frac{1}{||\theta||}.

The objective function becomes the one we saw in the lecture:

$J = \frac{1}{n}\sum_{i=1}^{n} L_h(\$y^i(\theta \cdot x^i + \theta_0)) + \alpha * ||\theta||^2$

Note that the regularisation term is a function of θ alone.

Testing and Training Error as Regularization Increases

We will now see how different values of alpha affect the performance of the model.

Our objective function is

$J = L_(\theta, \theta_0, x_{data}, y_{data}) + \alpha R(\theta)$

At $\alpha = 0$, the obj model becomes just the loss model, and our functions should approximate very well f_{data}, the relation between training data and training outputs.

Accuracy is defined as $1/J$, as we are trying to minimise J.

As we increase α, J increases and the accuracy reduces.

What about the J function as computed from the testing dataset? As α increases it first reduce, as the model is gaining generability, but then it start back to increases as the model becomes too much general. The opposite for its accuracy.

There is indeed a value α^* for which indeed the J function under the testing set is minimised (accuracy is maximised).

And what't the α that we want to use. How do we find it with only training set data ? (we can't use testing data, as we don't yet have that data). The method is "cross validation" (next video).

Cross Validation

We want to find α^* . We just split our training data in a "new" training data, and a "validation set" to act as the "test set" and fine tune our α hyperparameter.

We start to discretize our hyperparameter in a number of K values to test (there are many ways to do that… ex-ante using a homogeneous grid or sampling approach, or updating the values to test as we have the results of the first α we tested…)

We then divide our training data in n partitions (where n depends on the data available), and, for each α_k, we choose one of that n partitions to be the "validation set": we minimise the J function using α_k and the remaining $(n-1)$ partitions, and we compute the accuracy $S_n(\alpha_k)$ for the validation set.

We then compute the average accuracy for α_k: $S(\alpha_k) = \frac{1}{n}\sum_{i=1}^{n} S_n(\alpha_k)$.

Finally we "choose" the α^* with the lowest $S(\alpha_k)$.

Tumor Diagnosis Demo

- Python code
- Julia code

In this demo we found the best value of alpha in a SVM model that links breast cancer characteristics (such as cancer texture or size) with its benign/malign nature.

The example use sikit-learn, a library that allows to train your own machine learning algorithm (linear SVM in our case).

We use the function `linear_model.SGDClassifier(loss='hinge', penalty='l2', alpha=alpha[i])` that set up a SVM model using gradient descendent algorithm on hinge loss, using the L2 norm as regularisation term ("penalty") and the specific α we want to test.

We then use `ms.cross_val_score(model, X, y, cv=5)` to actually train the model on the training set divided in 5 different partitions. The function return already an array of the scores of the remaining validation partition. To compute the score for that particular alpha we now need just to average the score array obtained by that function.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 02 - Nonlinear Classification, Linear regression, Collaborative Filtering

Lecture 5. Linear Regression

5.1. Unit 2 Overview

Building up from the previous unit, in this unit we will introduce:

- linear regression (output a number in R)
- non-linear classification methods
- recommender problems (sometime called collaborative filtering problems)

5.2. Objectives

At the end of this lecture, you will be able to

- write the training error as least squares criterion for linear regression
- use stochastic gradient descent for fitting linear regression models
- solve closed-form linear regression solution
- identify regularization term and how it changes the solution, generalization

5.3. Introduction

Today we will see Linear Classification In the last unit we saw linear *classification*, where we was trying to land the mapping between the feature vectors of our data ($x^{(t)} \in \mathbb{R}^d$) and the corresponding (binary) label ($y^{(t)} \in \{-1, +1\}$).

This relatively simple set up can be already used to answer pretty complex questions, like making recommendations to buy or not some stocks, where the feature vector is given by the stock prices in the last d-days.

We can extend such problem to return, instead of just a binary output (price will increase - buy, vs price will drop - sell) a more informative output on the extent of the expected variation in price.

With regression we want to predict things that are continuous in nature.

The set up is very similar to before with $x^{(t)} \in \mathbb{R}^d$. The only differences is that now we consider $y^{(t)} \in \mathbb{R}$).

The goal of our model will be to map-- to learn how to map-- feature vectors into these continuous values.

We will consider for now only linear regression:

$f(\mathbf{x}; \theta, \theta_0) = \sum_{i=1}^{d} \theta_i x_i + \theta_0 = \theta \cdot \mathbf{x} + \theta_0$

For compactness of the equations we will consider $\theta_0 = 0$ (we can always think of θ_0 of being just a $d + 1$ dimension of θ).

Are we limiting ourself to just linear relations? No, we can still use linear classifier by doing an appropriate representation of the feature vector, by mapping into some kind of complex space and from that mapping then apply linear regression (note that at this point we will not yet talk about how we can construct this feature vector position. We will assume instead that somebody already gave us an appropriate feature vector).

3 questions to address:

1. Which would be an appropriate objective that can quantify the extent of our mistake?. How do we address the correctness of our output? In classification we had a binary error term, here it must be a range, our prediction could be “almost there” or very far.
2. How do we set up the learning algorithm. We will see two today, a numerical, gradient based ones, and a analytical, closed-form algorithm where we do not need to approximate.

3. How we do regularisation. How we perform a better generalization to be more robust when we don't have enough training data or when the data is noisy

5.4. Empirical Risk

Let's deal with the first question, the objective. We want to measure how much our prediction deviates from the know values of y, how far from y they are. We call our objective **empirical risk** (here denoted with R) and will be a sort of average loss of all our data points, depending from the parameter to estimate.

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}_h(y^{(t)} - \theta \cdot \mathbf{x}^{(t)}) = \frac{1}{n}\sum_{t=1}^{n} \frac{(y^{(t)} - \theta \cdot \mathbf{x}^{(t)})^2}{2}$$

Why squaring the deviation? Intuitively, since our training data may be noisy and the values that we record may be noisy, if it is a small deviation between our prediction and the true value, it's OK. However, if the deviation is large, we want really, truly penalize. And this is the behaviour we are getting from the squared function, that the bigger difference would actually result in much higher loss.

Note that the above equation use the squared error as loss function, but other loss functions could also be used, e.g. the hinge loss we already saw in unit 1:

$$\text{Hinge Loss}_h(z) = \begin{cases} 0 & \text{if } z \geq 1 \\ 1 - z & \text{oth.} \end{cases}$$

I will minimise this risk for the known data, but what I really want to do it so get it minimised for the unknown data I don't already see.

2 mistakes are possible:

1. **structural mistakes** Maybe the linear function is not sufficient for you to model your training data. Maybe the mapping between your training vectors and y's is actually highly nonlinear. Instead of just considering linear mappings, you should consider a much broader set of function. This is one class of mistakes.
2. **estimation mistakes** The mapping itself is indeed linear, but we don't have enough training data to estimate the parameters correctly.

There is a trade-off between these two kind of error: on one side, minimising the structural mistakes ask for a broader set of functions with more parameters, but this, at equal training set size, would increase the estimation mistakes. On the other side, minimising the estimation mistakes call for simpler set of functions, with less parameters, where however I become susceptible for structural mistakes.

In this lesson we remain commit to linear regression, and we want to minimise the empirical risk.

5.5. Gradient Based Approach

In this segment we will study the first of the two algorithms to implement the learning phase, the gradient based approach.

The advantage of the Empirical Risk function with the squared error as loss is that it is differentiable everywhere. Its gradient with respect to the parameter is:

$$\nabla_\theta\left(\frac{(y^{(t)} - \theta \cdot \mathbf{x}^{(t)})^2}{2}\right) = -(y^{(t)} - \theta \cdot \mathbf{x}^{(t)}) * \mathbf{x}^{(t)}$$

We will implement its stochastic variant: we start by randomly select one sample in the training set, look at its gradient, and update our parameter in the opposite direction (as we want to minimise the empirical risk).

The algorithm will then be as follow:

1. We initialise the thetas to zero
2. We randomly pick up a data pair
3. We compute the gradient and update θ as $\theta = \theta - \eta * \nabla_\theta = \theta + \eta * (y^{(t)} - \theta \cdot \mathbf{x}^{(t)}) * \mathbf{x}^{(t)}$ where η is the learning rate, influencing the size of the movement of the parameter at each iteration. We can have η constant or making it depends from the number of k iteration we already have done, e.g. $\eta = 1/(1+k)$, so to minimise the steps are we get closer to our minimum.

Note that the parameter updates at each step, not only on some “mistake” like in classification (i.e. we treat all deviations as “mistake”), and that the amount depends on the deviation itself (i.e. not of a fixed amount like in classification). Going against the gradient assure that the algorithm self-correct itself, i.e. we obtain parameters that lead to predictions closer and closer to the actual true y.

5.6. Closed Form Solution

The second learning algorithm we study is the closed form (analytical) solution. This is quite an exception in the machine learning field, as typically closed form solutions do not typically exist. But here the empirical risk happens to be a convex function that we can solve it exactly.

Let’s compute the gradient with respect of θ, but this time of the whole empirical risk, not just the loss function:

$$R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}$$

$$\nabla_\theta R_n(\hat{\theta}) = \frac{1}{n}\sum_{t=1}^{n}\nabla_\theta\left(\frac{(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})^2}{2}\right) = -\frac{1}{n}\sum_{t=1}^{n}(y^{(t)}-\hat{\theta}\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)}$$

$$\nabla_\theta R_n(\hat{\theta}) = -\frac{1}{n}\sum_{t=1}^{n}y^{(t)}*\mathbf{x}^{(t)} + \frac{1}{n}\sum_{t=1}^{n}\mathbf{x}^{(t)}*(\mathbf{x}^{(t)})^T*\hat{\theta}$$

$$\nabla_\theta R_n(\hat{\theta}) = -\mathbf{b} + \mathbf{A}\hat{\theta} = 0$$

With $b = \frac{1}{n}\sum_{t=1}^{n}y^{(t)}x^{(t)}$ is a $(d\times 1)$ vector and $\mathbf{A} = \frac{1}{n}\sum_{t=1}^{n}x^{(t)}(x^{(t)})^T$ is an $(d\times d)$ matrix. If $\mathbf{A}$ is invertible we can finally write $\hat{\theta} = \mathbf{A}^{-1}b$.

We can invert $\mathbf{A}$ only if the feature vectors $\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(n)}$ span over R^d, that is if $n >> d$.

Also, we should put attention that inverting $\mathbf{A}$ is an operation of order $O(d^3)$, so we should be carefully when d is very large, like in bag of words approaches used in sentiment analysis where d can be easily be in the tens of thousands magnitude.

5.7. Generalization and Regularization

We now focus the discussion in how do we assure that the algorithm we found, the parameters we estimated, will be good also for the unknown data, will be robust and not too much negatively impacted by the noise that it is in our training data?

This question is more and more important as we have less data to train the algorithm.

The way to solve this problem is to use a mechanism called regularisation that try to push us away to fit the training data “perfectly” (where we “fit” also the errors, the noises embedded in our data) and try to instead generalise to possibly unknown data.

The idea is to introduce something that push the thetas to zero, so that it would be only worth for us to move our parameters if there is really a very strong pattern that justify the move.

5.8. Regularization

The implementation of the regularisation we see in this lesson is called **ridge regression** and it is the same we used in the classification problem:

The new objective function to minimise becomes:

$$J_{n,\lambda} = R_n(\theta) + \frac{\lambda}{2}||\theta||^2$$

The first term is our empirical risk, and it catches how well we are fitting the data. The second term, being the square norm of thetas, tries to push the thetas to remain zero, to not move unless there is a significant advantage in doing so. And λ is the parameter that determine the trade off, the relative contribution, between these two terms. Note that being the norm it doesn’t influence any specific dimension of theta. Its role is is actually to determine how much do I care to fit my training examples versus how much do I care to be staying close to zero. In other words, we don’t want any weak piece of evidence to pull our thetas very strongly. We want to keep them grounded in some area and only pulls them when we have enough evidence that it would really, in substantial way, impact the empirical loss.

In other terms, the effect of regularization is to restrict the parameters of a model to freely take on large values. This will make the model function smoother, leveling the ‘hills’ and filling the ‘vallyes’. It

will also make the model more stable, as a small perturbation on x will not change y significantly with smaller $||\theta||$.

What's very nice about using the squared norm as regularisation term is that actually everything that we discussed before, both the gradient and closed form solution, can be very easily adjusted to this new loss function.

Gradient based approach with regularisation

With respect to a single point the gradient of J is :

$$\nabla_\theta(\frac{(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})^2}{2}+\frac{\lambda}{2}||\theta||^2)=-(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)}+\lambda\theta$$

We can modify the gradient descent algorithm where the update rule becomes:

$$\theta=\theta-\eta*\nabla_\theta=\theta-\eta*(\lambda\theta-(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)})=(1-\eta\lambda)\theta+\eta*(y^{(t)}-\theta\cdot\mathbf{x}^{(t)})*\mathbf{x}^{(t)})$$

The difference with the empirical risk without the regularisation term is the $(1-\eta\lambda)$ term that multiply θ trying to put it down at each update.

The closed form approach with regularisation

First solve for θ_O and then solve for θ (see exercise given in Homework 3, tab "5. Linear Regression and Regularization").

In matrix form the closed form solution is:

$\theta=(X^TX+\lambda I)^{-1}X^TY$ (also known as Tikikhonov regularisation)

Note that again we can consider θ_0 just as a further dimension of θ where the X are all ones.

5. 9. Closing Comment

By using regularisation, by requiring much more evidence to push the parameters into the right direction we will increase the mistakes of our prediction within the training set, but we will reduce the test error when the fitted thetas are used with respect to data that has not been used for the fitting step.

However if we continue to increase λ, if we continue to give weight to the regularisation term, then also the testing error will start to increase as well.

Our objective is to find the "sweet spot" of lambda where the test error is those that is minimised, and we can do that using the validation set to calibrate the value that lambda should have.

We saw regularization in the context of linear regression, but we will see regularization across many different machine learning tasks. This is just one way to implement it. We will see some other mechanism to implement regularisation, for example in neural networks.

Lecture 6. Nonlinear Classification

6.1. Objectives

At the end of this lecture, you will be able to

- derive non-linear classifiers from feature maps
- move from coordinate parameterization to weighting examples
- compute kernel functions induced from feature maps
- use kernel perceptron, kernel linear regression
- understand the properties of kernel functions

6.2. Higher Order Feature Vectors

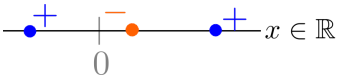
Outline

- Non-linear classification and regression
- Feature maps, their inner products
- Kernel functions induced from feature maps
- Kernel methods, kernel perceptron
- Other non-linear classifiers (e.g. Random Forest)

This lesson deals with non-linear classification, with the basic idea to expand the feature vector, map it to a higher dimensional space, and then feed this new vector to a linear classifier.

The computational disadvantage of using higher dimensional space can be avoided using so-called kernel functions. We will then see linear models applied to these kernel models, and in particular, for simplicity, the perceptron linear model (that when used with kernel function becomes the “kernel perceptron”).

Let’s see an example in 1D: we have the real line on which we have our points we want to classify. We can easily see that if we have the set of points {-3: positively labelled, 2: negatively labelled, 5:positively labelled} there is no linear classifier that can correctly classify this data set:

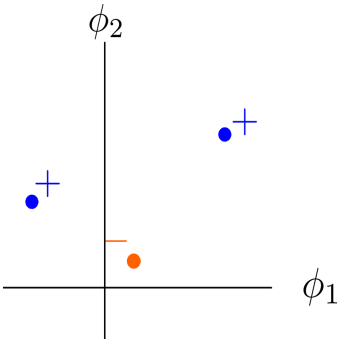


We can remedy the situation by introducing a feature transformation feeding a different type of example to the linear classifier.

We will always include in the new feature vector the original vector itself, so as to be able to retain the power that was available prior to feature transformation. But we will also add to it additional features. Note that, differently from statistics, in Machine Learning we know nothing (assume nothing) about the distribution of our data, so removing the original data to keep only the new added feature would risk to remove information that is not captured in the transformation.

In this case we can add for example x^2. So the mapping is $\mathbf{x} \in \mathbb{R}^2 = \phi(x \in \mathbb{R}) = \begin{bmatrix} x \\ x^2 \end{bmatrix}$. As result also the θ parameter of the classifier became bidimensional.

Our dataset becomes {(-3,9)[+], (2,4)[-], (5,25)[+]} that can be easily classified by a linear classifier in 2D (i.e. a line) $h(x;\theta,\theta_0) = \text{sign}(\theta \cdot \phi(x) + \theta_0)$:



Note that the linear classifier in the new feature space we had found, back in the original space becomes a non-linear classifier: $h(x;\theta,\theta_0) = \text{sign}(\theta_1 * x + \theta_2 * x^2 + \theta_0)$.

An other example would be having our original dataset in 2D as {(2,2)[+],(-2,2)[-],(-2,-2)[+],(2,-2)[-]} that is not separable in 2D, but it becomes separable in 3D when I use a feature transformation like $\mathbf{x} \in \mathbb{R}^3 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{bmatrix}$, for example by the plane given by $\left(\theta = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \theta_0 = 0\right)$.

6.3. Introduction to Non-linear Classification

We can get more and more powerful classifiers by adding linearly independent features, x^2, x^3,... This x, x^2,..., as functions are linearly independent, so the original coordinates always provide something above and beyond what were in the previous ones.

Note that when $\mathbf{x}$ is already multidimensional, even just $\phi(\mathbf{x}) = \mathbf{x}^2$ would result in dimensions exploding, e.g. $\mathbf{x} \in \mathbb{R}^5 = \phi(\mathbf{x} \in \mathbb{R}^2) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ \sqrt{2}x_1 x_2 \end{bmatrix}$ (the meaning of the scalar associated to the cross term will be discussed later).

Once we have the new feature vector we can make non-linear classification or regression in the original data making a linear classification or regression in the new feature space:

- Classification: $h(x;\theta,\theta_0) = \text{sign}(\theta \cdot \phi(x) + \theta_0)$
- Regression: $f(x;\theta,\theta_0) = \theta \cdot \phi(x) + \theta_0$

More feature we add (e.g. more polinomian grades we add), better we fit the data. The key question now is when is time to stop adding features ? We can use the validation test to test which is the polynomial form that, trained on the training set, respond better in the validation set.

At the extreme, you hold out each of the training example in turn in a procedure called **leave one out cross validation**. So you take a single training sample, you remove it from the training set, retrain the method, and then test how well you would predict that particular holdout example, and do that for each training example in turn. And then you average the results.

While very powerful, this explicit mapping into larger dimensions feature vectors is indeed… that the dimensions could become quickly very high as our original data is already multidimensional

Let's our original $\mathbf{x} \in \mathbb{R}^d$. Then a feature transformation:

- quadratic (order 2 polynomial): would involve $d+ \approx d^2$ dimensions (the original dimensions plus all the cross products)
- cubic (order 3 polynomial): would involve $d+ \approx d^2+ \approx d^3$ dimensions

The exact number of terms of a feature transformation of order p of a vector of d dimensions is $\sum_{i=1}^{p} \binom{d+i-1}{i}$ (the sum of multiset numbers).

So our feature vector becomes very high-dimensional very quickly if we even started from a moderately dimensional vector.

So we would want to have a more efficient way of doing that – operating with high dimensional feature vectors without explicitly having to construct them. And that is what kernel methods provide us.

6.4. Motivation for Kernels: Computational Efficiency

The idea is that you can take inner products between high dimensional feature vectors and evaluate that inner product very cheaply. And then, we can turn our algorithms into operating only in terms of these inner products.

We define the kernel function of two feature vectors (two different data pairs) applied to a given ϕ transformation as the dot product of the transformed feature vectors of the two data:

$$k(x,x';\phi) \in \mathbb{R}^+ = \phi(x) \cdot \phi(x')$$

We can hence think of the kernel function as a kind of similarity measure, how similar the x example is to the x' one. Note also that being the dot product symmetric and positive, kernel functions are in turn symmetric and positive.

For example let's take x and x' to be two dimensional feature vectors and the feature transformation $\phi(x)$ defined as $\phi(x) = [x_1, x_2, x_1^2, \sqrt(2)x_1x_2, x_2^2]$ (so that $\phi(x')$ is $[x'_1, x'_2, {x'_1}^2, \sqrt(2)x'_1x'_2, {x'_2}^2]$)

This particular ϕ transformation allows to compute the kernel function very cheaply:

$$k(x,x';\phi) = \phi(x) \cdot \phi(x')$$

$$= x_1x'_1 + x_2x'_2 + {x_1}^2{x'_1}^2 + 2x_1x'_1x_2x'_2 + {x_2}^2{x'_2}^2$$

$$= (x_1x'_1 + x_2x'_2) + (x_1x'_1 + x_2x'_2)^2$$

$$= x \cdot x' + (x \cdot x')^2$$

Note that even if the transformed feature vectors have 5 dimensions, the kernel function return a scalar. In general, for this kind of feature transformation function ϕ, the kernel function evaluates as $k(x,x';\phi) = \phi(x) \cdot \phi(x') = (1 + x \cdot x')^p$, where p is the order of the polynomial transformation ϕ.

However, it is only for *some* ϕ for which the evaluation of the kernel function becomes so nice! As soon we can prove that a particular kernel function can be expressed as the dot product of two particular feature transformations (for those interested the *Mercer's theorem* stated in these notes) the kernel function is *valid* and we don't actually need to construct the transformed feature vector (the output of ϕ).

Now our task will be to turn a linear method that previously operated on $\phi(x)$, like sign$(\theta \cdot \phi(x) + \theta_0)$ to an inter-classifier that only depends on those inner products, that operates in terms of kernels.

And we'll do that in the context of kernel perception just for simplicity. But it applies to any linear method that we've already learned.

6.5. The Kernel Perceptron Algorithm

Let's show how we can use the kernel function in place of the feature vectors in the perceptron algorithm.

Recall that the perceptron algoritm was (excluding for simplicity θ_0):

```
θ = 0                # initialisation
for t in 1:T
  for i in 1:n
    if yⁱ θ·Φ(xⁱ) ≦ 0   # checking if sign is the same, i.e. data is on the right
side of its label
      θ = θ + yⁱΦ(xⁱ)   # update θ if mistake
```

Which is the final value of the parameter θ resulting from such updates ? We can write it as

$$\theta^* = \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})$$

where α is the vector of number of mistakes (and hence updates) underwent for each data pair (so $\alpha^{(j)}$ is the (scalar) number of errors occurred with the *j*-th data pair).

Note that we can interpret α^j in terms of the relative importance of the *j*-th training example to the final predictor. Because we are doing perceptron, the importance is just in terms of the number of mistakes that we make on that particular example.

When we want to make a prediction of a data pair $(x^{(i)}, y^{(i)})$ using the resulting parameter value θ^* (that is the "optimal" parameter the perceptron algorithm can give us), we take an inner product with that:

$$\text{prediction}^{(i)} = \theta^* \cdot \phi(x^{(i)})$$

We can rewrite the above equation as :

$$\theta^* \cdot \phi(x^{(i)}) = [\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} \phi(x^{(j)})] \cdot \phi(x^{(i)})$$

$$= \sum_{j=1}^{n} [\alpha^{(j)} y^{(j)} \phi(x^{(j)}) \cdot \phi(x^{(i)})]$$

$$= \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)})$$

But this means we can now express success or errors in terms of the α vector and a valid kernel function (typically something cheap to compute) !

An error on the data pair $(x^{(i)}, y^{(i)})$ can then be expressed as $y^{(i)} * \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)}) \leq 0$. We can then base our perceptron algorithm on this check, where we start with initiating the error vector α to zero, and we run trought the data set checking for errors and, if found, updating the corresponding error term. I practice, our endogenous variable to minimise the errors is no longer directly theta, but became the α vector, that as said implicitly gives the contribution of each data pair to the θ parameter. The perceptron algorithm becomes hence the **kernel perceptron algorithm**:

```
α = 0                # initialisation of the vector
for t in 1:T
  for i in 1:n
    if yⁱ ∑ⱼ[αʲyʲk(xʲ,xⁱ)] ≦ 0   # checking if prediction is right
      αⁱ += 1  # update αⁱ if mistake
```

When we have run the algorithm and found the optimal α^* we can either:

- immediately retrieve the optimal θ^* by the above equation and make predictions using $\hat{y}^{(i)} = \theta^* \cdot \phi(x^{(i)})$
- entirely skip $\phi(x^{(i)})$ making predictions on $x^{(i)}$ just using $\hat{y}^{(i)} = (\sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x^{(i)}) > 0) * 2 - 1$ (where the outer transformation is to have back y in the form {-1,1})

Which method to use depends on which one is easier to compute, noting that when $\phi(x)$ has infinite dimensions we are forced to use the second one.

6.6. Kernel Composition Rules

Now instead of directly constructing feature vectors by adding coordinates and then taking it in the product and seeing how it collapses into a kernel, we can construct kernels directly from simpler kernels by made of the following **kernel composition rules**:

1. $K(x, x') = 1$ is a valid kernel whose feature representation is $\phi(x) = 1$;
2. Given a function $f : \mathbb{R}^d \to \mathbb{R}$ and a valid kernel function $K(x, x')$ whose feature representation is $\phi(x)$, then $\tilde{K}(x, x') = f(x)K(x, x')f(x')$ is also a valid kernel whose feature representation is $\tilde{\phi}(x) = f(x)\phi(x)$
3. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x)$ and $\phi_b(x)$, then $K(x, x') = K_a(x, x') + K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_a(x) \\ \phi_b(x) \end{bmatrix}$
4. Given $K_a(x, x')$ and $K_b(x, x')$ being two valid kernels whose feature representations are respectively $\phi_a(x) \in \mathbb{R}^A$ and $\phi_b(x) \in \mathbb{R}^B$, then $K(x, x') = K_a(x, x') * K_b(x, x')$ is also a valid kernel whose feature representation is $\phi(x) = \begin{bmatrix} \phi_{a,1}(x) * \phi_{b,1}(x) \\ \phi_{a,1}(x) * \phi_{b,2}(x) \\ \phi_{a,1}(x) * \phi_{b,...}(x) \\ \phi_{a,1}(x) * \phi_{b,B}(x) \\ \phi_{a,2}(x) * \phi_{b,1}(x) \\ \phi_{a,...}(x) * \phi_{b,...}(x) \\ \phi_{a,A}(x) * \phi_{b,B}(x) \end{bmatrix}$ (see this lecture notes for a proof)

Armed with these rules we can build up pretty complex kernels starting from simpler ones.

For example let's start with the identity function as ϕ, i.e. $\phi_a(x) = x$. Such feature function results in a kernel $K(x, x'; \phi_a) = K_a(x, x') = (x \cdot x')$ (this is known as the **linear kernel**).

We can now add to it a squared term to form a new kernel, that by virtue of rules (3) and (4) above is still a valid kernel:

$$K(x, x') = K_a(x, x') + K_a(x, x') * K_a(x, x') = (x \cdot x') + (x \cdot x')^2$$

6.7. The Radial Basis Kernel

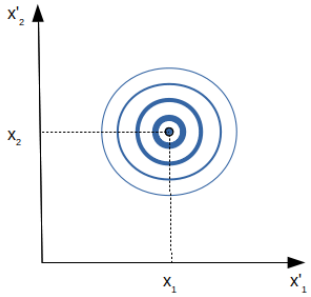
We can use kernel functions, and have them in term of simply, cheap-to-evaluate functions, even when the underlying feature representation would have infinite dimensions and would be hence impossible to explicitly construct.

One example is the so called **radial basis kernel**:

$$K(x, x') = e^{-\frac{1}{2}||x - x'||^2}$$

It can be proved that suck kernel is indeed a valid kernel and its corresponding feature representation $\phi(x) \in \mathbb{R}^\infty$, i.e. involves polynomial features up to an infinite order.

Does the radial basis kernel look like a Gaussian (without the normalisation term) ? Well, because indeed it is:



The above picture shows the contour lines of the radial basis kernel when we keep fixed x (in 2 dimensions) and we let x' to move away from it: the value of the kernel then reduces in a shape that in 3-d would resemble the classical bell shape of the Gaussian curve. We could even parametrise the radial basis kernel replacing the fixed $1/2$ term with a parameter γ that would determine the width of the bell-shaped curve (the larger the value of γ the narrower will be the bell, i.e. small values of γ yield wide bells).

Because the feature has infinite dimensions, the radial basis kernel has infinite expressive power and can correctly classify any training test.

The linear decision boundary in the infinite dimensional space is given by the set $\{x : \sum_{j=1}^{n} \alpha^{(j)} y^{(j)} k(x^{(j)}, x) = 0\}$ and corresponds to a (possibly) non-linear boundary in the original feature vector space.

The more difficult task it is, the more iterations before this kernel perception (with the radial basis kernel) will find the separating solution, but it always will in a finite number of times. This is by contrast with the “normal” perceptron algorithm that when the set is not separable would continue to run at the infinite, changing its parameters unless it is stopped at a certain arbitrary point.

Other non-linear classifiers

We have seen as we can have nonlinear classifiers extending to higher dimensional space and evenutally using kernel methods to collapse the calculations and operate only *implicitly* in those high dimension spaces.

There are certainly other ways to get nonlinear classifiers.

Decision trees make classification or regression operating sequentially on the various dimensions and making at each step a separation of the data in two subsets and continue recursivelly to “split” the data like in a tree. And you can “learn” these trees incrementally: the idea is that at each step you will look for the best criteria for splitting by iterating over every feature and each value for each ferature and calculating the information gain that you would obtain from splitting the data using that criteria, that is looking on how much you gained in homogeneity in the splitted data compared with the unsplitted one (frequently used function to measure the (dis)homogeneity are `gini` or `entropy` for categorical tasks and `variance` for numerical tasks).

There is a way to make these decision trees more robust, called **random forest classifiers**, that adds two type of randomness: the first one is in randomly choosing the dimension on which to operate the cut, the second in randomly selecting the single example on which operate from the data set (with replacement) and then just average the predictions obtained from these trees.

So the procedure of a random forest classifier is:

- boostrap the sample
- build a randomized (by dimension) decision tree
- average the predictions (ensemble)

The biggest advantage of Decision trees / random forests is that they can “crunch” almost any kind of data without any transformation. On top of coninuous numerical values, they can direclty work on any categorical or ordinal feature, and can be used for either regression or classification. Further, trained decision trees are relatively easy to interpret.

Summary

- We can get non-linear classifiers (or regression) methods by simply mapping our data into new feature vectors that include non-linear components, and applying a linear method on these resulting vectors;
- These feature vectors can be high dimensional, however;

- We can turn linear methods into kernel methods by casting the computation in terms of inner products;
- A kernel function is advantageous when the inner products are faster to evaluate than using explicit feature vectors (e.g. when the vectors would be infinite dimensional!)
- We saw the radial basis kernel that is particularly powerful because it is both (a) cheap to evaluate and (b) has a corresponding infinite dimensional feature vector

Lecture 7. Recommender Systems

7.1. Objectives

At the end of this lecture, you will be able to

- understand the problem definition and assumptions of recommender systems
- understand the impact of similarity measures in the K-Nearest Neighbor method
- understand the need to impose the low rank assumption in collaborative filtering
- iteratively find values of U and V (given $X = UV^T$) in collaborative filtering

7.2. Introduction

This lesson deals about recommender systems, when the algorithm try to guess preferences based on choices already made by the user (like film to watch or products to buy).

We'll see:

- the exact problem definition
- the historically used algorithm, the K-Nearest Neighbor algorithm (KNN);
- the algorithm in use today, the matrix factorization or collaborative filtering.

Problem definition

We keep as example across the lecture the recommendation of movies.

We start with a $(n \times m)$ matrix Y of preferences for user $a = 1, \dots, n$ of movie $i = 1, \dots, m$. While there are many ways to store preferences, we will use real numbers.

The goal is to base the prediction on the prior choices of the users, considering that this Y matrix could be very sparse (e.g. out of 18000 films, each individual ranked very few of them!), i.e. we want to fill these "empty spaces" of the matrix.

Why not to use classification/regression based on feature vectors as learned in Lectures 1? For two reasons:

1. Deciding which feature to use or extracting them from data could be hard/infeasible (e.g. where can I get the info if a film has a happy or bad ending ?), or the feature vector could become very very large (things about in general "products" for Amazon, could be everything)
2. Often we have little data about a single users preferences, while to make a recommendation based on its own previous choices we would need lot of data.

The "trick" is then to "borrow" preferences from the other users and trying to measure how much a single user is closer to the other ones in our dataset.

7.3. K-Nearest Neighbor Method

The number K here means, how big should be your advisory pool on how many neighbors you want to look at. And this can be one of the hyperparameters of the algorithm.

We look at the k closest users that did score the element I am interested to, look at their score for it, and average their score.

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} Y_{b,i}}{K}$$

where $KNN(a, i; K)$ defines the set of K users closer to the user a that have a score for item i.

Now, the question is of course how do I define this similarity? We can use any method to define similarity between vectors, like cosine similarity ($\cos\theta = \frac{x_a \cdot x_b}{\|x_a\| \|x_b\|}$) or Euclidean distance ($\|x_a - x_b\|$).

We can make the algorithm a bit more sophisticated by weighting the neighbour scores to the level of similarity rather than just take their unweighted average:

$$\hat{Y}_{a,i} = \frac{\sum_{b \in KNN(a,i;K)} sim(a,b) * Y_{b,i}}{\sum_{b \in KNN(a,i;K)} |sim(a,b)|}$$

where $sim(a,b)$ is some similarity measure between users a and b.

There has been many improvements that has been added to this kind of algorithm, like adjusting for the different "average" score that each user gives to the items (i.e. they compare the deviations from user's averages rather than the raw score itself).

Still they are very far from today's methods. The problem of KNN is that it doesn't enable us to detect the hidden structures that is there in the data, which is that users may be similar to some pool of other users in one dimension, but similar to some other set of users in a different dimension. For example, loving machine learning books, and having there some "similarity" with other readers of machine learning books on some hidden characteristics (e.g. liking equation-rich books or more discursive ones), and plant books, where the similarity with other plant-reading users would be based on completely different hidden features (e.g. loving photos or having nice tabular descriptions of plants).

Conversely in collaborative filtering, the algorithm would be able to detect these hidden groupings among the users, both in terms of products and in terms of users. So we don't have to explicitly engineer very sophisticated similarity measure, the algorithm would be able to pick up these very complex dependencies that for us, as humans, would be definitely not tractable to come up with.

7.4. Collaborative Filtering: the Naive Approach

Let's start with a *naive* approach where we just try to apply the same method we used in regression to this problem, i.e. minimise a function J made of a distance between the observed score in the matrix and the estimated one and a regularisation term.

For now, we treat each individual score independently… and this will be the reason for which (we will see) this method will not work.

So, we have our (sparse) matrix Y and we want to find a dense matrix X that is able to replicate at best the observed points of $Y_{a,i}$ when these are available, and fill the missing ones when $Y_{a,i} = missing$.

Let's first define as D the set of points for which a score in Y is given: $D = \{(a,i) : Y_{a,i} \neq \text{missing}\}$.

The J function then takes any possible X matrix and minimise the distance between the points in the D set less a regularisation parameter (we keep the individual scores to zero unless we have strong belief to move them from such state):

$$J(X;Y,\lambda) = \frac{\sum_{(a,i) \in D} (Y_{a,i} - X_{a,i})^2}{2} + \frac{\lambda}{2} \sum_{(a,i)} X_{a,i}^2$$

To find the optimal $X^*_{a,i}$ that minimise the FOC $(\partial X_{a,i} / \partial Y_{a,i}) = 0$ we have to distinguish if (a,i) is in D or not:

- $(a,i) \in D$: $X^*_{a,i} = \frac{Y_(a,i)}{1+\lambda}$
- $(a,i) \notin D$: $X^*_{a,i} = 0$

Clearly this result doesn't make sense: for data we already know we obtain a bad estimation (as worst as we increase lambda) and for unknown scores we are left with zeros.

7.5. Collaborative Filtering with Matrix Factorization

What we need to do is to actually relate scores together instead of considering them independently.

The idea is then to constrain the matrix X to have a lower rank, as rank captures how much independence is present between the entries of the matrix.

At one extreme, constraining the matrix to be rank 1, would means that we could factorise the matrix X as just the matrix product of two single vectors, one defining a sort of general sentiment about the

items for each user (u), and the other one (v) representing the average sentiment for a given item, i.e. $X = uv^T$.

But representing users and items with just a single number takes us back to the KNN problem of not being able to distinguish the possible multiple groups hidden in each user or in each item.

We could then decide to divide the users and/or the items in respectively $(n \times 2)U$ and $(2 \times m)V^T$ matrices and constrain our X matrix to be a product of these two matrices (hence with rank 2 in this case): $X = UV^T$

The exact numbers K of vectors to use in the user/items factorisation matrices (i.e. the rank of X) is then a hyperparameter that can be selected using the validation set.

Still for simplicity, in this lesson we will see the simplest case of constraining the matrix to be factorisable by a pair of single vectors (i.e. $K = 1$).

7.6. Alternating Minimization

Using rank 1, we can adapt the J function to take the two vectors u and v instead of the whole X matrix, and our objective becomes to find their elements that minimise such function:

$$J(\mathbf{u}, \mathbf{v}; Y, \lambda) = \frac{\sum_{a,i \in D} (Y_{a,i} - u_a * v_i)^2}{2} + \frac{\lambda}{2} \sum_a^n u_a^2 + \frac{\lambda}{2} \sum_i^m v_i^2$$

How do we minimise J ? We can take an iterative approach where we start by randomly sampling values for one of the vector and minimise for the other vector (by setting the derivatives with respect on its elements equal to zero), then fix this second vector and going minimise for the first one, etc., until the value of the function J doesn't move behind a certain threshold, in an alternating minimisation exercise that will guarantee us to find a local minima (but not a global one!).

Note also that when we minimise for the individual component of one of the two vectors, we obtain derivatives with respect to the individual vector elements that are independent, so the first order condition can be expressed each time in terms of a single variable.

Numerical example

Let's consider a value of λ equal to 1 and the following score dataset:

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix}$$

and let start out minimisation algorithm with $v = [2, 7, 8]$

L becomes :

$$J(\mathbf{u}; \mathbf{v}, Y, \lambda) = \frac{(5-2u_1)^2 + (7-8u_1)^2 + (1-2u_2)^2 + (2-7u_2)^2}{2} + \frac{u_1^2 + u_2^2}{2} + \frac{2^2 + 7^2 + 8^2}{2}$$

From where, setting $\partial L / \partial u_1 = 0$ and $\partial L / \partial u_2 = 0$ we can retrieve the minimising values of (u_1, u_2) as 22/23 and 8/27. We can now compute $J(\mathbf{v}; \mathbf{u}, Y, \lambda)$ with these values of u to retrieve the minimising values of v and so on.

Project 2: Digit recognition (Part 1)

The softmax function

Summary from: https://en.wikipedia.org/wiki/Softmax_function

The softmax function is a "normalisation function" defined as:

$$\text{Softmax}(\mathbf{Z} \in \mathbb{R}^K) \in \mathbb{R}^{+K} = \frac{1}{\sum_{j=1}^k e^{Z_j}} * e^{\mathbf{Z}}$$

- it maps a vector in $\mathbb{R}^K$ to a new vector in $\mathbb{R}^{+K}$ where all values are positive and sum up to 1, hence loosing one degree of freedom and with the output interpretable as probabilities
- the larger (in relative term) is the input Z_K, the larger will be its output probability
- note that, for each K, the map $Z_K \to P(Z_K)$ is continuous
- it can be seen as a smooth approximation to the arg max function: the function whose value is which index has the maximum

The softmax function can be used in multinomial logistic regression to represent the predicted probability for the *j*'th class given a sample vector $\mathbf{x}$ and a weighting vector $\mathbf{w}$:

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\mathsf{T}\mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\mathsf{T}\mathbf{w}_k}}$$

It can be parametrised by a parameter τ referred as "temperature" in allusion to statical mechanics:

$$\text{softmax}(\mathbf{Z}; \tau \in \mathbb{R}) = \frac{1}{\sum_{j=1}^{k} e^{Z_j/\tau}} * e^{\mathbf{Z}/\tau}$$

- For high temperatures ($\tau \to \infty$), all actions have nearly the same probability and the lower the temperature, the more expected rewards affect the probability.
- For a low temperature ($\tau \to 0^+$), the probability of the action with the highest expected reward tends to 1.

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 03 - Neural networks

Lecture 8. Introduction to Feedforward Neural Networks

8.1. Unit 3 Overview

This unit deals with Neural Networks, powerful tools that have many different usages as self-driving cars or playing chess.

Neural networks are really composed of very simple units that are akin to linear classification or regression methods.

We will consider:

- feed-forward neural networks (simpler)
- recurrent neural networks: can model sequences and map sequences to class labels and even to other sequences (e.g. in machine translation)
- convolutional neural network: a specific architecture designed for processing images (used in the second part of project 2 for image classification)

At the end of this **unit**, you will be able to

- Implement a feedforward neural networks from scratch to perform image classification task.
- Write down the gradient of the loss function with respect to the weight parameters using back-propagation algorithm and use SGD to train neural networks.
- Understand that Recurrent Neural Networks (RNNs) and long short-term memory (LSTM) can be applied in modeling and generating sequences.
- Implement a Convolutional neural networks (CNNs) with machine learning packages.

8.2. Objectives

At the end of this **lecture**, you will be able to

- Recognize different layers in a feedforward neural network and the number of units in each layer.
- Write down common activation functions such as the hyperbolic tangent function `tanh`, and the rectified linear function `ReLU`.
- Compute the output of a simple neural network possibly with hidden layers given the weights and activation functions .
- Determine whether data after transformation by some layers is linearly separable, draw decision boundaries given by the weight vectors and use them to help understand the behavior of the network.

8.3. Motivation

The topic of feedforward neural networks is split into two parts:

- part one (this lesson): the model
  - motivations (what they bring beyond non-linear classification methods we have already seen);
  - what they are, and what they can capture;
  - the power of hidden layers.
- part two (next lesson): learning from data: how to use simple stochastic gradient descent algorithms as a successful way of actually learning these complicated models from data.

Neural networks vs the non-linear classification methods that we saw already

Let's consider a linear classifier $\hat{y} = sign(\theta \cdot \phi(\mathbf{x}))$.

We can interpret the various dimensions of the original feature vector $\mathbf{x} \in \mathbb{R}^d$ as d nodes. Then these nodes are mapped (non necessairly in a linear way) to D nodes of the feature representation of $\phi(\mathbf{x} \in \mathbb{R}^D)$. Finally we take a linear combination of these nodes (dimensions), we apply our sign function and we obtain the classification.

The key difference between neural networks and the methods we saw in unit 2 is that, there, the mapping from x to $\phi(x)$ in not part of the data analysis, it is done ex-ante (even, although implicitly, with the choice of a particular kernel), and then we optimise once we chose ϕ.

In neural network instead the choice of the ϕ mapping is endogenous to the learning step, together with the choice of θ.

Note that we have a bit of a chicken and egg problem here, as, in order to do a good classification - understand and learn the parameters theta for the classification decision - we would need to know what that feature representation is. But, on the other hand, in order to understand what a good feature representation would be, we would need to know how that feature representation is exercised in the ultimate classification task.

So we need to learn ϕ and θ jointly, trying to adjust the feature representation together with how it is exercised towards the classification task.

Motivation to Neural Networks (an other summary):

So far, the ways we have performed non-linear classification involve either first mapping x explicitly into some feature vectors $\phi(x)$, whose coordinates involve non-linear functions of x, or in order to increase computational efficiency, rewriting the decision rule in terms of a chosen kernel, i.e. the dot product of feature vectors, and then using the training data to learn a transformed classification parameter.

However, in both cases, the feature vectors are chosen. They are not learned in order to improve performance of the classification problem at hand.

Neural networks, on the other hand, are models in which the feature representation is learned jointly with the classifier to improve classification performance.

8.4. Neural Network Units

Real neurons:

- take a signal in input using dendroids that connect the neuron to ~ 10^3 _ 10^4 other neurons
- the signal potential increases in the neuron's body
- once a threshold is reached, the neuron in turn emits a signal that, through its axon, reaches ~ 100 other neurons

Artificial neural networks:

- the parameter associated to each nodes of a layer (input coordinates) takes the role of weights trying to mimic the strength of the connection that propagates to the cell body;
- the response of a cell is in terms of a nonlinear transformation of the aggregated, weighted inputs, resulting in a single real number.

A bit of terminology:

- The individual coordinates are known as **nodes** or **units**.
- The **weights** are denoted with w instead of θ as we were used to.
- Each node's **aggregated input** is given by $z = \sum_{i=1}^{d} x_i w_i + w_0$ (or, in vector form, $z = \mathbf{x} \cdot \mathbf{w} + w_0$, with $z \in \mathbb{R}, \mathbf{x} \in \mathbb{R}^d, \mathbf{w} \in \mathbb{R}^d$)
- The output of the neuron is the result of a non-linear transformation of the aggregated input called **activation function** $f = f(z)$
- A **neural network unit** is a primitive neural network that consists of only the "input layer", and an output layer with only one output.

Common kind of activation functions:

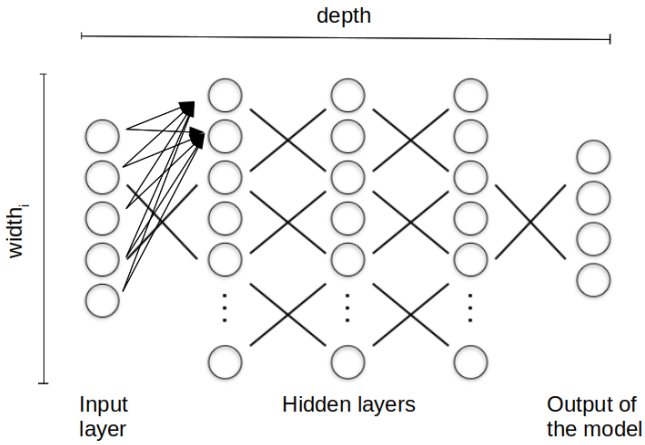
- linear. Used typically at the very end, before measuring the loss of the predictor;
- relu ("rectified linear unit"): $f(z) = max\{0, z\}$
- tanh ("hyperbolic tangent"): it mimics the sine function but in a soft way: it is a sigmoid curve spanning from -1 (at $z = -\infty$) to +1 (at $z = +\infty$), with a value of 0 at $z = 0$. Its smoothness property is useful since we have to propagate the training signal through the network in order to adjust all the parameters in the model. $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 1 - \frac{2}{e^{2z}+1}$. Note that $tanh(-z) = -tanh(z)$.

Our objective will be to learn the weights that make this node, in the context of the whole network, to then function appropriately, to behave well.

8.5. Introduction to Deep Neural Networks

Overall architecture

In **deep forward neural networks**, neural network units are arranged in **layers**, from the *input layer*, where each unit holds the input coordinate, through various *hidden layer* transformations, until the actual *output* of the model:



In this layerwise computation, each unit in a particular layer takes input from *all* the preceding layer units. And it has its own parameters that are adjusted to perform the overall computation. So parameters are different even between different units of the same layer. A deep (feedforward) neural network refers hence to a neural network that contains not only the input and output layers, but also hidden layers in between.

- **Width** (*of the layer*): number of units in that specific layer
- **Depth** (*of the architecture*): number of layers of the overall transformation before arriving to the final output

Deep neural networks

- loosely motivated by biological neurons, networks
- adjustable processing units (~ linear classifiers)
- **highly parallel** (important!), typically organized in layers
- deep = many transformations (layers) before output

For example the input could be an image, so the input vector is the individual pixel content, from which the first layer try to detect edges, then these are recombined into parts (in subsequent layers), objects and finally characterisation of a scene: edges -> simple parts-> parts -> objects -> scenes

One of the main advantages of deep neural networks is that in many cases, they can learn to extract very complex and sophisticated features from just the raw features presented to them as their input. For instance, in the context of image recognition, neural networks can extract the features that differentiate a cat from a dog based only on the raw pixel data presented to them from images.

The initial few layers of a neural networks typically capture the simpler and smaller features whereas the later layers use information from these low-level features to identify more complex and sophisticated features.

Note: it is interesting to note that a neural network can represent any given binary function.

Subject areas

Deep learning has overtaken a number of academic disciplines in just a few years:

- computer vision (e.g., image, scene analysis)
- natural language processing (e.g., machine translation)
- speech recognition
- computational biology, etc.

Key role in recent successes in corporate applications such as:

- self driving vehicles
- speech interfaces
- conversational agents, assistants (Alexa, Cortana, Siri, Google assistant,…)
- superhuman game playing

Many more underway (to perform prediction and/or control):

- personalized/automated medicine
- chemistry, robotics, materials science, etc.

Deep learning … why now?

1. Reason #1: lots of data

- many significant problems can only be solved at scale
- lots of data enable us to model very complex rich models solving more realistic tasks;

2. Reason #2: parallel kid computational resources (esp. GPUs, tensor processing units)

- platforms/systems that support running deep (machine) learning algorithms at scale in parallel

3. Reason #3: large models are actually easier to train

- contrary to small, rigid models, large, richer and flexible models can be successfully estimated even with simple gradient based learning algorithms like stochastic gradient descent.

4. Reason #4: flexible neural “lego pieces”

- common representations, diversity of architectural choices
- we can easily compose these models together to perform very interesting computations. In other words, they can serve as very flexible computational Lego pieces that can be adapted overall to perform a useful role as part of much larger, richer computational architectures.

Why #3 (large models are easier to learn) ?

We can think about the notions of width (number of units in a layer) and depth (number of layers). Small models (low width and low depth) are quite rigid and don’t allow for a good abstraction of reality i.e. learning the underlying structures based on observations. Large models can use more width and more depth to generate improved abstractions of reality i.e. improved learning.

See also the conclusion of the video on the next segment: “Introducing redundancy will make the optimization problem that we have to solve easier.”

8.6. Hidden Layer Models

Let’s now specifically consist a deep neural network consisting of the input layer $\mathbf{x}$, a single hidden layer $\mathbf{z}$ performing the aggregation $z_i = \sum_j x_j * w_{j,i} + w_{0,i}$ and using $tanh(\mathbf{z})$ as activation function f, and the output node with a linear activation function.

We can see each layer (one in this case) as a “box” that takes as input $\mathbf{x}$ and return output $\mathbf{f}$, mediated trough its weights $\mathbf{W}$, and the output layer as those box taking $\mathbf{f}$ as input and returning the final output f, mediated trough its weights $\mathbf{W}'$

And we are going to try to understand how this computation changes as a function of W and W'.

What these hidden units are actually doing ? Since they are like linear classifiers, they take linear combination of the inputs, pass through that nonlinear function, we can also visualize them as if they were linear classifiers with norm equal to $\mathbf{w}$.

The difference is that instead of having a binary output (like in $sign(\mathbf{w} \cdot \mathbf{x})$) we have now $f(\mathbf{w} \cdot \mathbf{x})$. f typically takes the form of $tanh(\mathbf{w} \cdot \mathbf{x})$ for the hidden layers that we will study, whose output range is (-1,+1). More the point is far from the boundary, more $tanh()$ move toward the extremes, with a speed that is proportional to the norm of $\mathbf{w}$.

If we have (as it is normally) multiple nodes per layer, we can thing on a series of linear classifiers on the same $depth_{i-1}$ space, each one identified by its norm $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{depth_i}$.

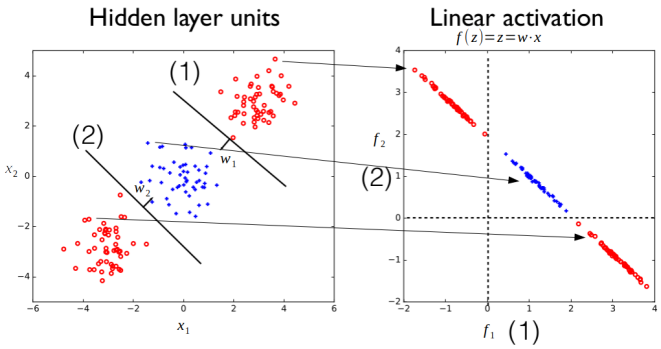
Given a neural network with one hidden layer for classification, we can view the hidden layer as a feature representation, and the output layer as a classifier using the learned feature representation.

There're also other parameters that will affect the learning process and the performance of the model, such as the learning rate and parameters that control the network architecture (e.g. number of hidden units/layers) etc. These are often called hyper-parameters.

Similar to the linear classifiers that we covered in previous lectures, we need to learn the parameters for the classifier. However, in this case we also learn the parameters that generate a representation for the data. The dimensions and the hyper-parameters are decided with the structure of the model and are not optimized directly during the learning process but can be chosen by performing a grid search with the evaluation data or by more advanced techniques (such as meta-learning).

2-D Example

Let's consider as example a case in 2-D where we have a cloud of negative points (red) in the bottom-left and top-right corners and a cloud of positive points (blue) in the center, like in the following chart (left side):

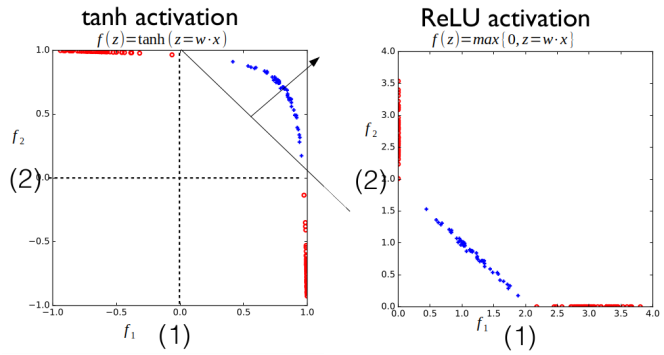


Such problem is clearly non linearly separable.

The chart on the right depicts the same points in the space resulting from the application of the two classifiers, using just a linear activation, i.e. the dot product between w and x.

The appearance that it draws exactly as a line derives from the fact that the two planes are parallel, but the general idea is that still we have a problem that is not separable, as any linear transformation of the feature space of a linearly in-separable classification problem would still continue to remain linearly inseparable.

However, when we use $tanh(x)$ as activation function we obtain the output as depicted in the following chart (left), where the problem now is clearly linearly separable.



This is really where the power of the hidden layer lies. It gives us a transformation of the input signal into a representation that makes the problem easier to solve.

Finally we can try the ReLU activation ($f(z) = max\{0, z\}$): in this case the output is not actually strictly linearly separable.

So this highlights the difficulty of learning these models. It's not always the case that the same non-linear transformation casts them as linearly separable.

However if we flip the planes directions, both the tanh and the ReLU activation results in outputs that become linearly separable (for the ReLU, all positive points got mapped to the (0,0) point).

What if we chose the two planes as random (i.e. $\mathbf{w}_1$ and $\mathbf{w}_2$ are random) ? The resulting output would likely not be linearly separable. However if we introduce redundancy, e.g. using 10 hidden units for an original two dimensional problem, the problem would likely become linearly separable

even if these 10 planes are chosen at random. Notice this is quite similar to the systematic expansion that we did earlier, in terms of polynomial features.

So introducing redundancy here is actually helpful. And we'll see how this is helpful also when we are actually learning these hidden unit representations from data. Introducing redundancy will make the optimization problem that we have to solve easier.

Summary

- Units in neural networks are linear classifiers, just with different output non-linearity
- The units in feed-forward neural networks are arranged in layers (input, one or plus hidden, output)
- By learning the parameters associated with the hidden layer units, we learn how to represent examples (as hidden layer activations)
- The representations in neural networks are learned directly to facilitate the end-to-end task
- A simple classifier (output unit) suffices to solve complex classification tasks if it operates on the hidden layer representations

The outward layer is their prediction that we actually want. And the role of the hidden layers is really to adjust their transformation, adjust their computation in such a way that the output layer will have an easier task to solve the problem.

The next lecture will deal with actually learning these representations together with the final classifier.

Lecture 9. Feedforward Neural Networks, Back Propagation, and Stochastic Gradient Descent (SGD)

9.1. Objectives

At the end of this lecture, you will be able to

- Write down recursive relations with back-propagation algorithm to compute the gradient of the loss function with respect to the weight parameters.
- Use the stochastic descent algorithm to train a feedforward neural network.
- Understand that it is not guaranteed to reach global (only local) optimum with SGD to minimize the training loss.
- Recognize when a network has overcapacity .

9.2. Back-propagation Algorithm

We start now to consider how to learn from data (feedforward) neural network, that is estimate its weights.

We recall that feed-forward neural networks, with multiple hidden layers mediating the calculation from the input to the output, are complicated models that are trying to capture the representation of the examples towards the output unit in such a way as to facilitate the actual prediction task.

It is this representation learning part – we're learning the feature representation as well as how to make use of it – that makes the learning problem difficult. But it turns out that a simple stochastic gradient descent algorithm actually succeeds in finding typically a good solution to the parameters, provided that we give the model a little bit of overcapacity. The main algorithmic question, then, is how to actually evaluate that gradient, the derivative of the Loss with respect to the parameters. And that can be computed efficiently using so-called back propagation algorithm.

Given an input X, a neural network characterised by the overall weight set W (so that its output, a scalar here for simplicity, is $f(X;W)$), and the "correct" target vector Y, the task in the training step is find the W that minimise a given loss function $\mathcal{L}(f(X;W),Y)$. We do that by computing the derivative of the loss function for each weight $w_{i,j}^l$ applied by the j-th unit at each l-th layer in relation to the output of the i-th node at the previous $l-1$ layer: $\frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$.

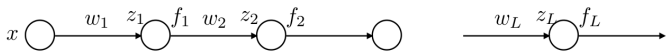
Then we simply apply the SDG algorithm in relation to this $w_{i,j}^l$:

$$w_{i,j}^l \leftarrow w_{i,j}^l - \eta * \frac{\partial \mathcal{L}(f(X;W),Y)}{\partial w_{i,j}}$$

The question turns now on how do we evaluate such gradient, as the mapping from the weight to the final output of the network can be very complicated (so much for the weights in the first layers!).

This computation can be efficiently done using a so called **back propagation algorithm** that essentially exploits the chain rule.

Let's take as example a deep neural network with a single unit per node, with both input and outputs as scalars, as in the following diagram:



Let's also assume that the activation function is $\tanh(z)$, also in the last layer (in reality the last unit is often a linear function, so that the prediction is in $\mathbb{R}$ and not just in $(-1,+1)$), that there is no offset parameter and that the specific loss function for each individual example is $Loss = \frac{1}{2}(y - f_L)^2$.

Then, for such network, we can write $z_1 = xw_1$ and, more in general, for $i = 2, \ldots, L$: $z_i = f_{i-1}w_i$ where $f_{i-1} = f(z_{i-1})$.

So, specifically, $f_1 = tanh(z_1) = tanh(xw_1)$, $f_2 = tanh(z_2) = tanh(f_1 w_2)$,...

In order to find $\frac{\partial Loss}{\partial w_i}$ we can use the chain rule by start evaluating the derivative of the loss function, then evaluate the last layer, and then eventually go backward until the first layer:

$$\frac{\partial Loss}{\partial w_1} = \frac{\partial f_1}{\partial w_1} * \frac{\partial f_2}{\partial f_1} * \frac{\partial f_3}{\partial f_2} * \ldots * \frac{\partial f_L}{\partial f_{L-1}} * \frac{\partial Loss}{\partial f_L}$$

$$\frac{\partial Loss}{\partial w_1} = [(1 - tanh^2(xw_1))x] * [(1 - tanh^2(f_1 w_2))w_2] * [(1 - tanh^2(f_2 w_3))w_3] * \ldots [(1 - tanh^2(f_{L-1} w_L))w_L] * [f_L - y]$$

Now based on the nature the calculator, the fact that we evaluate the loss at the very output, then multiply by these Jacobians also highlights how this can go wrong. Imagine if these Jacobians here, the value is the derivatives of the layer-wise mappings, are very small. Then the gradient vanishes very quickly as the depth of the architecture increases. If these derivatives are large, then the gradients can also explode.

So there are issues that we need to deal with when the architecture is deep.

9.3. Training Models with 1 Hidden Layer, Overcapacity, and Convergence Guarantees

Using the SGD, the average hinge loss evolves at each iterations (or epochs), where the algorithm runs through all the training examples (in sample order), performing a stochastic gradient descent update Typically, after a few runs over the training examples, the network actually succeeds in finding a solution that has zero hinge loss.

When the problem is however complex, a neural network with just the capacity in terms of number of nodes that would be theoretically enough to find a separable solution (classify all the example correctly) may not actually arrive to such optimal classification. More in general, for multi-layer neural networks, stochastic gradient descent (SGD) is not guaranteed to reach a global optimum (but they can find a locally optimal solution, which is typically quite good).

We can facilitate the optimization of these architectures by giving them **overcapacity** (increasing the number of nodes in the layer(s)), making them a little bit more complicated than they need to be to actually solve the task.

Using overcapacity however can lead to artefacts in the classifiers. To limit these artefacts and have good models even with overcapacity one can use two tricks:

1. Consider random initialisation of the parameters (rather than start from zero). And as we learn and arrive at a perfect solution in terms of end-to-end mapping from inputs to outputs, then in that solution, not all the hidden units need to be doing something useful, so long as many of them do. The randomization inherent in the initialization creates some smoothness. And we end up with a smooth decision boundary even when we have given the model quite a bit of overcapacity.
2. Use the ReLU activation function ($max(0, z)$) rather than $tanh(z)$. ReLU is cheap to evaluate , works well with sparsity and make parameters easier to be estimated in large models.

We will later talk about regularization – how to actually squeeze the capacity of these models a little bit, while in terms of units, giving them overcapacity.

Summary

- Neural networks can be learned with SGD similarly to linear classifiers
- The derivatives necessary for SGD can be evaluated effectively via back-propagation
- Multi-layer neural network models are complicated. We are no longer guaranteed to reach global (only local) optimum with SGD
- Larger models tend to be easier to learn because their units only need to be adjusted so that they are, collectively, sufficient to solve the task

An example of fully-connected feed-forward neural network in Julia can be found on https://github.com/sylvaticus/lmlj/blob/master/fnn.jl

Lecture 10. Recurrent Neural Networks 1

10.1. Objective

Introduction to recurrent neural networks (RNNs)

At the end of this lecture, you will be able to:

- Know the difference between feed-forward and recurrent neural networks(RNNs).
- Understand the role of gating and memory cells in long-short term memory (LSTM).
- Understand the process of encoding of RNNs in modeling sequences.

10.2. Introduction to Recurrent Neural Networks

In this and in the next lecture we will use neural networks to model sequences, using so called **recurrent neural networks** (*RNN*).

This lecture introduces the topic: the problem of modelling sequences, what are RNN, how they relate to the feedforward neural network we saw in the previous lectures and how to *encode* sequences into vector representations. Next lecture will focus on how to *decode* such vectors so that we can use them to predict properties of the sequences, or what comes next in the sequence.

Exchange rate example

Let's consider a time serie of the exchange rate between US Dollar and the Euro, with an objective of predict its value in the future.

We already saw how to solve this kind of problem with linear predictors or feedforward neural networks.

In both case the first task is to compile a feature vector, for example of the values of the exchange rate at various times, for example, at time $t-1$ to $t-4$ for a prediction at time t.

As we have long time-serie available we can use some of the observation as training, some as validation and some as test (ideally by random sampling).

Language completion example

In a similar way we can see a text as a sequence of words, and try to predict the next world based on the previous words, for example using the previous two words, where each of them is coded as a 1 in a sparse array of all the possible words (à la bag of words approach).

Limitations of these approaches

While we could use linear classifiers or feedforward neural networks to predict sequences we are still left with the problem of how much "history" look at in the creation of the feature vector and the fact that this history length may be variable, for example there may be words at the beginning of the sentence that are quite relevant for predicting what happens towards the end, and we would have to somehow retain that information in the feature representation that we are using for predicting what happens next.

Recurrent Neural Networks can be used in place of feedforwrd neural networks to learn not only the weight given the feature vectors, but also how to encode in the first instance the history into the feature vector.

10.3. Why we need RNNs

The way we chose how to encode data in feature vectors depends on the task we need. For example, sentiment analysis, language translation, and next word suggestion all requires a different

feature representation as they focus on different parts of the sentence: while sentiment analysis focuses on the holistic meaning of a sentence, translation or next word suggestion focuses instead more on individual words.

While in feed-forward networks we have to manually engineer how history is mapped to a feature vector (representation) for the specific task at hand, using RNN's this task is also part of the learning process and hence automatised.

Note that very different types of objects can be encoded in feature vectors, like images, events, words or videos, and once they are encoded, all these different kind of objects can be used together.

In other words, RNNs can not only process single data points (such as images), but also entire sequences of data (such as speech or video).

While this lecture deals with **encoding**, the mapping of a sequence to a feature vector, the next lecture deals with **decoding**, the mapping of a feature vector to a sequence.

10.4. Encoding with RNN

While in feedforward neural network the input is only at the beginning of the chain and the parameter matrix W is different at each layer, in recurrent neural networks the "external input" arrives at each layer and contribute to the argument of the activation function together with the flow of information coming from the previous layer (*recurrent* refers to this state information that, properly transformed, flows across the various layers).

One simple implementation of each layer transformation (that here we can see it as an "update") is hence (omitting the offset parameters):

$$s_t = \tanh(W^{s,s} s_{t-1} + W^{s,x} x_t)$$

Where:

- s_{t-1} is a $m \times 1$ vector of the old "context" or "state" (the data coming from the previous layer)
- $W^{s,s}$ is a $m \times m$ matrix of the weights associated to the existing state, whose role is to deciding what part of the previous information should be keep (and note that this is not changing in each layer.). Can also be interpreted as giving how the state would evolve in absence of any new information.
- x_t is a $d \times 1$ feature representation of the new information (e.g. a new word)
- $W^{s,x}$ is a $m \times m$ weights whose role is deciding how to take into account the new information, so that the result of wx multiplication is specific to each new information arriving;
- $tanh(\cdot)$ is the activation function (to be applied elementwise)
- s_t is a $m \times 1$ vector of the new "context" or "state" (the updated state with the new information taken into account)

RNN have hence a number of layers equal to the data in the sequence, like the words in a sentence. So there is a single, evolving, NN for the whole sequence rather than a different NN for each element of the sequence. The initial state (S_0) is a vector of m zeros.

Note that this parametric approach let the way we introduce new data ($W^{s,x}$) to adjust to the way we use the network, i.e. to be learned according to the specific problem on hand…

In other words, we can adjust those parameters to make this representation of a sequence appropriate for the task that we are trying to solve.

10.5. Gating and LSTM

Learning RNNs

Learning a RNN is similar to learning a feedforward neural network: the first task is to define a Loss function and then find the weights that minimise it, for example using a SGD algorithm where the gradient with respect to the weights is computed using back-propagation. The fact that the parameters are shared in RNN's means that we add the contribution of each of the suggested modifications for parameters at each of these positions where the transformation is flagged. One problem of simple RNN models is that the gradient can vanish or explode. Here even more than in feedforward NN, as the sequences can be quite long and we apply this transformation repeatedly. In real cases, the design of the transformation can be improved to counter this issue.

Simple gated RNN

Further, in simple RNN the state information is *always* updated with new data, so far away information is easily "forget". It is often helpful to retain (or learn to retain) some control over what is written over and what is incorporated as new information. We can apply this control over what we overwrite and what instead we retain from the old state when meet new data using a form of RNN called **gated recursive neural networks**. Gated networks adds *gates* controlling the flow of information. Its simplest implementation can be written as:

$$g_t = \text{sigmoid}(W^{g,s} s_{t-1} + W^{g,x} x_t) = \frac{1}{1+e^{-(W^{g,s} s_{t-1} + W^{g,x} x_t)}}$$

$$s_t = (1 - g_t) \odot s_{t-1} + g_t \odot \tanh(W^{s,s} s_{t-1} + W^{s,x} x_t)$$

Where the first equation defines a **gate** responsible to filter in the new information (trough its own parameters to be learn as well) and results in a continuous value between 0 and 1.

The second equation then uses the gate to define, for each individual value of the state vector (the $\odot$ symbol stands for element-wise multiplication), how much to retain and how much to update with the new information. For example, if $g_t[2]$ (the second element of the gate at the t transformation) is 0 (an extreme value!), it means we keep in that transformation the old value of the state for the second element, ignoring the transformation deriving from the new information.

Long Short Term Memory neural networks

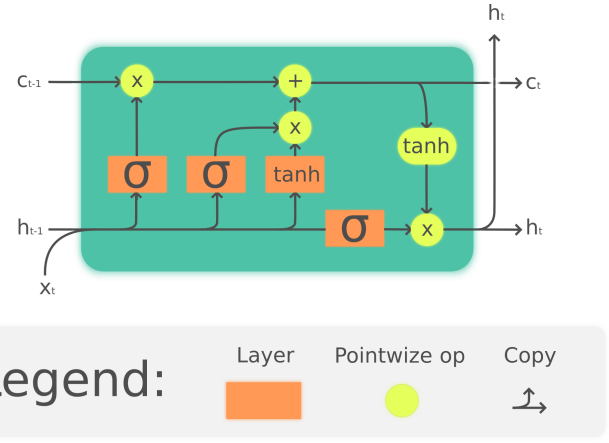
Real in-use gated RNN are even more complicated. In particular, **Long Short Term Memory** (recursive neural) networks (shortered as LSTM) are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series and have the following gates defined:

$$\begin{aligned}
f_t &= sigmoid(W^{f,h} h_{t-1} + W^{f,x} x_t) && \text{forget gate} \\
i_t &= sigmoid(W^{i,h} h_{t-1} + W^{i,x} x_t) && \text{input gate} \\
o_t &= sigmoid(W^{o,h} h_{t-1} + W^{o,x} x_t) && \text{output gate} \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W^{c,h} h_{t-1} + W^{c,x} x_t) && \text{memory cell} \\
h_t &= o_t \odot tanh(c_t) && \text{visible state}
\end{aligned}$$

The input, forget, and output gates control respectively how to read information into the memory cell, how to forget information that we've had previously, and how to output information from the memory cell into a visible form.

The "state" is now represented collectively by the memory cell c_t 'sometimes indicated as *long-term memory*) and its "visible" state h_t (sometimes indicated as *working memory* or *hidden state*).

LSTM cells have hence the following diagram:



The memory cell update is helpful to retain information over a longer sequences. But we keep his memory cell hidden and instead only reveal the visible portion of this tape. And it is this h_t then at the end of the whole sequence applying this box along the sequence that we will use as the vector representation for the sequence.

On the LSTM topic one could also look at these external resources:

- http://blog.echen.me/2017/05/30/exploring-lstms/
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/

- https://machinelearningmastery.com/handle-long-sequences-long-short-term-memory-recurrent-neural-networks/

Key things

- Neural networks for sequences: encoding
- RNNs, unfolded
  - state evolution, gates
  - relation to feed-forward neural networks
  - back-propagation (conceptually)
- Issues: vanishing/exploding gradient
- LSTM (operationally)

In other words:

- RNN's turn sequences into vectors (encoding)
- They can be understood as feed-forward neural networks whose architecture has changed from one sequence to another
- Can be learned with back-propagation of the error signal like for feedforward NN
- They too suffer of vanishing or exploding gradient issues
- Specific architectures such as the LSTM maintain a better control over the information that's retained or updated along the sequence, and they are therefore easier to train

Lecture 11. Recurrent Neural Networks 2

11.1. Objective

From Markov model to recurrent neural networks (RNNs)

- Formulate, estimate and sample sequences from Markov models.
- Understand the relation between RNNs and Markov model for generating sequences.
- Understand the process of decoding of RNN in generating sequences.

11.2. Markov Models

Outline

- Modeling sequences: language models
  - Markov models
  - as neural networks
  - hidden state, Recurrent Neural Networks (RNNs)
- Example: decoding images into sentences

While in the last lesson we saw how to transform a sentence into a vector, in a parametrized way that can be optimized for what we want the vector to do, today we're going to be talking about how to generate sequences using recurrent neural networks (decoding). For example, in the translation domain, how to unravel the output vector as a sentence in an other language.

Markov models

One way to implement prediction in a sequence is to just first define probabilities for any possible combination looking at existing data (for example, in a next word prediction - "language modelling", look at all two-word combinations in a serie of texts) and then, once we observe a case, sample the next element from this conditional (discrete) probability distribution. This is a first order Markov model.

Markov textual bigram model exemple

In this example we want to predict the next word based exclusively on the previous one.

We first learn the probability of any pair of words from data ("corpus"). For practical reasons, we consider a vocabulary V of the n more frequent words (and symbols), labelling all the others as `UNK` (a catch-all for "unknown"). To this vocabulary we also add two special symbols `<beg>` and `<end>` to mark respectively the beginning and the ending of the sentence. So a pair `(<beg>,w)` would represent a word $w \in V$ starting the sentence and `(w,<end>)` would represent the world w ending the sentence.

We can now estimate the probability for each words w and $w' \in V$ that w' follows w, that is the conditional probability that next word is w' given the previous one was w, by a normalised counting of the successive occurrences of the pair (w, w') (matching statistics):

$$\hat{P}(w'|w) = \frac{count(w,w')}{\sum_{w_i \in V} count(w,w_i)}$$

(we could be a bit smarter and consider at the denominator only the set of pairs whose first element is w rather than the whole $V \in n^2$)

For a bigram we would obtain a probability table like the following one:

		w_i				
		ML	course	is	UNK	<end>
w_{i-1}	<beg>	0.7	0.1	0.1	0.1	0.0
	ML	0.1	0.5	0.2	0.1	0.1
	course	0.0	0.0	0.7	0.1	0.2
	is	0.1	0.3	0.0	0.6	0.0
	UNK	0.1	0.2	0.2	0.3	0.2

At this point we can generate a sentence by each time sampling from the conditional probability mass function of the last observed word, starting with `<beg>` (first row in the table) and until we sample a `<end>`.

We can use the table also to define the probability of any given sentence by just using the probability multiplication rule. The probability of any N words sentence (including `<beg>` and `<end>`) is then $P = \prod_{i=2}^{N} P(w_i|w_{i-1})$. For example, given the above table, the probability of the sentence "Course is great" would be $0.1 * 0.7 * 0.6 * 0.2$.

Note that, using the counting approach, the model maximise the probability that the generated sequences correspond to the observed sequences in the corpus, i.e. counting corresponds here to the Maximum Likelihood Estimation of the conditional probabilities. Also, the same approach can be used to model words characted by character rather than sentences world by world.

Outline detailed

In this segment we considered language modelling, using the very simple sequence model called Markov model. In the next segments of this lecture we're going to turn those Markov models into a neural network models, first as feed-forward neural network models. And then we will add a hidden state and turn them into recurrent neural network models. And finally, we'll just consider briefly an example of unraveling a vector representation, in this case, coming from an image to a sequence.

11.3. Markov Models to Feedforward Neural Nets

We can represent the first order Markov model described in the previous segment as a feed-forward neural network,

We start by a one-hot encoding of the words, i.e. each input word would activate one unique node on the input layer of the network ($\mathbf{x}$). In other words, if the vocabulary is composed of K words, the input layer of the neural network has width K, and it is filed all with zeros except for the specific word encoded as 1.

We want as output of the neural network the PMF conditional to that specific word in the input. So the output layer has too one unit for each possible word, returning each node the probability that the next word is that specific one given that the previous one was those encoded in x: $P_k = P(w_i = k|w_{i-1})$

Given the weights of this neural network W (not to be confused with the words w), the argument of the activation function of each node of the output layer is $z_k = \mathbf{x} \cdot \mathbf{W}_k + W_{0,k}$.

These z are real numbers. To transform in probabilities (all positive, sum equal to 1) we use as activation function the non-linear Softmax function:

$$P_k = \frac{e^{Z_k}}{\sum_{j=1}^{K} e^{Z_j}}$$

(and the output layer is then called **softmax output layer**).

Advantages of neural network representation

Frexibility

We can use as input of the neural network a vector x coposed of the one-hot econding of the previous world, *plus* the vector of the one-hot encoding of the second previous word, obtaining the probability that the next word is w_k conditional to the two preceding words (roughly similar to a second order Markov model).

Further, we could also insert an hidden layer in between th input and output layers, in order to look at more complex combinations of the preceding two words, in terms of how they are mapped to the probability values over the next word.

Parsimony

For the tri-gram model above, the neural network would need a weight matrix W of $2K \times K$ parameters, i.e; a total og $2K^2$ parameters.

Using a second order Markov model would require instead K^3 parameters: we take two preceding words, and for any such combination, we predict a probability of what the next word is. So if we have here the possible words roughly speaking, then each combination of preceding words, the square root of them, and then for each of those combinations, we have to have a probability value of each of the next words. So we will look at that times the number of parameters in the tri-gram model. So roughly, the number of words to the power of 3.

As a result, the neural network representation for a tri-gram model is not as general, but it is much more feasible in terms of the number of parameters that we have to estimate. But we can always increase the complexity of the model in the neural network representation by adding a hidden layer.

11.4. RNN Deeper Dive

We can now translate the feedforward neural network in a Recursive Neural Network that accepts a *sequence* of words as input and can account for a variable history in making predictionsfor the next word rather than on a fixed number of elements (as 1 or 2 in bigrams and trigrams respectively).

The framework is similar to the RNN we saw in the previous lesson, with a state (initially set to $\mathbf{0} \in R^K$) that is updated, at each new information, with a function of the previous state and the new observed word (typically with something like $s_t = tanh(W^{s,s}s_{t-1} + W^{s,w}x_t)$). Note that x_t is the one hot encoding of the new observed word.

The difference is that in addition, at each step, we have also an output that transforms the state in probability (representing the new, conditional PMF): $p_t = softmax(W^0 s_t)$.

Note that the state here retains information of all the history of the sentence, hence the probability is conditional to all the previous history in the sentence.

In other words, while $W^{s,s}$ and $W^{s,w}$ role is to select and encode the relevant features from the previous history and the new data respectively, W^0 role is to extract the relevant features from the memorised state with the aim of making a prediction.

Finally we can have more complex structures, like LSTM networks, with forget, input and output gates and the state divided in a memory cell and a visible state. Also in this case we would however have a further transformation that output the conditional PMF as function of the visible state ($p_t = softmax(W^0 h_t)$).

Note that the training phase is done computing the average loss at the level of sentences, i.e. our labels are the full sentences, and are these that are compared in the loss function with those obtained by sampling the PMFs resulting from the RNN. While in *training* we use the true words specified as input for the next time step, in *testing* instead we let the neural network to predict the sentence on its own, using the sampled output at one time step as the input for the next step.

11.5. RNN Decoding

The question is how to use now these (trained) RNN models to generate a sentence from a given encoded state (e.g. in testing)?

The only difference is that the initial state is not a vector of zero, but the “encoded sentence”. We just start with that state and the `<beg>` symbol as x and then let the RNN produce a PDF, sample from

that and use that sampled data as the new x for the next step and so on until we sample an <end> .

We don't just have to take a vector from a sentence and translate it into another sentence. We can take a vector of representation of an image that we have learned or are learning in the same process and translate that into a sentence.

So we could take an image, translate into a state using a convolutional neural network and then this state is the initial state of an other neural network predicting the caption sentence associated to that image (e.g. "A person riding a motorcycle on a dirt road")

Key summary

- Markov models for sequences
  - how to formulate, estimate, sample sequences from
- RNNs for generating (decoding) sequences
  - relation to Markov models (how to translate Markov models into RNN)
  - evolving hidden state
  - sampling from the RNN at each point
- Decoding vectors into sequences (once we have this architecture that can generate sequences, such as sentences, we can start any vector coming from a sentence, image, or other context and unravel it as a sequence, e.g. as a natural language sentence.)

Homework 4

Lecture 12. Convolutional Neural Networks (CNNs)

12.1. Objectives

At the end of this lecture, you will be able to

- Know the differences between feed-forward and Convolutional neural networks (CNNs).
- Implement the key parts in the CNNs, including *convolution*, *max pooling* units.
- Determine the dimension of each channel in different layers with a given CNNs.

12.2. Convolutional Neural Networks

CNNs Outline:

- why not use unstructured feed-forward models?
- key parts: convolution, pooling
- examples

The problem: image classification, i.e. multi-way classification of images mapping to a specific category (e.g. x is a given image, the label is "dog" or "car").

Why we don't use "normal" feed-forward neural networks ?

1. *It would need too many parameters*. If an image is mid-size resolution 1000×1000, each layer would need a weight matrix connecting all these 10^6 pixel in input with 10^6 pixel in output, i.e. 10^12 weights
2. *Local learning (in the image) would not extend to global learning*. If we train the network to recognise cars, and it happens that our training photos have the cars all in the bottom half, then the network would not recognise a car in the top half, as these would activate different neurons.

We solve these problems employing specialised network architectures called **convolution neural network**.

In these networks the layer l is obtained by operating over the image at layer $l-1$ a small **filter** (or **kernel**) that is slid across the image with a step of 1 (typically) or more pixels at the time. The step is called **stride**, while the whole process of sliding the filter throughout the whole image can be mathematically seen as a **convolution**.

So, while we slide the filter, at each location of the filter, the output is composed of the dot product between the values of the filter and the corresponding location in the image (both vectorised), where the values of the filters are the weigths that we want to learn, and they remain constant across the sliding. If our filter is a 10×10 matrix, we have only 10 weights to learn by layer (plus one for the

offset). Exactly as for feedforward neural networks, then the dot product is passed through an activation function, here typically the `ReLU` function ($max(0, x)$) rather than `tanh`.

For example, given an image $x = \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 3 & 1 & 4 & 1 & 1 \\ 1 & 3 & 1 & 2 & 2 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \end{bmatrix}$ and filter weights $w = \begin{bmatrix} 1 & -2 & 0 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix}$, then the output of the filter z would be $\begin{bmatrix} 8 & -3 & 6 \\ 4 & -3 & 5 \\ -3 & 5 & -2 \end{bmatrix}$. For example, the element of this matrix $z_{2,3} = 5$ is the result of the sum of the scalar multiplication between $x' = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 1 \end{bmatrix}$ and w.

Finally, the output of the layer would be (using ReLU) $\begin{bmatrix} 8 & 0 & 6 \\ 4 & 0 & 5 \\ 0 & 5 & 0 \end{bmatrix}$.

For example you can obtain the above with the following Julia code:

```
ReLU(x) = max(0,x)
x = [1 1 2 1 1;
     3 1 4 1 1;
     1 3 1 2 2;
     1 2 1 1 1;
     1 1 2 1 1]
w = [ 1 -2  0;
      1  0  1;
     -1  1  0]
(xr,xc) = size(x)
(wr,wc) = size(w)
z = [sum(x[r:r+wr-1,c:c+wc-1] .* w) for c in 1:xc-wc+1 for r in 1:xr-wr+1] # Julia
is column mayor
u = ReLU.(z)
final = reshape(u, xr-wr+1, xc-wc+1)
```

(You can play with the above code snippet online without registration on https://bitly.com/convLayer)

You can notice that, applying the filter, we obtain a dimensionality reduction. This reduction depends on both the dimension of the filter and the stride (sliding step). In order to avoid this, a padding of one or more zeros can be applied to the image in order to keep the same dimensions in the output (in the above example a padding of one zeros on both sides - and both dimensions - would suffice).

Because the weight of the filters are the same, it doesn't really matter where the object is learned, in which part of the image. The lecture explains this with a mushroom example: ff the mushroom is in a different place, in a feed-forward neural network the weight matrix parameters at that location need to learn to recognize the mushroom anew. With convolutional layers, we have translational invariance as the same filter is passed over the entire image. Therefore, it will detect the mushroom regardless of its location.

Still, it is often convenient to operate some **data augmentation** to the training set, that is to add slightly modified images (rotated, mirrored…) in order to improve this translational invariance.

A further way to improve translational invariance, but also have some dimensionality reduction, it called **pooling** and is adding a layer with a filter whose output is the `max` of the corresponding area in the input. Note that this layer would have no weights! With pooling we contribute to start separating what is in the image from where it is in the image, that is pooling does a fine-scale, local translational invariance, while convolution does more a large-scale one.

Keeping the output of the above example as input, a pooling layer with a 2×2 filter and a stride of 1 would result in $\begin{bmatrix} 8 & 6 \\ 5 & 5 \end{bmatrix}$.

In a typical CNN, these convolutional and pooling layers are repeated several times, where the initial few layers typically would capture the simpler and smaller features, whereas the later layers would

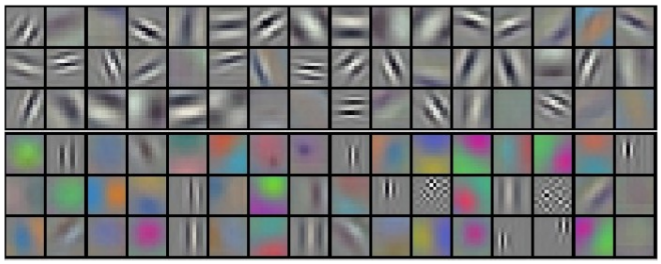
use information from these low-level features to identify more complex and sophisticated features, like characterisations of a scene. The learned weights would hence specialise across the layers in a sequence like edges -> simple parts-> parts -> objects -> scenes.

Concerning the topic of CNN, see also the superb lecture of Andrej Karpathy on YouTube (here or here).

12.3. CNN - Continued

CNN's architectures combine these type of layers successively in a variety of different ways.

Typically, one single layer is formed by applying multiple filter, not just one. This is because we want to learn different kind of features… for example one filter will specialize to catch vertical lines in the image, an other obliques ones… and maybe an other different colours. Indeed in real implementation, both the image and the filter have normally a further dimensions to account for colours, so they can modelled as volumes rather than areas:



96 convolutional filters on the first layer (filters are of size 11x11x3, applied across input images of size 224x224x3). They all have been learned starting from random initialisation.

So in each layer we are going to map the original image into multiple feature maps where each feature map is generated by a little weight matrix, the filter, that defines the little classifier that's run through the original image to get the associated feature map. Each of these feature maps defines a channel for information.

I can then combine these convolution, looking for features, and pooling, compressing the image a little bit, forgetting the information of where things are, but maintaining what is there.

These layers are finally followed by some "normal", "fully connected" layers (*à la* feed-forward neural networks) and a final `softmax` layer indicating the probability that each image represents one of the possible categories (there could be thousand of them).

The best network implementation are tested in so called "competitions", like the yearly ImageNet context.

Note that we can train this networks exactly like for feedformard NN, defining a loss function and finding the weights that minimise the loss function. In particular we can apply the stochastic gradient descendent algorithm (with a few tricks based on getting pairs of image and the corresponding label), where the gradient with respect to the various parameters (weights) is obtained by backpropagation.

Take home

- Understand what a convolution is;
- Understand the pooling, that tries to generate a slightly more compressed image forgetting where things are, but maintaining information about what's there, what was activated.

Recitation: Convolution/Cross Correlation:

Convolution (of continuous functions): $(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$

Cross-correlation: $(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$ (i.e. like convolution, but where the g function is not reversed/flipped)

In neural network we use the cross-correlation rather than the convolution. Indeed, in such context, f is the data signal and g is the filter. But the filter needs to be learn, so if it is expressed straight or reversed, doesn't really matter, so we just use the cross-correlation and save one operation.

1-D Discrete version

Cross correlation of discrete functions: $(f * g)(t) := \sum_{\tau=-\infty}^{\infty} f(\tau)g(t+\tau)$

The cross-correlation $h = (f * g)(t)$ for discrete functions can be pictured in the following scheme, where the output is the cross product of the values of f and g in the same column, and where out of domain values are padded with zeros.

```
f:           1 2 3
...
       0| 1 2
h(t=0) 2|   1 2
h(t=1) 5|     1 2
h(t=2) 8|       1 2
h(t=3) 3|         1 2
h(t=4) 0|           1 2
...
```

2-D Discrete version

In 2-D the cross correlation becomes: $(f * g)(x, y) := \sum_{\tau_1=-\infty}^{\infty} \sum_{\tau_2=-\infty}^{\infty} f(\tau_1, \tau_2) g(\tau_1 + x, \tau_2 + y)$.

Graphically, it is the sliding of the filter (first row, left to right, second row, left to right, …) that we saw in the previous segment.

Project 3: Digit recognition (Part 2)

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 04 - Unsupervised Learning

Lecture 13. Clustering 1

13.1. Unit 4: Unsupervised Learning

In this unit we start talking of problems where we don't have labels, where we need to find a structure in the data itself. For example, Googe news provide clusters of stories over the same event. It introduce a structure in the feed of news stories.

We start looking at hard assignment clustering algorithms, where we assume that each data point can belong to just a single cluster of (somehow to define) "similar" points.

We will then move to recognise data structures where each point can belong to multiple clusters, trying to discover the different components embedded in the data and looking at the likelihood a certain point was generated by that component.

And we will introduce generative model, which provide us the probability distribution over the points. We will look at multinomials, Gaussian and mixture of Gaussians. And by the end of this unit, we will get to a very powerful unsupervised learning algorithm for uncovering then, the line structure, which is called the expectation–maximization (EM) algorithm.

We will finally practice both clustering and the EM algorithm in a Netflix-recommandation project in Project 4.

13.2. Objectives

- Understand the definition of clustering
- Understand clustering cost with different similarity measures
- Understand the K-means algorithm

13.3. Introduction to Clustering

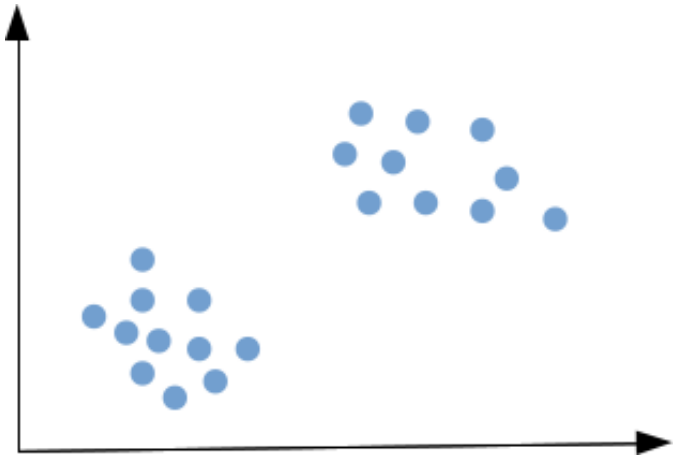
The topic in this unit is unsupervised machine-learning algorithms. In unsupervised learning, we will still have a training set, but we will not have a label like in supervised learning.

So in this case, we are provided with just a set of feature vectors, and we are trying to learn some meaningful structure which would represent this set. In this lesson in particolar we will cover *clustering* algorithms.

We will first start from the definition of a cluster.

While in a *classification* problem we already have the possible categories and the label associated to it in the training data, in clustering we need to find a structure in the data and associate it with features of the data itself, in order to predict the category from the data itself without the need for a explicit label.

For example take this chart:



It is intuitive that the points belong to two different categories, in this case based on their position on the (x,y) axis, even if the points have no label (colour) associated to them.

Similarly, in many cases the meaningful structure is actually quite intuitive in the context of a specific application.

Find a structure is however not enough. We need to be able to describe how the structure we found is good in representing the data, how good is the cluster, the partitioning of the data (the “clustering cost”). And we will need for that a measure of similarity between points.

We will finish the lecture dealing with the K-means algorithm, an important and widely used algorithm which actually can take any set of training point and find a partitioning to K clusters.

To conclude the segment with an example, let’s consider again Google news. In google News we have both an example of classification and one of clustering.

The grouping of the news in one of the categories (sport, science, business,…) is a categorisation task. The categories are fixed, and someone has trained the algorithm manually setting the categories of each news for a training set, so that now it can be predicted automatically.

The grouping of news around individual new events (the covid-19, the brexit, the US elections…) is instead a clustering problem: here the categories are not predefined and there has been no training with manually label all possible events.

What do we need for such task?

- We first need to provide a representation of the news stories, for example employing a *bag-of-words* approach, giving a (sparse) vector representation to each word;
- Second, we need to decide how to compare each of these representations, in this case vectors;
- Finally we need to implement the clustering algorithm itself, based on the pairwise measure of proximity found in step two, or, for example, selecting one representative story to show as a central story in the news.

13.4. Another Clustering Example: Image Quantization

If the previous example was dealing with text clustering for visualisation purposes, let’s now deal with image clustering for compression purposes, and more specifically for creating a colour palette for the image, a form of **image quantisation** (“quantisation” refers to the process of mapping input values from a large setto output values in a countable smaller set, often with a finite number of elements).

A typical 1024 x 1024 pixel image coded in 24 bits per pixel (8 bits per each of the RGB colours) would size 1024x1024x24 = 25165824 bits = 3 MB.

If we create a palette restricted to only 32 colours, and represent each pixel with one of the colours in this palette, the image then would size only 1024x1024x5+5*24 = 5243000 bits = 640.01 KB (this by the way is a compression technique used in PNG and GIF images).

Note that the image quality depends from the number of colours of the palette we choose, i.e. the number of clusters K. While cartoon, screenshots, logos would likely require very small K to appear of a quality similar to the original image, for photos we would likely need higher K to achieve the same quality.

Still, to get the compressed image we need to undergo the three steps described in the previous segment. We need to represent the image (as a vector of pixels with a colour code). The second step is to find some similarity measure between colours, and finally we can run the clustering algorithm to replace each pixel 24 bits representation with the new 5 bits one, choosing a map from the 12777216 original colours (24 bits) to the 32 “representative ones” (also to be found by the algorithm!).

13.5. Clustering Definition

There are two concepts related to a cluster. The first one is the partitioning of the data, how the different data records are distributed among the clusters. The second one is the representativeness, how we aggregate the cluster members in a representative element per each cluster.

Partitioning

Let’s $S_n = \{x^{(i)} | i = 1, \ldots, n\}$ be the set of n feature vectors indexed by i and $C_K = \{C_j | j = 1, \ldots, K\}$ being the set of K clusters containing the *indexes* of the data points,

then $C_1, \ldots, C_j, \ldots, C_K$ forms a K-partition of the data if, at the same time, their union contain all the data indexes and their intersection is empty: $\cup(C_1, \ldots, C_j, \ldots, C_K) = \{1, 2, \ldots, n\}$ and $\cap(C_1, \ldots, C_j, \ldots, C_K) = \{\}$

This partitioning define a so-called **hard clustering**, where every element just belongs to one cluster.

Representativeness

The representativeness view of clustering is the problem of how to select the feature vector to be used as representative of the cluster, like the new story to put as header on each event in Google News or the colour to be use for each palette item. Note that the representative element for each cluster doesn't necessarily be one existing element of the cluster set, can be also a combination of them, like the average.

We will see later in this lesson what is the connection between these two views. Clearly they are connected, because if you know what is your partitioning, you may be able to guess who is the representative. If you know who is a representative, you may be able to guess who is a constituent, but we will see how we can actually unify these two views together.

Note that finding the optimal number of K is itself another problem, as K is not part of the clustering algorithm, it is an input (parameter) to it. That is, K is a hyperparameter of the clustering algorithm.

13.6. Similarity Measures-Cost functions

Given the same set of feature vectors, we can have multiple clustering results, multiple way to partition the set. We need to define a "cost" to each possible partition of our data set, in order to rank them (and find the one that minimise the cost). We will make the assumption that the cost of a given partition is the sum of the cost of its individual clusters, where we can intuitively associate the cost to some measure of the cluster heterogeneity, like (a) the cluster diameter (the distance between the most extreme feature vectors, i.e. the outliers), (b) the average distance or © (what we will use) the sum of the distances between every member and z_j, the representative vector of cluster C_j.

In turn we have to choose which metric we use to measure such distance. We have several approaches, among which:

- cosine distance, $dist(x^{(i)}, x^{(j)}) = \frac{x^{(i)} \cdot x^{(j)}}{||x^{(i)}|| * ||x^{(j)}||}$
- square euclidean distance, $dist(x^{(i)}, x^{(j)}) = ||x^{(i)} - x^{(j)}||^2$

The cosine distance has the merit of being invariant to the magnitude of the vectors. For example, in terms of the Google News application, if we choose cosine to compare between two vectors representing two different stories, this measurement will not take it into account how long are the stories. On the other hand, the Euclidean distance would be sensitive to the lengths of the story.

While cosine looks at the angle between vectors (thus not taking into regard their weight or magnitude), euclidean distance is similar to using a ruler to actually measure the distance (and so cosine is essentially the same as euclidean on normalized data)

Cosine similarity is generally used as a metric for measuring distance when the magnitude of the vectors does not matter. This happens for example when working with text data represented by word counts. We could assume that when a word (e.g. `science`) occurs more frequent in document 1 than it does in document 2, that document 1 is more related to the topic of science. However, it could also be the case that we are working with documents of uneven lengths (Wikipedia articles for example). Then, `science` probably occurred more in document 1 just because it was way longer than document 2. Cosine similarity corrects for this (from https://cmry.github.io/notes/euclidean-v-cosine).

While we'll need to understand which metric is most suited for a specific application (and where it matters in the first instance), for now we will employ the Euclidean distance.

Given these two choices, our cost function for a specific partition becomes:

$$cost(C_1, \ldots, C_j, \ldots C_Z, z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}) = \sum_{j=1}^{Z} \sum_{i \in C_j} ||x^{(i)} - z^{(j)}||^2$$

Note that for now we are giving this cost function as a function of both the partition and the representative vectors, but when we will determine how to compute the representative vectors z_j endogenously, this will be a function of the partition alone.

13.7. The K-Means Algorithm: The Big Picture

How to find the specific partition that minimise the cost ? We can't iterate over all partitions, measure the cost and select the smaller one, as the number of (non empty) k-partitions of a set of n elements is huge.

To be exact it is known as the "Stirling numbers of the second kind" and it is equal to:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

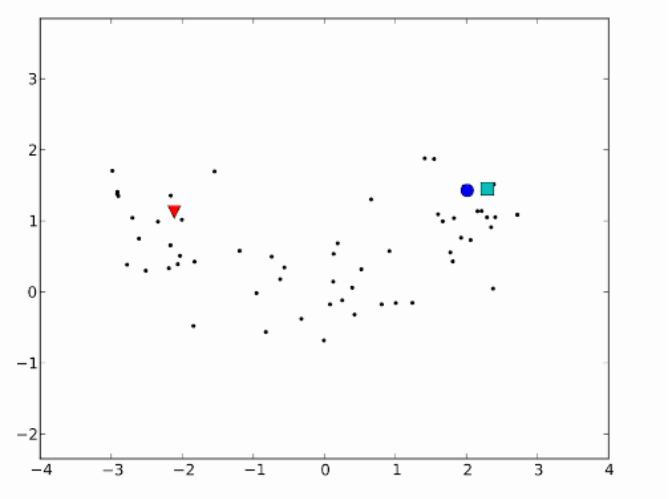
For example, $\left\{ {3 \atop 2} \right\} = 3$, but $\left\{ {100 \atop 3} \right\} \approx 8.58e+46$

The K-M algorithm helps in finding the minimal distance in a computationally feasible way.

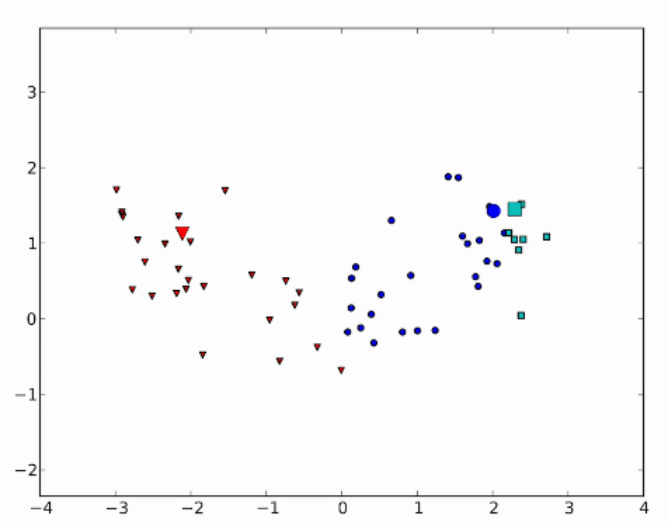
In a nutshell, it involves (a) to first randomly select k representative points. Then (b) iterate for each point to assign the point to the cluster of the closest representative (according with the adopted metric), and (c) move each representative at the center of its newly acquired cluster (where "center" depends again from the metric). Steps (b) and (c) are reiterated until the algorithm converge, i.e. the tentative k representative points (and their relative clusters) don't move any more.

Graphically:

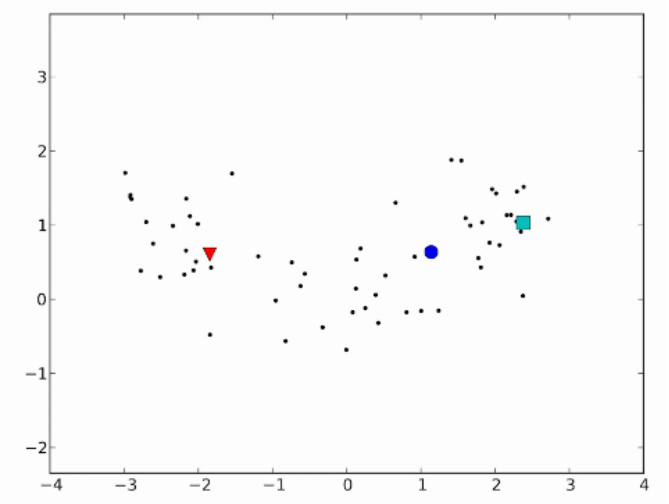
Random selection of 3 representative points in 2D:



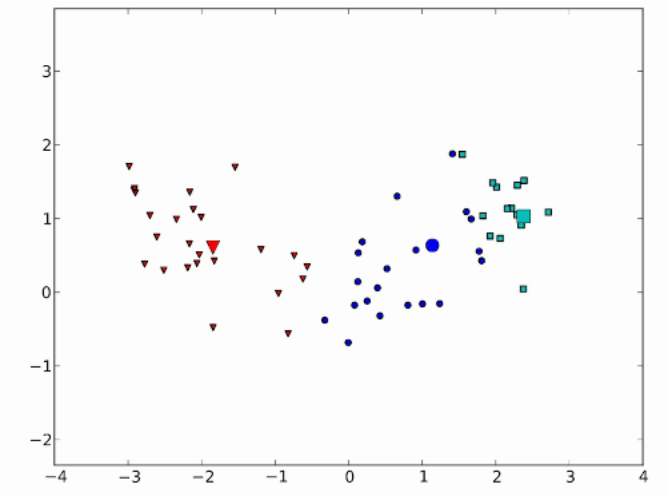
Assignment of the "constituent" of each representative:



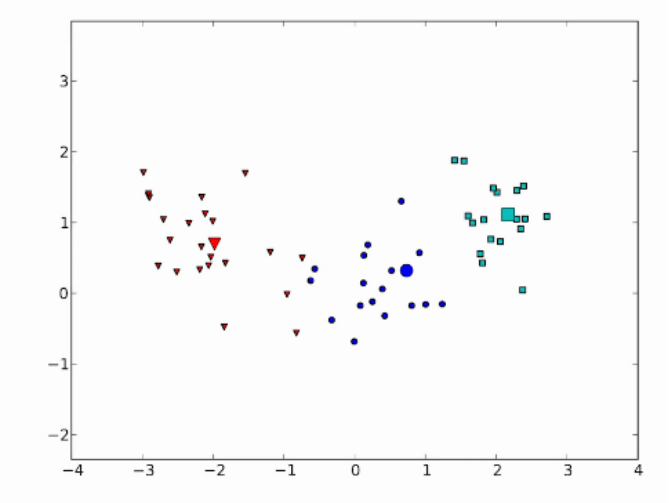
Moving the representative at the center of (the previous step) constituency:



Redefinition of the constituencies (note that some points have changed their representative):



Moving again the representatives and redefinition of the constituencies:



Let's describe this process a bit more formally in terms of the cluster costs.

- 1. Randomly select the representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(K)}$
- 2. Iterate:
  - 2.1. Given $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, assign each data point $x^{(i)}$ to the closest representative $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, so that the resulting cost will be $cost(z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}) = \sum_{i=1}^{n} min_{j=1,\ldots,K} ||x^{(i)} - z^{(j)}||^2$
  - 2.2. Given partition $C_1, \ldots, C_j, \ldots, C_K$, find the best representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$ such to minimise the total cluster cost, where now the cost is driven by the clusters: $cost(C_1, .., C_j, \ldots, C_K) = min_{z^{(1)},\ldots,z^{(j)},\ldots,z^{(Z)}} \sum_{j=1}^{K} \sum_{i \in C_J} ||x^{(i)} - z^{(j)}||^2$

Note that in both step 1 and step 2.2 we are not restricted that the initial and final representatives are part of the data sets. The fact that the final representatives will be guarantee to be part of the dataset is instead one of the advantages of the K-medoids algorithm presented in the next Lesson.

13.8. The K-Means Algorithm: The Specifics

Finding the best representatives

While for step (2.1) we can just iterate for each point and each representative to find the representative for each point that minimise the cost, we still need to define how to do exactly the step (2.2). We will see later extensions to the KM algorithm with other distance metrics, but for now let's stuck with the squared geometric distance and note that each cluster select its own representative independently.

Using the squared Euclidean distance, the "new" z_j representative vector for each cluster, must satisfy $j : min_{z_j} cost(z_j; C_j) = min_{z_j} \sum_{i \in C_J} ||x^{(i)} - z_j||^2$.

When we compute the gradient of the cost function with respect to z_j and set it to zero, we retrieve the optimal z_j as $z_j = \frac{\sum_{i \in C_J} x^{(i)}}{|C_J|}$, where $|C_J|$ is the number of elements of the cluster C_J. Intuitively the optimal representative vector is at the center of the cluster, i.e. it is the centroid of the group.

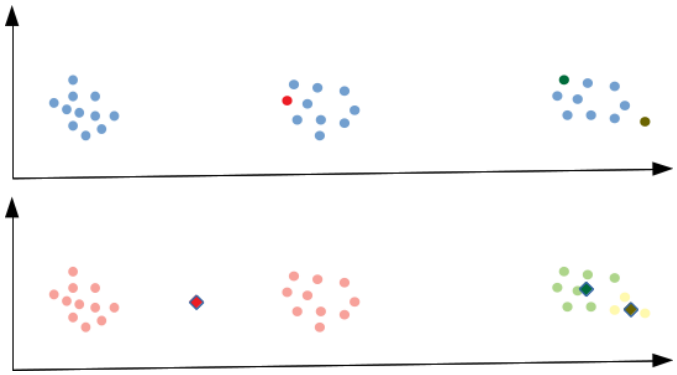
We stress however that this solution is linked to the specific definition of distance used, the squared Euclidean one.

Impact of initialisation

Note that the KM algorithm is guarantee to converge and find a *local* cost minimisation, because at each iteration the cost can only decrease, the cost function is non-negative and the number of possible partitions, however large, is finite.

It *doesn't* however guarantee to find a *global* cost minimisation, and it is indeed very sensitive to the choice of the initial representative vectors. If we start with a different initialization, we may get a very different partitioning.

Take the case of the above picture:



In the upper part we see the initial random assignation of the representative vectors, and in the bottom picture we see the final assignment of the clusters. While the feature vectors moved a bit, the final partition is clearly not the optimal one (where the representative vectors would be at the center of the three groups).

In particular, we may run into troubles when the initial representative vectors are close to each others rather than spread up across the multidimensional space.

While there are improvements to the K_Mean algorithm to perform a better initialisation than a random one, that take this consideration into account, we will use in class a vanilla KM algorithm with simple random initialisation. An example of a complete such KM algorithm in Julia can be found on https://github.com/sylvaticus/lmlj/blob/master/km.jl

Other drawbacks of K-M algorithm

In general we can say there are two classes of drawbacks of the K-M algorithm.

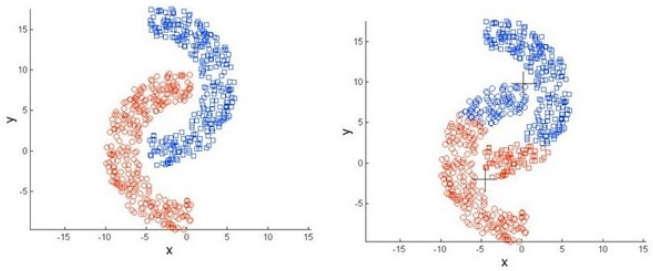
The first class is that the measure doesn't describe what we consider as a natural classification, so the cost function doesn't represent what we would like the cost of the partition to be, it does not

return a useful information concerning the partition ranking.

The second class of problems involves instead the computational aspects to reach the minimum of this cost, like the ability to find a global minimum.

While K-M algorithm scale well to large datasets, it has many other drawbacks.

One is that vanilla K-M algorithm tries to find spherical clusters in the data, even when the groups have other spatial grouping:



To account for this, k-means can be kernelized, where separation of arbitrary shapes can be reached theoretically using higher dimensional spaces. Or there could be used a regularized gaussian mixture model, like in this paper or in these slides.

Other drawbacks, includes the so called “curse of dimensionality”, where k-means algorithm becomes less effective at distinguishing between examples, the manual choice of the number of clusters (that need to be solved using cross-validation), the fact of not being robust to outliers.

These further drawbacks are discussed, for example, here, here or here.

An implementation of K-Means algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L65

Lecture 14. Clustering 2

14.1. Clustering Lecture 2

Objectives:

- Understand the limitations of the K-Means algorithm
- Understand how K-Medoids algorithm is different from the K-Means algorithm
- Understand the computational complexity of the K-Means and the K-Medoids algorithms
- Understand the importance of choosing the right number of clusters
- Understand elements that can be supervised in unsupervised learning

14.2. Limitations of the K Means Algorithm

On top of the limitations already described in segment 13.8 (computational problem to reach the minimum and the cost function not really measuring what we want) there is further significant limitation of the K-Mean algorithm: the fact that the z’s are actually not guaranteed to be the members of the original set of points x.

In some applications this is not a concern, but it is for others. For example, looking at Google News, if we create a representative of the cluster of the story which doesn’t correspond to any story, we actually have nothing to show.

We will now introduce the K-medoids algorithm that modifies the k-means one to consider any kind of distance (not only the squared Euclidean one) and return a set of representative vectors that are always part of the original data set.

We will finish the lesson discussing how to choose K, the number of K.

14.3. Introduction to the K-Medoids Algorithm

In K-means, whenever we randomly selected the initial representative points, we were not constrained to select within the data set points, we could select any point on the plane.

In K-Medoids algorithm instead we start by selecting the representatives only from within the data points.

The step 2.1 (determining the constituencies of the representatives) remains the same, while in step 2.2, instead of choosing the new representatives as centroids, we constrain again that these have to be one of the point of the cluster. This allow to just loop over all the points of the cluster to see which minimise the cluster cost. And as in both step 2.1 and 2.2 we explicitly use a “distance” function we can employ any kind of distance definition we want, i.e. we are no longer restricted to use the squared Euclidean one.

The algorithm becomes then:

- 1. Randomly select the representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(K)}$ from within $X^{(1)}, \ldots, X^{(j)}, \ldots, X^{(n)}$
- 2. Iterate:
  - 2.1. Given $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, assign each data point $x^{(i)}$ to the closest representative $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$, so that the resulting cost will be $cost(z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}) = \sum_{i=1}^{n} min_{j=1,\ldots,K} ||x^{(i)} - z^{(j)}||^2$
  - 2.2. Given partition $C_1, \ldots, C_j, \ldots, C_K$, find the best representatives $z^{(1)}, \ldots, z^{(j)}, \ldots, z^{(Z)}$ from within $X^{(1)}, \ldots, X^{(j)}, \ldots, X^{(n)}$ such to minimise the total cluster cost, where now the cost is driven by the clusters: $cost(C_1, .., C_j, \ldots, C_K) = \sum_{j=1}^{K} min_{z^{(j)}} \sum_{i \in C_J} dist(x^{(i)} - z^{(j)})$

14.4. Computational Complexity of K-Means and K-Medoids

The Big-O notation

Let’t now compare the computational complexity of the two algorithms, using the capital O (or “Big-O”) notation, which talks to us about the order of growth, which means that we are going to look at the asymptotic growth and eliminate all the constants.

We often describe computational complexity using the “Big-O" notation. For example, if the number of steps involved is $5n^2 + n + 1$, then we say it is “of order n^2" and denote this by $O(n^2)$. When n is large, the highest order term $5n^2$ dominates and we drop the scaling constant 5.

More formally, a function $f(n)$ is of order $g(n)$, and we write $f(n) \sim O(g(n))$, if there exists a constant C such that $f(n) < Cg(n)$ as n grows large.

In other words, the function f does not grow faster than the function g as n grows large.

The big-O notation can be used also when there are more input variables. For example, in this problem, the number of steps necessary to complete one iteration depends on the number of data points n, the number of clusters K, the dimension d of each vector x_i. Hence, the number of steps required are of $O(g(n, K, d))$ for some function $g(n, K, d)$.

The computational complexity of the two algorithms

We don’t know how many iterations the algorithms take, so let’s compare only the complexity of a single iteration.

The order of computation for the step 2.1 of the K-Mean algorithm is $O(n * k * d)$ as we need to go trough each point (n), compute the distance with each representative (k) and account that we deal with multidimensional vectors (d). The step 2.2 is the same, but because we’re talking about the asymptotic growth, whenever we sum them up, we’re still staying within the same order of complexity.

For the K-Medoids algorithm instead we can see that is much more complex, as in the step 2.2 we need to go trough all the clusters, and then all the point. Depending how the data is distributed across the cluster, we can go from the best case of all points in one cluster, and then we would have a computational complexity of $O(n^2 * d)$ to a situation where the points are homogeneously distributed across the clusters, and in such case we would have n/k points in each cluster and a total complexity $O(\frac{n}{k} * \frac{n}{k} * k * d) = O(\frac{n^2}{k} * d)$.

Given that normally $n \gg k$, what makes the difference is the n^2 term, and even in the best scenario the computational complexity of the K-Medoids algorithm is much higher than those of the K-Mean one, so for some applications with big datasets the K-Mean algorithm would be preferable.

At this point we already have seen two clustering algorithms, but there are hundreds of them available for our use. Each one has different strengths and weaknesses, and whenever we are selecting a clustering algorithm which fits for our application, we should account for all of them in order to find the best clustering algorithm for our needs.

14.5. Determining the Number of Clusters

First, let's recognise that more K we add to a model, more the cost function decreases, as the distances from each point and the closer representative will decrease, until the degenerated case where the number of clusters equals the number of data and the cost is zero.

When is it time then to stop the number of clusters ? To decide which is the "optimal" number K ? There are three general settings. In the first one the number of clusters is fixed, is really given to us by the application. For example we need to cluster our course content in 5 recitations, this is the space we have. Problem "solved" 😃

In the second setting, the number of clusters is driven by the specific application, in the sense that the optimal level of the trade-off between cluster cost and K is determined by the application, for example how many colours to include in a palette vs the compression rate of the image depends from the context of the application and our needs.

Finally, the "optimal" number of clusters could be determined in a cross-validation step when clustering is used as a pre-processing tool for a supervised learning tool (for example to add a dimension "distance from the cluster centroid" when we have few labelled data and many unlabelled ones). The supervised task algorithm would then have more information to perform its separation task. Here it is the result of the supervised learning task during validation that would determine the "optimal" number of clusters.

Finally let's have a thought on the word "unsupervised". One common misunderstaning is that in "usupervised" tasks, as there is no labels, we don't provide our system with any knowledge, we just "let the data speak", decide the algorithm and let it provide with a solution. Indeed, the people who develop those unsupervised algorithms actually provide quite a bit of indirect supervision, of expert knowledge.

In clustering for example we decide about which similarity measure to use and we decide about how many clusters to give and, as we saw, ow many clusters provide. And if you're thinking about bag of words approach of representing text, these algorithms cannot figure out which words are more semantically important than other words. So it would be up to use to do some weighting, so that the clustering results is actually acceptable.

Therefore, we should think very carefully how to do these decision choices so that our clustering is consistent with the expectation.

An implementation of K-Medoids algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L133

Lecture 15. Generative Models

15.1. Objectives

- Understand what Generative Models are and how they work
- Understand estimation and prediction phases of generative models
- Derive a relation connecting generative and discriminative models
- Derive Maximum Likelihood Estimates (MLE) for multinomial and Gaussian generative models

15.2. Generative vs Discriminative models

The model we studies up to now were **discriminative models** that learn explicit decision boundary between classes. For instance, SVM classifier, which is a discriminative model, learns its decision boundary by maximising the distance between training data points and a learned decision boundary.

At the contrary, **generative models** work by explicitly modelling the probability distribution of each of the individual classes in the training data. For instance, Gaussian generative models fit a Gaussian probability distribution to the training data in order to estimate the probability of a new data point belonging to different classes during prediction.

We will study two types of generative models, **Multinomial generative models** and **Gaussian generative models**, where for both we will ask two type of questions: (1) how do estimate the model ? How do we fit our data (the “estimation” question) and (2) how do we actually do prediction with a (fitted) model ? (the “prediction” question)

What are the advantages of generative models when compared to the discriminative ones?

For example, if your task is not just to do classification but to generate new samples of such (feature, label) pair based on the information provided in training data.

15.3. Simple Multinomial Generative model

We now think to a data-generator probabilistic model where we have different categories and hence a discrete probability distribution (a PMF, Probability Mass Function).

We name θ_w the probability for a given class w, so that $\theta_w \geq 0 \forall w$ and $\sum_w \theta_w = 1$.

Given such probabilistic model, the *generative model* $p(w|\theta)$ is nothing else than the *statistical model* to estimate the parameters θ_w given the observed data w, or more formally the statistical model described by the pair $(E, \{P_\theta\}_{\theta \in \Theta})$, where E is the sample space of the data and $\{P_\theta\}_{\theta \in \Theta})$ is the family of distributions parametrized by θ.

For example the probabilistic model may be a words generating model, given fixed vocabulary of W words and where each word would have its own probability. Then we could see a document as a serie of random (independent) extraction of these words so that a document with common words have more probabilities to be generated compared to a document made of uncommon words. As the words are independent in this model, the likelihood of the whole document to be generated is just the product of the likelihood of each world being generated.

If we are interested in the document as a specific sequence of words, for example “IT WILL RAIN IT WILL” then it’s probability is P(“IT”) * P(“WILL”) * P(“RAIN”) * P(“IT”) * P(“WILL”) = P(“IT”)² * P(“WILL”)² * P(“RAIN”). More in general it is $P(D|\theta) = \prod_{w \in W} \theta_w^{count(w)}$. The underlying probabilistic model is then the categorical distribution, where the sample space E is then the set $1, \ldots, K$ mapped to all the possible words, and $\{P_\theta\}$ is the categorical distribution parametrized by the probabilities θ. Θ, the space of the possible parameters of the probability distribution, is the set of K positive floats that sum to one.

If we are interested instead in all the possible combination of documents that use that specific number of words (in whatever order), {“IT”:2,“WILL”:2,“RAIN”:1}, we have to consider and count all the possible combinations, like “IT WILL IT WILL RAIN”, and there are $\frac{n!}{w_1!,\ldots,w_i!,\ldots w_k!}$ of them. The sample space E is then the set of all the possible documents, encoded as count of the different words, and $\{P_\theta\}$ is the multinomial distribution parametrized by the probabilities θ. Θ, the space of the possible parameters of the probability distribution, remains the set of K positive floats that sum to one.

Note that the categorical and multinomial distributions are nothing else than an extension of respectively the Bernoulli and binomial distributions to multiple classes (rather than just a binary 0/1 ones). They model the probability of respectively one or multiple, independent, categorical trials, i.e. the outcome for the categorical distribution is the single word, while those for the multinomial distribution is the whole document (encoded as world count).

For completion, there are $\frac{n!}{w_1!,\ldots,w_i!,\ldots w_k!}$ possible combinations of sequences of a document with $w_1, \ldots, w_i, \ldots w_k$ worlds count (with $\sum_{i=1}^k w_i = n$), so the PMF of the multinomial is $p(w_1, .., w_i, \ldots, w_k; \theta_1, \ldots, \theta_i, \ldots, \theta_k) = \frac{n!}{w_1!,\ldots,w_i!,\ldots w_k!} \prod_{i=1}^k \theta_i^{w_i}$.

15.4. Likelihood Function

Even if we will use the term “multinomial generative model”, in this lecture we will consider the first approach of encoding a document and its relative probabilistic model (the categorical distribution)

Note indeed that in some fields, such as machine learning and natural language processing, the categorical and multinomial distributions are conflated, and it is common to speak of a “multinomial distribution” when a “categorical distribution” would be more precise.

So, for a particular document D it’s probability to be generated by a sequence of samples from a categorical distribution with parameter θ is $P(D|\theta) = \prod_{w \in W} \theta_w^{count(w)}$.

Given an observed document and compelling probabilistic models (with different thetas) we can then determine wich is the one with the highest probability of having generated the observed document.

Let's consider for example a vocabulary of just two words, "cat" and "dog" and an observed Document "cat cat dog". Let's also assume that we have just two compelling models to choose from. The first (probabilistic) model has $\theta_{\text{cat}} = O.3$ and $\theta_{\text{dog}} = O.7$. The second model has $\theta_{\text{cat}} = O.9$ and $\theta_{\text{dog}} = O.1$ It is intuitive that it is more probable that is the second model that generated the document, but let's compute it. The probability of the document being generated by the first model is $0.3^2 * 0.7 = 0.0189$. The probability that D has been generated instead by the second model is $0.9^2 * 0.1 = 0.081$, that is higher than those for the first model.

15.5. Maximum Likelihood Estimate

The joint probability of a set of given outcomes when we want to stress it as a function of the parameters of the probabilistic model (thus treating the random variables as fixed at the observed values) is known as the likelihood.

While often used synonymously in common speech, the terms "likelihood" and "probability" have distinct meanings in statistics. *Probability* is a property of the sample, specifically how probable it is to obtain a particular sample for a given value of the parameters of the distribution; *likelihood* is a property of the parameter values.

We want now to find which are the parameters that maximise the likelihood.

For the multinomial model it is $\text{argmax}_\theta \{L(D|\theta_0) = \prod_{w \in W} \theta_w^{\text{count}(w)}\}$.

It turns out that maximising its log (known as the log-likelihood) is computationally simpler while equivalent (in terms of the argmax, not in terms of its value) because the log function is a monotonically increasing function: wherever the likelihood is on its maximum, its log it is on its maximum as well.

We hence compute the first order conditions in terms of theta for the log-likelihood to find the theta that maximise it:

$\text{argmax}_\theta \{lL(D|\theta_0) = \sum_w \text{count}(w) log(\theta_w)\}$

Max likelihood estimate for the cat/dog example

Let's first consider the specific case of just two categories to later generalise (and let's calling the outcomes just 0/1 instead of cat/dog).

In such setting we have really just one parameter, θ_0, (the probability of a 0) as we can write θ_1 as $1 - \theta_0$ The log_Likelihood of a given document is then:

$$lL(D|\theta_0) = \text{count}(0) log(\theta_0) + \text{count}(1) log(1 - \theta_0)$$

Taking the derivative with respect to θ_0 and setting it equal to zero we obtain:

$$\frac{\partial lL}{\partial \theta_0} = \frac{\text{count}(0)}{\theta_0} - \frac{\text{count}(1)}{1 - \theta_0}$$

$$\frac{\partial lL}{\partial \theta_0} = 0 \rightarrow \tilde{\theta}_0 = \frac{\text{count}(0)}{\text{count}(0) + \text{count}(1)}$$

Note the symbol of the tilde to indicate an estimated parameter. This is our "best guess" parameter given the observed data.

15.6. MLE for Multinomial Distribution

Let's now consider the full multinomial case. So given a document (or, as the words are assumed to be independent, a whole set of documents… just concatenated one to the other) and a vocabulary W we want to find $\theta_w \forall w \in W$ as to maximise $\${L(D|\theta_0) = \prod_{w \in W} \theta_w^{\text{count}(w)}\}$.

The problem is however that the thetas are not actually free, but we have the constrain that the thetas have to sum to one (we have actually also those that the thetas need to be positive, but for the nature of this maximisation problem we can ignore this constraint).

The method of Lagangian multipliers for constrained optimisation

We have hence a constrained optimisation problem that we can solve with the method of the **Lagrange multipliers**, a method to find the stationary values (min or max) of a function subject to *equality* constraints.

Let's consider a case of an objective function of two variables $z = f(x, y)$ subject to a single constraint $g(x, y) = c$, where c is a constant (both the objective function and the constraint doesn't need to be necessarily linear).

We can then write the so-called *Lagrangian function* as $\mathscr{L} = f(x, y) + \lambda[c - g(x, y)]$, i.e. we "augment" the original function we want to find the extremes with a term made by a new variable (the Lagrangian multiplier) that multiplies the relative constraint (if we have multiple contraints, we would have multiple additional terms and Lagrangian multipliers).

All we need to do now is to find the stationary values of $\mathscr{L}$, regarded as a function of the original variables x and y but also of the new variable(s) λ we introduced.

The First Order necessary Condition (FOC) are:

$$\begin{aligned}\mathscr{L}_x &= f_x - \lambda g_x \quad &= 0\\ \mathscr{L}_y &= f_y - \lambda g_y \quad &= 0\\ \mathscr{L}_\lambda &= c - g(x, y) &= 0\end{aligned}$$

Solving this system of 3 equations in 3 unknown will equivalently find the stationary point of the unconstrained function $\mathscr{L}$ and original function f, as the third equation in the FOC guarantee that indeed the constrain is respected and, at the stationary point, the two functions have the same value.

For a numerical example, let's the function to optimize be $z = xy$ and be it subjected tothe constraint $x + y = 6$. We can write the Lagrangian as $\mathscr{L} = xy + \lambda[6 - x - y]$ whose FOC are:

$$\begin{aligned}\mathscr{L}_x &= y - \lambda \quad &= 0\\ \mathscr{L}_y &= x - \lambda \quad &= 0\\ \mathscr{L}_\lambda &= 6 - x - y &= 0\end{aligned}$$

Solving for (x, y, λ) we find the optimal values $x^* = 3$, $y^* = 3$, $\lambda^* = 3$ and $z^* = 9$.

The constrained multinomial log-likelihood

As the sum of thetas must be one, the Lagrangian of the log-likelihood is:

$$\mathscr{L} = \log P(D|\theta) + \lambda\left(\sum_{w \in W} \theta_w - 1\right) = \sum_{w \in W} n_w \log \theta_w + \lambda\left(\sum_{w \in W} \theta_w - 1\right)$$

where n_w is the count of word w in the document.

The FOC of the lagrangian are then:

$$\begin{aligned}\mathscr{L}_w &= \frac{n_w}{\theta_w} - \lambda \quad &= 0\\ \mathscr{L}_\lambda &= \sum_{w \in W} \theta_w - 1 &= 0\end{aligned}$$

Where the first equation is actually a gradient with respect to all the w.

Solving the above system of equation we obtain $\hat{\theta}_w = \frac{n_w}{\sum_{w \in W} n_w} \forall w \in W$.

These set of θ_w parameters are the **maximum likelihood estimates**, the values of θ that maximize the likelihood function for the observed data and this multinomial generative model.

15.7. Prediction

Let's now see how can we actually use generative Multinomial Models to make predictions, for example deciding if a document is in Spanish of Portuguese language.

For that we have first given a large set of documents in both languages, together with their label (the language).

From there we can estimate the PMF of the words using the MLE estimator (i.e., counting the relative proportions) for both Spanish and Italian (so our vocabulary would have both Spanish and Italian terms).

We are now given a new document without the language label. Calling the two languages as "+" and "-", we compute the likelihood of the document under the first hypothesis ($P(D|\theta^+)$) as the joint probability using the first PMF , and the likelihood under the second hypothesis ($P(D|\theta^-)$) using the second PMF, and we decide how to classify the document according to which one is bigger.

The probability of a data item to belong to a given class, conditional to the probabilities of its feature vector is known also as class-conditional probability.

We can equivalently say that we category the document as "+" if $log(\frac{P(D|\theta^+)}{P(D|\theta^-)}) > 0$.

But $\log \frac{P(D|\theta^+)}{P(D|\theta^-)} = \log P(D|\theta^+) - \log P(D|\theta^-)$

$= \log \prod_{w \in W} (\theta_w^+)^{\text{count}(w)} - \log \prod_{w \in W} (\theta_w^-)^{\text{count}(w)}$

$= \sum_{w \in W} \text{count}(w) \log \theta_w^+ - \sum_{w \in W} \text{count}(w) \log \theta_w^-$

$= \sum_{w \in W} \text{count}(w) \log \frac{\theta_w^+}{\theta_w^-}$

$= \sum_{w \in W} \text{count}(w) \hat{\theta}_w$

where $\hat{\theta}_w$ is just a symbol for $\log \frac{\theta_w^+}{\theta_w^-}$.

The last expression shows that this classifier, derived looking through a generative view on classification, should actually remind us a linear classifier that goes through origin with respect to this parameter $\hat{\theta}_w$.

15.8. Prior, Posterior and Likelihood

Let's now try to compute $P(\theta|D)$, i.e. the probability of the language of the document (that is the probability of the generating PMF being those of the Spanish or Portuguese language) given the observed document.

To compute it, we will apply the **Bayesian rule** that states:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

and the **total probability law** that states:

$$P(X = x) = \sum_y P(Y = y) * P(X = x|Y = y)$$

$$P(lang = spanish|D) = \frac{P(D|lang=Spanish) * P(lang=Spanish)}{P(D)}$$

Where
$$P(D) = P(D|lang = Spanish) * P(lang = Spanish) + P(D|lang = Portuguese) * P(lang = Portuguese)$$

We can think for example to a bilingual colleague giving us a document in either Spanish or Portuguese. $P(lang = Spanish)$ is the **prior** probability that the document is Spanish (because maybe we know that this guy works more with Spanish documents than Portuguese ones). Then $P(D|lang = Spanish)$ and $P(D|lang = Portuguese)$ are the two conditional probabilities that depends on the world frequencies of the two languages respectively and $P(D)$ is the probability that our colleague gave us exactly that document. $P((lang = spanish|D)$ is known as the **posterior** probability, as it is the one we have once we observe document D.

Going back to our plus/minus class notation instead of the languages, we can compute the log of the ratio between the posteriors for the two classes as:

$$log(\frac{P(y=+|D)}{P(y=-|D)}) = log(\frac{P(D|y=+) * P(y=+)}{P(D|y=-) * P(y=-)}) = log(\frac{P(D|y=+)}{P(D|y=-)}) + log(\frac{P(y=+)}{P(y=-)})$$

Using the expression from the previous segment for the first term and $\hat{\theta}_0$ as a symbol representing the second term, we can rewrite the equation as:

$$log(\frac{P(y=+|D)}{P(y=-|D)}) = \sum_{w \in W} \text{count}(w) \hat{\theta}_w + \hat{\theta}_0.$$

So now what we actually see here that we translated it again to a linear classifier. But in contrast to the previous case, when we had a linear classifier that went through origin, now we have a linear

classifier with an offset, and the offset itself would be actually guided by our prior, which will drive the location of the separator.

So what we've seen here that in this model we can very easily incorporate our prior knowledge about the likelihood of certain classes. And at the end, what we got, that even though we're talking about generative models and we're using a different mechanism on some ways of estimating the parameters of this multinomial, at the end, we actually are getting the same linear separators that we see in our discriminative modelling.

15.9. Gaussian Generative models

We will now switch, in place to a categorical distribution, to a generative (probabilistic) model based on the multidimensional Normal (or "Gaussian") distribution.

While the categorical distribution was opportune for modelling discrete random variables, like classes, the Normal is the analogous counterpart for continuous random variables.

The multidimensional Normal has Probability Density Function (PDF):

$$p_{\mathbf{X}}(\mathbf{x};\mu,\mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)}, \quad \mathbf{x} \in \mathbb{R}^d$$

where $\mathbf{x} \in \mathbb{R}^d$, $\mu \in \mathbb{R}^d$ is the vector of the means in the d dimensions and $\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$ is the covariance matrix (with $\det(\Sigma)$ being its determinant).

So the PDF has a single spike in correpondance of $x = \mu$ and then gradually declines in a typical "bell shape". The "speed" of the decline is regulated by the sigma parameter: more sigma is high, more the spike is low and the tails are "fat"; making the whole curve "flatted".

When all the components (dimensions) are uncorrelated (i.e. the covariant matrix Σ is diagonal) and have the same standard deviation σ, the Normal PDF reduces to:

$$p_{\mathbf{X}}(\mathbf{x};\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2}*||x-\mu||^2}$$

We should also specify that a random vector $\mathbf{X} = (X^{(1)},\ldots,X^{(d)})^T$ is defined as a **Gaussian vector**, or "multivariate Gaussian" or "normal variable", if any linear combination of its components is a (univariate) Gaussian variable or a constant (a "Gaussian" variable with zero variance), i.e., if $\alpha^T\mathbf{X}$ is (univariate) Gaussian or constant for any constant non-zero vector $\alpha \in \mathbb{R}^d$.

It is important to note that in generative models we are introducing a specific functional form for the probabilistic model (here the gaussian), and our freedom relates only on the parameters of the model, not on the form itself. Hence we first need to consider if our data, our specific case, is appropriate to be modelled by such functional form or not.

15.10. MLE for Gaussian Distribution

As for the multinomial case, if we have a training set $S_n = \{x^{(t)}|t = 1...n\}$ we can compute the -log-likelihood of this set and find the (μ,σ^2) that maximise it:

The likelihood is $p(\mu,\sigma^2;S_n) = \prod_{t=1}^n p(\mu,\sigma^2;x^{(t)}) = \prod_{t=1}^n \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2}*||x-\mu||^2}$

The log-likelihood is then:

$$logp(\mu,\sigma^2;S_n) = log(\prod_{t=1}^n \frac{1}{(2\pi\sigma^2)^{d/2}} * e^{-\frac{1}{2\sigma^2}*||x^{(t)}-\mu||^2})$$

$$= -\frac{nd}{2}log(2\pi\sigma^2) + \frac{1}{2\sigma^2}||x^{(t)}-\mu||^2$$

From this expression we can take the derivatives of μ and σ and equal them to zero to find the MLE estimators $\hat{\mu} = \frac{1}{n}\sum_{t=1}^n x^{(t)}$ and $\hat{\sigma}^2 = \frac{\sum_{t=1}^n ||x^{(t)}-\mu||^2}{nd}$.

Lecture 16. Mixture Models; EM algorithm

16.1. Mixture Models and the Expectation Maximization (EM) Algorithm

Objectives:

- Review Maximum Likelihood Estimation (MLE) of mean and variance in Gaussian statistical model
- Define Mixture Models
- Understand and derive ML estimates of mean and variance of Gaussians in an Observed Gaussian Mixture Model
- Understand Expectation Maximization (EM) algorithm to estimate mean and variance of Gaussians in an Unobserved Gaussian Mixture Model

16.2. Recap of Maximum Likelihood Estimation for Multinomial and Gaussian Models

So far, in clustering we have assumed that the data has no probabilistic generative model attached to it, and we have used various iterative algorithms based on similarity measures to come up with a way to group similar data points into clusters. In this lecture, we will assume an underlying probabilistic generative model that will lead us to a “natural” clustering algorithm called the **EM algorithm** .

While a “hard" clustering algorithm like k-means or k-medoids can only provide a cluster ID for each data point, the EM algorithm, along with the generative model driving its equations, can provide the posterior probability (“soft" assignments) that every data point belongs to any cluster.

Today, we will talk about mixture model and the EM algorithm. The rest of the segment is just a recap of the two probabilistic models we saw, the multinomial (actually the categorical) and the multidimansional Gaussian.

The corresponding statistical model is, given observed data and the assumption that the data has been generated by the given probabilistic model, which is the most likely parameter of the distribution, like the individual category probabilities θ_w for the categorical or (μ, σ^2) for the Gaussian ?

This is known in statistics as parametric estimation, and can be solved by interpreting the joint probability function of a set of observed data in terms of a function of the parameter of the model given the observed data (and in doing this we name the joint probability function “likelihood”) and then find the parameter(s) of the model that maximise it, using first order conditions, eventually employing Lagrange Multipliers if the maximisation involves respecting some contraints, like setting the sum of all the categorical probabilities equal to one.

Please also consider that the sum of independent normal random variables remains normal:

$$X \sim N(\mu, \sigma^2) \rightarrow aX + b \sim N(a\mu + b, a^2\sigma^2)$$

16.3. Introduction to Mixture Models

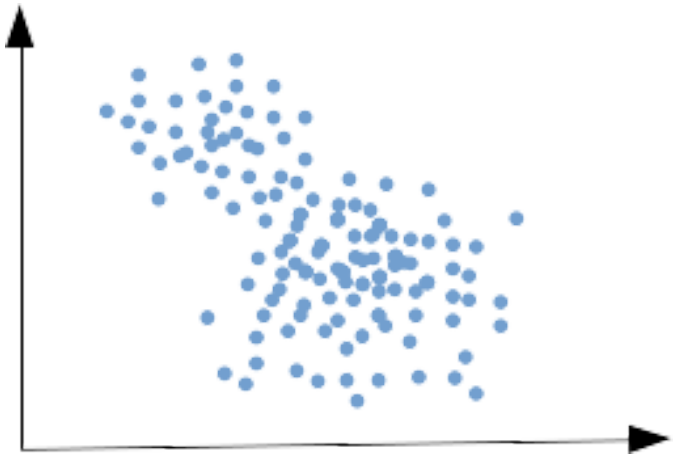
Let’s know consider a generative (probabilistic) model consisting of two subsequent steps. We first draw a class using a K-categorical distribution with probabilities $p_1, \ldots, p_j, \ldots p_K$.

We then draw the final data from a Normal distribution with parameter $(x^{(j)}$ and $\sigma^{2(j)})$, i.e. specific of the class sampled in the first step.

This generative model goes under the name of **mixture model** and mixture models are actually very useful for describing many real-life scenarios.

The full set of parameters of this mixture model is then collectively represented by $\theta = \{p_1, \ldots, p_j, \ldots p_K; \mu^{(1)}, \ldots, \mu^{(j)}, \ldots, \mu^{(K)}, \sigma^{2(1)}, \ldots, \sigma^{2(j)}, \ldots, \sigma^{2(K)}\}$ where the non-negative p_j are called **mixture weights** (and sum to 1) and the individual Gaussians with parameters $\mu^{(j)}$ and $\sigma^{2(j)}$ are the **mixture components** (so this is more specifically a **Gaussian mixture model (GMM)**, not to be confonded with the “Generalized Method of Moments”). K is the size of the mixture model.

The output of sampling a set of points from such model, using 2 components (two classes of the categorical distribution) and a two-dimensional Gaussian, would look something like this:



We can intuitively distinguish the two clusters and see that one is bigger than the other, both in terms of points (i.e. its p_j is larger) and in term of spread (its $\sigma^{2^{(j)}}$ is bigger).

But where is the border between the two points? While the K-Mean or K-Medoids algorithm would have provided a classifier, returning one cluster id for each point (**hard clustering model**), a clustering based on a mixture generative model (to be implemented using the EM algorithm we will discuss in the next segment) would instead return for each point the *probability* to belong to each cluster (**soft clustering model**). For example, the points in the top-left corner of the picture will have probability to belong to the first cluster almost one and those in the bottom-right corner probability to belong to the second cluster (i.e. being generated by a Normal distribution with parameters (μ_2, σ_2^2)) almost 1, but those in the center will have much more balanced probabilities.

So here, we have a much more refined way to talk about our distribution, because every point kind of belongs to different components, just with different likelihoods.

If we know all of the parameters of the K Gaussians and the mixture weights, the probability density function $p(x|\theta)$ can be computed using the law of total probability as $p(\mathbf{x}|\theta) = \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}; \mu^{(j)}, \sigma_j^2)$.

Using the Bayes rule we can also find the posterior probability that $\mathbf{x}$ belongs to a Gaussian component j.

Likelihood of Gaussian Mixture Model

Given the GMM we can find the PDF associated with each point x using the total probability theorem: $p(\mathbf{x}; \theta) = \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}, \mu^{(j)}, \sigma_j^2)$

The likelihood associated to a set S_n of observed, independent, data is then $p(\theta, S_n) = \prod_{i=1}^{n} \sum_{j=1}^{K} p_j \mathcal{N}(\mathbf{x}^{(i)}, \mu^{(j)}, \sigma_j^2)$

Now, given this expression, we want to take the derivatives of it with respect to my parameters, make them equal to 0, and find the parameters. It turns out however that's actually a pretty complex task.

We will first start to reason in a setting where we know the category associated to each point, to further generalise in a context where we don't know the class from which it belongs, with the EM algorithm.

Note also that while we will consider different means of the Gaussian across its dimensions (i.e. $\mu^{(j)}$ is a vector) we will assume the same variance across them (i.e. σ_j^2 is a scalar).

16.4. Mixture Model - Observed Case

Let's hence first assume that we know the cluster j to which each point i belong to and we need to estimate the parameters of the likelihood using MLE.

For that let's define as **indicator function** a function that, for a given pair (i, j), returns 1 if the j cluster is the one to with point i belongs to, 0 otherwise.

$$\mathbf{1}(j; i) := \begin{cases} 1 & \text{if } x^{(i)} \in \text{cluster}_j, \\ 0 & \text{if otherwise.} \end{cases}$$

The log-likelihood can then be rewritten using such indicator function and inverting the summation as:

$logp(\theta, S_n) = \sum_{j=1}^{K} \sum_{i=1}^{n} \mathbf{1}(j;i) log[p_j \mathcal{N}(\mathbf{x}^{(i)}, \mu^{(j)}, \sigma_j^2)]$

Note that in general $log(sum(x))$ is not the same as $sum(log(x))$, but the presence of the indication function guarantees that in reality there is no summation over the cluster when we have a specific data example.

And what this expression actually demonstrates is that, whenever we are trying to find all the parameters that optimize for each mixture component, we can actually do this computation for each cluster independently. We can independently find the best parameters that fit the cluster, because the fact that these points are observed, we actually know where the point is belonging to. So we can separately find, here the mu, sigma and the p.

By maximising the likelihood, we would find the following estimators:

- $\hat{n}_j$, the number of points of each cluster: $\hat{n}_j = sum_{i=1}^{n} \mathbf{1}(j;i)$ (this actually just derives from our assumption to know the indicator function output)
- $\hat{p}_j = \frac{\hat{n}_j}{n}$
- $\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{i=1}^{n} \mathbf{1}(j;i) * x^{(i)}$
- $\hat{\sigma}_j^2 = \frac{1}{\hat{n}_j d} \sum_{i=1}^{n} \mathbf{1}(j;i) * ||x^{(t)} - \mu^{(j)}||^2$

These equations show how, given the observed case, when we know to which component each point belong, we can estimate all the parameters that we need to define our mixture of Gaussian.

16.5. Mixture Model - Unobserved Case: EM Algorithm

We now consider the unobservated case, where we don't know the mapping between data x_i and cluster j.

The notation of the indicator function implies a hard counting: the point belongs to one cluster with certainty or it doesn't belong. But one of the powers of mixture model is that we can see each point to doesn't solely belong to one single cluster, we know that it can be generated from different ones with different probabilities. So now, instead of talking about hard counts - belong or doesn't - we'll actually talk about **soft counts**: how much does this point belongs to this particular cluster?

What we want is $p(j|x^{(i)})$: the probability that, having observed point i, this would have been generated by cluster j.

If we would know all the parameters we could compute it using the Bayes rule:

$p(j|x^{(i)}) = \frac{p(j,x^{(i)})}{p(x^{(i)})} = \frac{p(j) * p(x^{(i)}|j)}{\sum_{k=1}^{K} p(k) * p(x^{(i)}|k)}$

Indeed $p(j)$ would be just our p_j and $p(x^{(i)}|j)$ would be $N(\mu_j, \sigma_j^2)$.

But unfortunately there is no closed solution for maximising the likelihood $p(\theta, S_n)$ involving the full set of parameters, even with a very large S_n observed, as the parameters are very interconnected.

We need hence to break these dependence with a solving procedure in two parts: given my best estimated parameters, I first compute $p(j|x^{(i)})$. Then, treating the estimated $p(j|x^{(i)})$ as given, I maximise the likelihood $p(\theta, S_n)$ to retrieve the parameters. Solving for the parameters I would then find something very similar to the results in the previous segment:

- $\hat{n}_j$, the number of points of each cluster: $\hat{n}_j = sum_{i=1}^{n} p(j|i)$ (this is actually a weighted sum of the points, where the weights are the relative probability of the points to belong to cluster j)
- $\hat{p}_j = \frac{\hat{n}_j}{n}$
- $\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{i=1}^{n} p(j|i) * x^{(i)}$
- $\hat{\sigma}_j^2 = \frac{1}{\hat{n}_j d} \sum_{i=1}^{n} p(j|i) * ||x^{(t)} - \mu^{(j)}||^2$

At this point I can re-compute the $p(j|x^{(i)})$ and so on until my results don't change any more or change very little.

Finding the posterior $p(j|x^{(i)})$ given the parameters and deriving the expected likelihood under this $p(j|x^{(i)})$ is called the **E step** (for "expectations"), while then finding the parameters by maximising the likelihood given the estimated $p(j|x)$ is called the **M step** (for "maximisation"). Note that the EM

algorithm is not specific to the GMM, but it can be used any time we need to estimate the parameters in a statistical models where the model depends on unobserved latent variables.

Of course we are still left with the problem of how to initiate all the parameters for the first E step. The EM algorithm indeed is very sensitive to initial conditions as it guarantee to find only a *local* maximisation of the $p(\theta, S_n)$, not a global one.

We could either do a random initialization of the parameter set θ or we could employ k-means to find the initial cluster centers of the K clusters and use the global variance of the dataset as the initial variance of all the K clusters. In the latter case, the mixture weights can be initialized to the proportion of data points in the clusters as found by the k-means algorithm.

Summary of Likelihood and EM algorithm

Which is the Likelihood of the observed dataset S_n ? It is the joined PDF of each x_i:

$L(S_n) = \prod_{i=1}^{n} \sum_{j=1}^{K} p_j N(x_i; \mu_j, \sigma_j^2)$

We could try to directly maximize that for all the p_j, μ_j and σ_j^2 but there are too many parameters interconnected, the FOC would not have a closed form.

Rather let's try to implement a Likelihood for each single cluster, and try to maximise them.

The fist step is to find the weights to relate each x_i to a given cluster j, that is $p(J = j|X = x)$, the probability that given a certain x_i, the category for which it belongs is j.

Let's use the Bayes rule for that, noticing that the joined density to have both class j and data x_i is $f(J = j, X = x_i) = p(J = j) * f(X = x_i|J = j)$, that is; $p_j * N(x_i; \mu_j, \sigma_j^2)$. But this is also equal to $f(X = x_i) * p(J = j|X = x_i)$ where the last term is the one we want to find.

Given that $f(X = x_i)$ by the total probability theorem is $f(X = x_i) = \sum_{j=1}^{K} p(J = j) * f(X = x_i|J = j)$ that is $\sum_{j=1}^{K} p_j * N(x_i; \mu_j, \sigma_j^2)$

$p(J = j|X = x_i)$ is then equal to $\frac{p(J=j)*f(X=x_i|J=j)}{f(X=x_i)}$, that is $p(J = j|X = x_i) = \frac{p_j * N(x_i;\mu_j,\sigma_j^2)}{\sum_{j=1}^{K} p_j * N(x_i;\mu_j,\sigma_j^2)}$

We can now rewrite the likelihood using a different probability for each x_i rather than a single p_j:

$L(S_n) = \prod_{i=1}^{n} \sum_{j=1}^{K} p(J = j|X = x_i) N(x_i; \mu_j, \sigma_j^2)$. The key point is that once we replaced $p(J = j|X = x_i)$ with some values find in the previous step (the E step), we can now (in the M step) treat the problem as if it would be separable, i.e. writing the likelihood of the data for each cluster as $L(S_n; j) = \prod_{i=1}^{n} p(J = j|X = x_i) N(x_i; \mu_j, \sigma_j^2)$ where we can easily write the FOC in terms of two parameters μ_j, σ_j^2 that at this time do not depend from the other js.

At this point we go back to the E step until the likelihood converges to a (local) maximum.

An implementation of E-M algorithm in Julia: https://github.com/sylvaticus/lmlj.jl/blob/master/src/clusters.jl#L222

Homework 5

The reason while the EM is guarantee to converge (although on a local extreme) is to be found in the Jensen's inequality (details in tab 3 or this thread).

Project 4: Collaborative Filtering via Gaussian Mixtures

We would like to extend our EM algorithm for the case of the problem of matrix completion, which is a very important problem in machine learning. We've already seen how to do matrix completion using K nearest and Matrix Factorisation approaches (Lecture 7), and this is another approach to do matrix completion, which assumes a statistical model for the underlying data that we observe.

In this model there are N users and D items to rate, and each rating is the outcome of a Gaussian Mixture Model where each user soft-belongs to one of the k-user types (the Gaussian mixtures with mean μ_k), and then a random component determines the exact rating $x_{n,d}$.

Our task is to run the EM algorithm to find the posteriors $p_{(n,k)}$ and the mean of the mixtures μ_k, so that in a second step we can fill the empty rating with their expected values: $E[x_{n,d}] = \sum_{k=1}^{K} p_{(n,k)} * E[\text{mixture}_k] = \sum_{k=1}^{K} p_{(n,k)} * \mu_k$.

The only differences with the EM algorithm we saw earlier is that:

- (1) we need to account that the learning of the parameters have to come only from the available data, i.e. we need to "mask" our x_n and μ_k arrays for accounting only of the dimensions we have. The computation of the posteriors in the M step becomes: $p(k \mid n) = \frac{\pi_k N(x_{C_n}^{(n)}; \mu_{C_n}^{(k)}, \sigma_k^2 I_{C_n \times C_n})}{\sum_{k=1}^{K} \pi_k N(x_{C_n}^{(n)}; \mu_{C_n}^{(k)}, \sigma_k^2 I_{C_n \times C_n})}$.

  The FOC for μ and σ^2 that maximise the likelihood in the m step become:

  - $\widehat{\mu_d^{(k)}} = \frac{\sum_{n=1}^{N} p(k|n) C_{(n,d)} x_d^{(n)}}{\sum_{n=1}^{N} p(k|n) C_{(n,d)}}$
  - $\widehat{\sigma_k^2} = \frac{1}{\sum_{n=1}^{N} |C_n| p(k|n)} \sum_{n=1}^{N} p(k \mid n) \| x_{C_n}^{(n)} - \widehat{\mu_{C_n}^{(k)}} \|^2$

  Where C is the mask matrix.

- (2) When the dimensions are so much, or when there are few (or even none) observations for a user, would lead to numerical instabilities (very low value of the normal PDf, or divisions by zero). Some *computational tricks* are then needed, like working ion the log-domain (including computing the log-Normal instead of the "normal" Normal 😉), updating the μ only when its denominator is higher than zero (to avoid erratic behaviour) or keeping a minimum variance (suggested: `0.25`).

P4.3. Expectation–maximization algorithm

Data Generation Models

[MITx 6.86x Notes Index]

[MITx 6.86x Notes Index]

Unit 05 - Reinforcement Learning

Lecture 17. Reinforcement Learning 1

17.1. Unit 5 Overview

This unit covers reinforcement learning, a situation somehow similar to supervised learning, but where instead of receiving feedback for each action/decision/output, the algorithm get a overall feedback only at the very end. For example, in "playing" a game like Chess or Go, we don't give to the machine a reward for every action, but what counts whether the whole game has been successful or not.

So we will start approaching the problem of reinforcement learning by looking at a more simplified scenario, which is called **Markov Decision Processes (MDPs)**, where the machine needs to identify the best action plan when it knows all possible rewards and transition between states (more on these terms soon). Then, we are going to lift some of the assumptions and look at the case of full reinforcement learning, when we experience different transitions without knowing their implicit reward, and collect them only at the end, which is a typical case in life.

In the project at the end of this unit we will implement an agent that can play text-based games.

Summary:

- Lesson 17: How to provide the optimal policy for a Markov Decision Process (using Bellman equations) when you know all the state probabilities/rewards
- Lesson 18: How to do the same while you don't know them and you need to learn them little by little with experience. Very interesting concept of exploration vs exploitation
- Lesson 19: Qualitative lesson (but with some interesting concepts) on the ML research in Natural Language Processing

17.2. Learning to Control: Introduction to Reinforcement Learning

This lesson introduces reinforcement learning. The kind of tasks we can solve are more extended than supervised learning, as the algorithm doesn't need te be given an info of the payoff at each single step. And in many real-world problems we cannot get supervision to the algorithm at every point of the progression.

In real world, we go, we try different things, and eventually some of them succeed and some of them don't. And as intelligent human being, we actually know to trace back an attribute, what made our whole exploration successful.

For example, a mouse learning to escape a maize and get some food, learns how to exit the maize even if he doesn't get a reward at each crossing, but only at the end of exiting the maize.

In many occasions we are faced with the same scenarios: computer or board games, robots learning to arrive to a certain point (Andrew Ng PhD dissertation was about teaching a robot helicopter to fly), but also the business decisions, e.g. call, email, sending gift to lead a client to sign a contract (note that each of this option may have, taken individually, a negative reward, as they imply costs).

Note that in some cases you may also get some rewards in the middle, but still what must count, is the final situation, if the goal has been reached or not.

So these are the type of problems that are related to reinforcement learning, where our agent can take a lot of different actions, and what counts is the reward this agent gets or doesn't get at the end. And the whole question of reinforcement learning is how can we get this ultimate reward that we will get and propagate it back and learned how to behave.

Defined the objectives and the type of problems that can be solved with reinforcement learning, the next point of this lecture will discuss how to formalize these problems, setting the problem in clear mathematical terms.

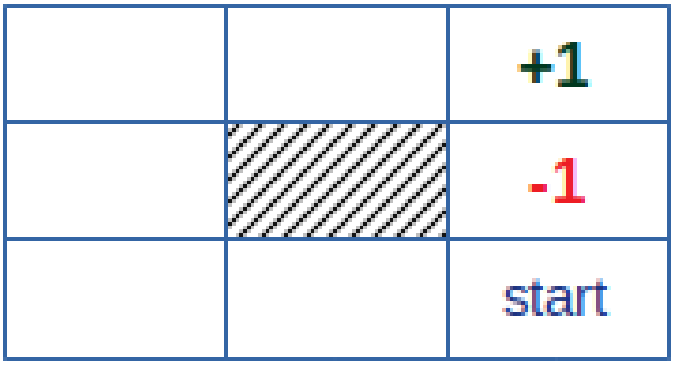
We will first introduce Markov Decision Processes, introducing concepts as state, reward, actions, and so on.

We will then discuss the so called **Bellman equations** that allow us to propagate the goodness of the state from the reward state to all the other state, including the initial one. Based on this discussion we would be able to formulate value iteration algorithms that can solve the Markov Decision Processes.

Across the remaining of the exposition in this lecture we will consider the following, very simple, example.

In the following table of 3 X 3 cells, a "robot" has to learn to reach the top-right corner, getting then a $+1$ reward, while if it ends in the lower cell it gets a negative reward -1. Also there will be some contraints, like being forbidden to go to the central cell.

Our goal is to find a strategy to send the robot on that particular top-right corner



17.3. RL Terminology

The terminology for today:

- **states** $s \in S$: in this lecture they are assumed to be all observed, i.e. the robot know in which state it is (the "observed case")
- **actions** $a \in A$: What we can actually do in this space, in this example going one cell up/down/right or left (and if we hit the border we remain in the same state)
- **transition** $T(s, a, s') = p(s'|s, a)$: The probability to end up to state s' conditional to being in state s and performing action a. For example, starting in the bottom-left cell and performing action "going up" you may have 80% probability of actually going up, 10% of going on the adiacent right cell, and 10% of remaining in the same cell. Note that $\sum_{s' \in S} T(s, a, s') = 1$
- **reward** $R(s, a, s')$: The reward for ending up in state s' conditional to being in state s and performing action a. In the example here the reward can be defined as only a function of s (i.e. the *leaving* cell), but more generically (as in the example of marketing where each action involves a cost) it is a function also of a and s'.

This problem in a deterministic set up would just require planning, nothing particular.

But here we frame the problem in a stochastic way, that is, there is a probability that even if the chosen action was to move to the top cell, the robot ends up instead to an other cell. This is why we introduce transitions. Note: this is very similar to the Markov Chains studied in the Probability course (6.431x Probability–The Science of Uncertainty and Data, Unit 10). There, there is a simple transition matrix. Here the transition matrix depends from the action employed by the agent.

Indeed: *"A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. Markov decision processes are an extension of Markov chains; the difference is the addition of actions (allowing choice) and rewards (giving motivation). Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), a Markov decision process reduces to a Markov chain."*

In MDP we assume that the sets of states, actions, transitions, and rewards are given to us: $MDP = < S, A, T, R >$. MDPs satisfy the Markov property in that the transition probabilities, the rewards and the optimal policies depend only on the current state and action, and remain unchanged regardless of the history (i.e. past states and actions) that leads to the current state. In other words, The state we may end up after taking an action and the reward we are getting only depends on the current state, which encapsulates all the relevant history.

We can consider several variants of this problem, like assuming a particular initial state or a state whose transitions are all zero except those leading to the state itself, e.g. once arrived on the rewarding cell on the top-right corner you can't move any more, you are done.

When we will start thinking about more realistic problem, we will want to have a more complicated definition of transition, function, and the rewards. But for now, for what we're discussing today, we can just imagine them as tables, i.e. constant across the steps.

Reward vs cost difference

What is the difference between *reward* and *cost* ?

The reward generally refers to the value an agent might receive for being in a particular state. This value is used to positively "reinforce" the set of actions taken by the agent to get itself into that state.

The cost generally refers to the value an agent might have to "pay", "expend", or "lose" in order to take an action.

Think about there are things you want to do (e.g. pass this course): doing this would result in some reward (e.g. getting a degree or qualification). But doing this isn't free: you have to spend time studying, etc. which can be seen as "costly". In other words, you're going to have to spend something (in this case, your time) in order to reach that rewarding state (in this case, passing this course).

In the same way, we can model RL settings where an agent is told that a particular state has a reward (completing a maze gives the agent +100) but each action it takes has a cost (walking forward, backward, left, or right gives the agent -1). In this way, the agent will be biased towards finding the most efficient/cheapest way to complete the maze.

17.4. Utility Function

The main problem for MDPs is to optimize the agent's behavior. To do so, we first need to specify the criterion that we are trying to maximize in terms of accumulated rewards.

We will define a utility function and maximize its expectation. Note that this should be a finite number, in order to compare different possible strategies, to find which is the best one.

We consider two different types of "bounded" utility functions:

- **Finite horizon based utility** : The utility function is the sum of rewards after acting for a fixed number n steps. For example, in the case when the rewards depend only on the states, the utility function is $U[s_0, s_1, \ldots, s_n] = \sum_{i=0}^{n} R(s_i)$ for some fixed number of steps n. In particular $U[s_0, s_1, \ldots, s_{n+m}] = U[s_0, s_1, \ldots, s_n]$ for any positive integer m
- **(Infinite horizon) discounted reward based utility** : In this setting, the reward one step into the future is discounted by a factor γ, the reward two steps ahead by γ^2, and so on. The goal is to continue acting (without an end) while maximizing the expected discounted reward. The discounting allows us to focus on near term rewards, and control this focus by changing γ. For example, if the rewards depend only on the states, the utility function is $U[s_0, s_1, \ldots] = \sum_{k=0}^{\infty} \gamma^k R(s_k)$.

While tempting for its simplicity, the finite horizon utility is not applicable to our context, as it would drop time-invariance property of the Markov Decision Model in terms of the optimal behaviour. Indeed, as there is a finite step (time), the behaviour of the agent would become non-stationary: it would not only depends from which state it is located, but also on which step we are, on how far we would be from such ending horizon. So if you arrive to a certain state, and you just have one step to go, you may decide to take a very different step versus if you have still many steps to go and you can do a lot of different things. So for instance, if you just go one step to go, you may go to extremely risky behaviour because you have no other chances.

We will then employ the discounted reward utility, where γ is the **discount factor**, a number between 0 and 1 that measures our "impatience for the future", how greedy we are to get the reward now instead of tomorrow. In economy it is often given as $\frac{1}{1+r}$ in discrete time applications (as here) or $\frac{1}{e^r}$ in continuous time ones, where r is the "discount rate" and we would then say that the utility is the net present value of this infinite serie of future rewards (a bit like the value of some land is equal to the discounted annuity you can get from that land, i.e. renting it or using it directly.)

But why the discounted reward is bounded?

The discounted utility is $U = \sum_{k=0}^{\infty} \gamma^k R(s_k)$ but we can write the inequality $U = \sum_{k=0}^{\infty} \gamma^k R(s_k) \leq \sum_{k=0}^{\infty} \gamma^k R_{max}$ where R_{max} is the maximal reward obtenible in any state, and then taking it outside the sum and using the geometric series poperties that $\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$ for $|\gamma| \leq 1$, we can write that $U \leq \frac{R_{max}}{1-\gamma}$

Going back to the land example, even if you can use or rent some land forever, its value is a finite one, indeed because of the discounting of future incomes.

17.5. Policy and Value Functions

We now define a last term, “policy”.

Given an MDP, and a utility function $U[s_0, s_1, \ldots, s_n]$, a **policy** is a function $\pi : S \rightarrow A$ that assigns an action π to any state s. We denote the optimal policy π_s^* as the optimal action you can take in a given state, in term of maximising the expected utility $\pi_s^* : argmax_{a_s} E[U(a_s)]$.

Note that the goal of the optimal policy function is to maximize the expected discounted reward, even if this means taking actions that would lead to lower immediate next-step rewards from few states.

The policy depends from the structure of the rewards in the problem. For example, in our original robot example, let’s assume that for each action we have to pay a small price, like 0.01. Then the “policy” for our robot would likely be (it then depends from the transitions probabilities) to go round the table to arrive to the $+1$ reward without passing for the -1 one. But if instead the reward structure is such that we “pay” each action 10, the policy would likely be to go instead direct to the top-right cell.

Again, we want this policy to be independent on any previous actions or the time step we are, just a function of the state where we are.

When we are talking about solving MDPs, we’re talking about finding these optimal policies.

This problem of solving the MDP is very similar to the reinforcement learning problem, but with the sole difference that when you have an reinforcement learning problem, you’re not provided with a transition function and the reward function until you actually go in the world, experience, and collect it. But for today, we assume these are given.

We will now introduce the Bellman equation. We use it so somehow propagate our rewards.

In our robot example, considering only the $+1$ and -1 rewards as given, we need a tool to formalise the intuition that the cell adjacent to the -1, while not having any reward by itself, it is a “great place to be”, as it is only one step away from the -1 reward, the top-left cell a bit less great place, and the bottom-left cell an even less great place.

In other words, we need to provide our agent with some quantification is, how good is the state, which is its value, even if the reward is coming many steps ahead. So we need to introduce some notion of this value of each state, and what the Bellman equations do, they actually connect this notion of this value of the state and the notion of policy.

The **value function V(s)** of a given state s is defined as the expected reward (i.e. the expectation of the utility function) if the agent acts optimally starting at state *s*.

Setting the timing in discrete time modelling

One very important aspect of discrete time modelling with discounting is to make clear “when the time starts” and if rewards/costs are beared at the beginning or at the end of the period.

In our case the time start clocking when we leave the departing state. At that time we pay any cost incurred in the action and cash the reward linked to state 1. After one period (so discounted with γ) we move to the second state paying at that moment the action cost and the reward linked to that state and so on.

Mathematically: $U = R(a_{0\rightarrow1}, s_1) + \gamma * R(a_{1\rightarrow2}, s_2) + \gamma^2 * R(a_{2\rightarrow3}, s_3) + \ldots$

17.6. Bellman Equations

Let’s introduce two new definitions:

- the **value function** $V^*(s)$ is the expected reward from starting at state s and acting optimally, i.e. following the optimal policy
- the **Q-function** $Q^*(s,a)$ is the expected reward from starting at state s, then acting with action a (not necessarily the optimal action), and acting optimally afterwards.

The first of the two Bellman equations relate the value function to the Q function in terms that the value function is the Q-function when the optimal a is followed:

$V^*(s) = \max_a Q^*(s,a) = Q^*(s, a = \pi^*(s))$ where π^* is the optimal policy.

The second Bellman equation relates recursively the Q-function to the reward associated to the action cost and the destination reward (girst term) plus the value function of the destination state (second term):

$$Q^*(s,a) = R(s,a,s') + V^*(s')$$

When we consider the discounting and the transition probabilities it becomes $Q^*(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$

Expressed in these terms the value function becomes:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$$

This equation encodes that the best value I can take from this state s to the end can be divided into two parts: the rewards that I am getting from taking just one step, this is the reward, plus whatever reward I can take from the following step multiplied by the discounting factor.

17.7. Value Iteration

Given the Bellman equations would be tempting to try to compute the value of a state starting from the values of the reachable states from the optimal action (that, in turn, can be just picked up looking at all the possible actions).

The problem however is that the V(A), the value of state A, then may depends from V(B), but then V(B) may depends as well from V(A) and many more complicated interrelations. Aside very simple cases, this make impossible to find an analytical solution or even a direct numerical one.

The trick is then to initialise the values of the states to some values (we use 0 in this lecture) and then starting iteratively to loop to define the value of a state at iteration (k) from the value of the reachable states *that we did memorised in the previous iteration* from the optimal action :

$$V^*(s_k) = \max_a \sum_{s\prime} T(s,a,s')(R(s,a,s') + \gamma V^*(s'_{k-1}))$$

The algorithm than stop when the differences from the values at k iteration and those at $k+1$ become small enough on each state.

Note that $V^*(s_k)$ can be interpreted also as the expected reward from state s acting optimally for k steps.

Once we know the final values of each state we can loop over each state and each possible action on each state one last time to select the “optimal” action for each state and hence define the optimal policy:

$$\pi^*(s) = argmax_a Q^*(s,a) = argmax_a \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma V^*(s'))$$

And pretty much what this algorithm is doing, it is propagating information from these reward states to the rest of the board.

Convergence

While the iterative algorithm described above converges, we are now proving it only for the simplest case of a single state and single action.

In such case the algorithm rule at each iteration would reduce to:

$$V^*(s_k) = (R(s) + \gamma V^*(s_{k-1}))$$

In a fully converged situation the Bellman equation gives us that:

$$V^*(s) = (R(s) + \gamma V^*(s))$$

Putting them together we can write:

$$(V^*(s) - V^*(s_k)) = \gamma * (V^*(s) - V^*(s_{k-1})).$$

That is, at each new iteration of the algorithm, the difference between the converged value and the value of the state at that iteration get multiplied by a factor γ, that being a number between 0 and 1, it means this difference reduces at each iteration, implying convergence.

17.8. Q-value Iteration

Instead of computing first the *values* of the states and then the Q function to retrieve the policy, we can directly, using exactly the same approach, compute the Q function. the algorithm then is named **Q-value iteration** and is the one we will use in the context of reinforcement learning in the next lecture.

Using the Bellman equations we can write the Q function as:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

The iteration update rule than becomes:

$$Q^*_{k+1}(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*_k(s',a') \right)$$

Lecture 18. Reinforcement Learning 2

18.1. Revisiting MDP Fundamentals

We keep the MDP setup as in the previous model, and in particular we will work in this lecture with the Bellman equation in terms of the Q-value function:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

We will however frame the problem as a reinforcement learning problem where we need to find the best policy knowing a priori only the sets of states S and possible actions A but not those of the transition tables T and the rewards R. These will be instead estimated as posteriors when we start "acting in the world" (collecting data), seing where do we actually end up from each (state, action) and the rewards we may ended up with.

In this setup, you see that now actually acting in the world becomes an integral part of our algorithm. We can try as much as we wish, but eventually we need to provide the policy. So how are we doing these trials? Which and what to try will be an integral part of the algorithm.

18.2. Estimating Inputs for RL algorithm

So, out of the tuple $< S, A, T, R >$ that constitutes a Markov Decision Process, in most real world situation we may assume to know the sets of the states S and the possible actions A, but we may not know the transition possibilities T and the rewards R.

We may be tempted to start by letting our agents running with random decisions, observe where they land after performing rule a on state s, and which reward they collect, and then estimate T and R as:

$$\hat{T}(s,a,s') = \frac{\text{count}(s,a,s')}{\sum_{s''} \text{count}(s,a,s'')}$$

$$\hat{R}(s,a,s') = \frac{\sum_{t=1}^{\text{count}(s,a,s')} R_t(s,a,s')}{\text{count}(s,a,s')} \text{ (the average of the rewards we collect)}$$

The problem with this approach is however that certain states might not be visited at all while collecting the statistics for $\hat{T}, \hat{R}$, or might be visited much less often than others leading to very noisy estimates of them.

We will then take an other approach where we will not only passively collect the statistics during the training, but as we start “learning” the parameters we will also start guide our agents toward the action we need more to collect more useful data.

For this instance it is important to distinguish between model based approaches and model free learning approaches.

Instead of defining them, let’s take an example.

Let’s assume a discrete random variable X and our objective being to estimate the expected value of a function of it $f(X)$.

Model based approaches would try to estimate first the PMF of X $p_X(x) and then estimate the function $f(X)$ using the *Expected value rule for function of Random variables*:

$\hat{p}_X(x) = count(x)/K$ (where K is the number of samples)

$$E[f(X)] \approx \sum \hat{p(x)} f(x)$$

Model free approaches would instead sample K points from $P(X)$ and directly estimate the expectation of $f(X)$ as:

$$E[f(X)] \approx \frac{\sum_{i=1}^{K} f(X_i)}{K}$$

The “model free” approach is also known with “sample based” approach or “reduced” approach.

18.3. Q value iteration by sampling

We are left hence with two problems. The first one is to actually find how to incorporate in a live fashion our experience to our estimate, updating our estimation of the missing parameters of the model one record after the other, while observing the agent performing, doing an action in a certain state and ending up in a certain state eventually collecting a certain reward.

The second one is actually how to direct the agent while collecting the data, still with limited knowledge of the model.

We will tackle the first problem in this segment, assuming we know how to direct the agent, and we will come to the full picture in the next segment.

Exponential running average

But before we start the segment, let’s quickly introduce a new notion which is called **exponential running average**, which is a way to do averaging, but which is different from the standard averages that we are used to, because it weighs differently the significance of sample. It gives more weight to the current and most recent samples compared to the prior samples that we’ve collected.

If a normal average is $\bar{x}_n = \frac{\sum_{i=1}^{n} x_i}{n}$, the exponential running average is defined as $\bar{x}_n = \frac{\sum_{i=1}^{n} (1-\alpha)^{(n-i)} x_i}{\sum_{i=1}^{n} (1-\alpha)^{(n-i)}}$ with α being a value between 0 and 1.

The exponential running average is normally used in its recursive nature as $\bar{x}_n = (1-\alpha)\,\bar{x}_{n-1} + \alpha\, x_n$.

This formulation of the exponential average gives us a mechanism to take our prior estimate of the running average and add to it the new sample that we are taking.

Sample-based approach for Q-learning

We are now ready to live-updating the estimation for the Q-value algorithm.

As we don’t know the transitions and rewards ahead of time, the Q-value formula will now be based on the samples. We take here a sample-based approach and directly think to what happens to Q when I start taking samples.

Let’s recall the formula from Ballman equations that relate a given Q value to those of the neighbouring (reachable) states:

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right)$$

What so I obtain when I observe my first $sample_1$ of an action done in state s that lead me to state s'_1 ?

The Q-value(s,a) becomes $R(s,a,s'_1) + \gamma \max_{a'} Q^*(s'_1,a')$

More in general, when I observe a k-sample of the same (s,a) tuple (action a on state s) I would observe Q-value$(s,a) = R(s,a,s'_k) + \gamma \max_{a'} Q^*(s'_k,a')$

Of course at this point we don't yet know what $\max_{a'} Q^*(s'_k,a')$. When we will run the iteration algorithm, it will be the value of the previous iteration.

Now, to estimate my $Q(s,a)$ I can just average over these k samples.

Using normal average it would result in :

$$Q^*(s,a) = \frac{1}{k} \sum_{i=1}^{k} Q_i^*(s,a) = \frac{1}{k} \sum_{i=1} \left(R(s,a,s'_i) + \gamma \max_{a'} Q^*(s'_i,a') \right)$$

Estimating directly the $Q(s,a)$ we don't need to estimate firstly the T or R matrices.

The problem remains however that after we observe $Q(s,a)$ leading to $s\prime$, we may not be able to go bask to state s, it's not like rolling a dice I can try 1000 times the same experiment.

So we don't want to have to wait until we can sample it all and then average. What we want to do it to updates of our Q-value each time when we get new sample, to do it incrementally.

And here we can use the exponential average to compute differently our estimates for Q-values, in an incremental fashion.

The Q-value for (s,a) at sample $i+1$ would then be equal to

$$Q_{i+1}(s,a) = (1-\alpha) Q_i(s,a) + \alpha * sample_i(s,a)$$
$$Q_{i+1}(s,a) = Q_i(s,a) - \alpha (Q_i(s,a) - sample_i(s,a))$$
$$Q_{i+1}(s,a) = Q_i(s,a) - \alpha (Q_i(s,a) - (R(s,a,s'_i) + \gamma \max_{a'} Q^*(s'_i,a')))$$

where s_i is the state I ended up in sample i.

Attention that here $Q_i(s,a)$ is not the Q-value of the sample i, but the average Q-value up to the sample i including. It already embed all the experience up to sample i.

The above equation should recall those of the stochastic gradient descent, where we have our prior estimate, plus alpha, and the update factor. It's just a different form of this update.

The algorithm for Sample-based approach for Q-learning would then be something like

- Initialise all $Q(s,a)$, e.g. to zero
- Iterate until convergence (e.g. until small difference in values between consecutive steps)
  - Collect a sample at each step --> s_i, R(s,a,s_i\prime),
  - Update the Q-value based on the sample as in the above equation.

We can demonstrate that the same conditions on alpha that we used to demonstrate convergence with the gradient descent algorithm are exactly the same that can be shown that apply here. So under this condition, this algorithm is guaranteed to converge (locally).

18.4. Exploration vs Exploitation

Now that we learned how to incormorate our Q-value of the single samples in our $Q(s,a)$ matrix, we are left with the only problem of how to direct our agents during learning, i.e. which policy to adapt.

We are here introducing a very common concept in reinforcement learning (RL), those of Exploration versus Exploitation.

- **Exploration** is when we try to behind the "usual way", when we want to discover new things that may be better than what we already know. In real life is trying a new food, take a new road, experiment with a new programming language. In RL this means trying (e.g. randomly) actions

that we don't yet have data for or where our data is limited (like having no institutional memory). As in real life, exploration in RL makes more sense when we don't have yet much experience, to indeed help build it.
- **Exploitation** instead is acting rationally based on the acquired information, try to make the best action based on the available information. It is being very conservative, follow always the same routine. In RL it is simply acting by following the policy.

Epsilon-greedy

To allow in RL more exploitation at the beginning, and more exploitation in successive steps, our model can include a parameter ϵ that gives the probability of tacking an action randomly (vs those maximising the estimated q-value), and have it parametrised such that ϵ is highest at the beginning and reduces as the experiment continues, until becoming negligible when the q-value start to converge, i.e. at that time we follow mostly the policy.

Lecture 19: Applications: Natural Language Processing

19.1. Objectives

- Understand key problems/research areas in Natural Language Processing field
- Understand challenges in various NLP tasks
- Understand differences between symbolic and statistical Approaches to NLP
- Understand neural network based learnt feature representations for text encoding

19.2. Natural Language Processing (NLP)

This lesson deals with Natural Language Processing (NLP), one important application of machine learning technology.

Natural language Processing is all about processing natural languages and it frequently involves speech recognition, natural language understanding, natural language translation, natural language generation etc.

It can be doing something very complicated or something relatively simple, as string matching. An application doesn't need to imply *understanding* of the meaning of the text to be considered doing natural language processing. For example the spam filter in gmail is a "simple" classifier, it doesn't need to understand the text of the email to judge if it is spam.

History

The importance that language processing undergoes in the Artificial Intellingence field is well described by being the core of the famous *imitation game* of the **Turing test** to distinguish (and defining) an intelligent machine from a human:

"I believe that in about fifty years' time it will be possible to programme computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70 percent chance of making the right identification after five minutes of questioning." (Alan Turing, 1950)

In 1966, Joseph Weizenbaum at MIT created a program which appeared to pass the Turing test. The program, known as **ELIZA**, worked by examining a user's typed comments for keywords. If a keyword is found, a rule that transforms the user's comments is applied, and the resulting sentence is returned. If a keyword is not found, ELIZA responds either with a generic riposte or by repeating one of the earlier comments.

Status

The "success" of Natural Language Processing (NLP) technology largely depends on the specific task, with some tasks that are now relatively successful, others that are doing very fast progress, and others that even now remain very complex and hard to be realised my a machine.

Mostly solved tasks

- Spam detection
- Named Entity recognition (NER): extract entities from a text, like location or a person's name of a news

- Part of speech (POS) tagging: recognise in a sentence which word is a verb, which a noun, which an adjective…

Making good progress tasks

They were considered very hard, almost impossible to solve tasks, until a few decades ago.

- Machine translation: while far from perfect, many people now find it useful enough to use it
- Information extraction (IE): translate a free text into a structured representation, like a calendar event from an email or filling structured assessment of a free-text feedback on a medicine
- Sentiment analysis: distinguish, again from a free text, the sentiment, or the rating, over some subject
- Parsing: understanding the syntactic structure of the sentence.

Still very hard tasks

- (Uncostrained) question answering: except questions for which an almost string-matching approach could solve it (*factoid questions*, like “when Donald Trump has been elected”, it is easy find a sentence indicating “when Donald Trump has been elected”) , questions that really require analysis and comparison are very hard to solve
- Summarisation: extracting the main points out of a larger context
- Dialog: when we go outside of a specific domain, developing a human/macine dialogue is extremely difficult

19.3. NLP - Parsing

The above classification in three buckets of “readiness” of AI technologies for different tasks is however questionable for three reasons.

- *Measure problem*. First the “performance” depends on the measure we are looking. For example, in an experiment of text parsing, each single relation between words (“arcs”) has been found correctly by the machine 93.2% of times. Great. However, if we look at the times the whole sentence had all the relations correct (i.e. all arcs correct), the machine accuracy was below 30%. The beauty is in the eye of the beholder, it depends how you calculate your performance.

- *Data problem*. The second reason is that often it is not the algorithm to perform bad, but the availability of data for the area of the problem that is poor. For example machine translations are highly improving year after year and doing quite a good job, but still a food recipe would be badly translated. This because machine translations are trained mostly on news corpus, and very little on “food recipes” subjects. So it’s not that the task is hard, it is hard in the context of training data available to us. The difficulty of the task should always be assessed in the context of the training data available.

- *Decomposing sources of “difficulty”*. Third, when we are assessing the difficulty of the questions, we’re thinking about ourselves. How difficult is for us as humans, for example how much time it will take you to discover the answer in a question answering task. But since the machine operates and can very easily perform big searches, powerful word matches, an answer for which somewhere, even remotely, there is a almost exact matching in the data, is an easy task for the machine and could be a difficult one for a human. Conversely, if the task involve connecting parts together, “reasoning”, because the information that it needed to connect the dots is not readily available, like in text comprehension (you are given a paragraph, a question about it and a set of plausible answers to choose from) or summation/making highlights, there machines are doing very poorly. But, in connection with the previous point, if we will provide machine with a million news stories of the sort, together with their summaries/highlights (many news agencies, in addition to their stories, provided story highlights, where they extract some sentences or close paraphrases of the sentences as a highlight from the story), the pattern will be there and the machine will find it. So this whole assessment of asking machines of doing processing and compare it with humans. They just have different capacities and capabilities. And we need always to analyse the performances in context.

One area NPL is doing very useful things is in classification, where helps categorise free text from scientific medical articles to help summarise them and building evidence-based medical practices, find the distribution of different opinion on medical issues from the analysis of the individual article and associated medical experiments.

19.4. Why is NLP so hard

In short: due to different kind of ambiguity intrinsic in natural languages.

NLP in the industry (already operational)

- Search
- Information extraction
- Machine translation
- Text generation
- Sentiment analysis

Often we are not even aware we are interacting with NLP tecnologies, like when a machine read GRE or SAT exam essays (still, an human read it too) or a company provides automatised analysis/report on data, and how much this technology impact already our daily life (it is worth however consider some of the criticisms, on the risks of potentially amplifies biases and assumptions in "robo-journalism" software).

There are lots and lots of different opportunities, and it's a very fast-growing area. You can have a job, but also make an important social difference, for example automatising the creation of a US Mass Shooting database from news sources.

We turn now to the question: why machines are not yet perfect? Why they can't understand language perfectly in general settings (opposite to narrow applications)?

Natural language, contrary to programming language, is ambiguous in its nature, with words holding different meaning in different contexts ("Iraqi head seeks arms") or the scope of the syntactic phase may be ambiguous ("Ban on nude dancing on governor's desk").

But because we kind of understand contextually we actually can get to the correct interpretation (nobody is trying to dance nude on governor's desk).

A machine actually has much more difficulty when it's trying to disambiguate a sentence.

Note however that these could be ambiguous sentences for humans as well, as when the air traffic control center communicated a "Turn left, right now" leading the pilot to misunderstand and crash… or so was reported by a French linguistic to claim superiority of the French language over the English one…).

An other example of an ambiguous sentence: "At least, a computer that understand you like your mother"

The first is an **ambiguity at the syntactic level** (syntax: how the different pieces of the sentence fit together to create meaning): it is the computer that understand you like your mother does, or it is that the computer understand that you like your mother ? Each of this interpretation would correspond to a different parsing of the original tree.

The second is a **word-sense ambiguity** (aka "semantic ambiguity"): the word "mother" means both your female parent but also a stringy, slimy substance consisting of cells and bacteria (for example the dried mother yeast added to flour to make bread or those added to cider or wine to make vinegar).

The third one is an **ambiguity at discourse level** (*anaphora*)(discourse: how the sentences are tight together): because sentences do not show up in isolation., there could be ambiguity on how the pieces actually show up in a context of other sentences. For example "Alice says that they've built computer that understands you like your mother, but she … doesn't know any details … doesn't understand me at all". Here the personal pronoun "she" may co-refer to either Alice (most likely in the first case) or the mother (most likely for the second case) .

Finally, some ambiguities varies across languages. Depending on the specific language that you are operating, some things will be harder and some things will be easier. **Tokenisation** (identification of the units in the sentence, a sub task in any NPL task) is easy in English language, thanks to space separation, puntuation), much harder in some Asian or Semitic languages. Same for **named entity detection**, easy in English due to strong hint given by capitalisation ("Peter"), more challenging in other languages.

19.5. NLP - Symbolic vs Statistical Approaches

Symbolic approaches usually encode all the required information into the agent whereas statistical approaches can infer relevant rules from large language samples.

Knowledge bottleneck in NLP

to perform well in NLP tasks we need both:

- Knowledge about the language
- Knowledge about the world

We can employ two possible solutions:

- **Symbolic approach**: encode all the required information about the area into the agent, so that this can act based on these "rules"
- **Statistical approach**: infer language properties from large language samples

Symbolic approach was very common in early AI, and structured rules for "languages" to be used in specific domain were developed. This was in part the result of the Chomsky critic (1957) that a statistical approach could never distinguish between a nonsensal, never used before, grammatically correct sentence and one that other than nonsense would have been also grammatically incorrect. This view lead to the development of systems like SHRDLU or LUNA that, for very limited domain (like instructiong a robot to move boxes around using a natural language) looked promising. These systems however didn't scale up for larger domains, as to operate these tecnologies you need to be backed by a full model of the world, and then natural language is done on top of this reasoning. The problem was that you really need to encode by hand, all of those relations, there was no learning. Not a scalable solution clearly.

It was only in the early nineties, with teams lead by Prof Jelinek, that the statistical approach really started to take off. In 1993 it was released the "Penn TreeBank", a parsed and manually annotated corpus of one milion words from 2499 news stories from the Wall Street Journal (WSJ). So now the machine had a sentence and had a corresponding parse tree and concepts as training, learning and evaluation could rise in the NLP domain.

To illustrate the difference between statistical thinking and symbolic thinking, let's look at the task of determiner placement, i.e. given a text where determiners has been omitted find their correct placement. Determiners in English are, simply put, words that introduce a noun. They always come before a noun, not after, and they also come before any other adjectives used to describe the noun. Determiners are articles, but also demonstrative or possessive pronouns, or quantifiers ("all", "few", "many"…)

A symbolic approach would try to follow grammar rules for their placement, but there are so many rules… the placement of determiners depends by the type of noun, whether it is countable or not countable, and then in turn this bring back the concept on what is a number, whether it is unique … end, still, so many exceptions!

It is so hard to encode all this information, and it's very hard to imagine that somebody would really remember all these rules when they move to a new language.

So now, let's think about it in a way, the statistical approach, which is the way we're thinking about this problem after this class. Given many labelled texts, for each noun we can encode features as it is singular or plural, it is the first appearance? We can ask many feature type of question, and then we can just run a classifier and get very good results.

So this illustrates how we can take a problem, which is really hard to explain directly with the linguistic knowledge, with world knowledge, and cast it instead in a very simple machine learning problem and get a satisfactory solution.

Note however that the ML approach is conditional to the scope.

We're not making a claim that machines really understand the whole natural language, but we're just saying we want the particular task to do, let's say, spam prediction. We're going to give a lot of instances of material related to spam, and machines will learn it. They still don't understand language, but that's something useful.

18.6. Word Embeddings

Statistical approaches using classifiers over manually encoded features are great, but the job remains still hard.

One of the problem is that there is still a severe issues related to sparsity problem. If we take at unigram, the coverage improves, it's great. However if we are looking at pairs (bigrams), meaning

we're looking at a bigger construct because they have more information, you can see that their frequency of occurrences (of arcs in parsing) is very low.

And even if we increase our training size, we still cannot get good frequency count.

And even if you compare across languages. Some languages, like English, are not very morphologically rich, words will repeat many times. But in others, like in Russian, German or Latin, because of the case system, words get different endings. So if you say the dog, depending how the dog appeared, will have different ending. As consequence, such languages will not have sufficient frequency, will have a lot of sparse sounds, and their performance, keeping equal the amount of training data, would be much lower.

Neural networks have improved earlier classifiers because removed the need to manually encode features for words People, before neural models came into play, were spending a lot of time designing what was a good pattern, a good feature representation. Neural networks actually eliminated the need to do this kind of hand engineering in feature selection, replacing it with a "simple" bag-of-word approach (a one-hot encoding measuring presence or absence of each single word in the sentence using a vector of dimensions equal of the language vocabulary) or with the similar frequency representation.

Still there is a problem with such approach, as the bag of word representation can't encode the "distance" between two words. In such encoding there is no difference in a "distance" between a bulldog and a beagle on one side, and a bulldog and a lipstick (there is not even a metric indeed)

So, we want instead to move from this view of word embedding and translate it instead into a dense continuous vector, in such a way that now the distance (and specifically, the cosine distance) between the word dogs bulldog and beagle will be very close to each other.

Now we don't need in some ways annotated supervision. We can just take large amounts of English text and create models designed for language modelling tasks, where, given some partial sentence, you need to predict which word will come next.

We would like to learn word embeddings that are much less sparse than one hot vector based encoding because reducing the sparsity of input features lowers the sample complexity (number of training examples required to do an accurate task) of the downstream text classification task.

In order to do the above, we should cluster the similar or related words together in the embedding dimension space. For instance, the words "bulldog" and "beagle" must have similar embedding representations than "bulldog" and "lipstick". One way to learn word embeddings is by maximizing cosine similarity between words with related meaning. Word embeddings are practically very useful because they can be learnt without any significant manual effort and they generalize well to completely new tasks.

If you run a PCA and select two dimensions (for visualisation) over this embedding of the words, you will note that related terms will be clustered together, even if the machine don't know anything about them.

Using Recurrent Neural Networks we can construct a representation for the whole sentence (rather than individual words) and then use the resulting vector representation, which at that point will embed the meaning of the sentence, for translations, "decoding" it in an other language.

RNN works with stream of data, taking the current context, adding a new word, and then producing a new context. And we train them based on the task at hand, whichever task we have to solve. Because here, we will estimate all this ways based on the classification task we want to resolve. And maybe depending on different tasks, we're going to get different vectors.

LSTM or Recurrent Neural network based approaches encode an input sentence into a context vector capturing more than just a summation of its constituent parts together.

Having the whole meaning of a sentence as one vector it's so powerful that you not only can solve a classification task, you can actually stick to it something which is called decoder, which can take this vector and then roll it in a similar fashion and generate a sentence. And in this case, if your decoder is trained on Spanish, you would be able to start with Spanish, translate it into this vector, as so that was English translated into a vector, and then decode it in Spanish or in other language. And that's exactly how this encoder decoder architectures are used today in machine translation.

There are lots and lots of exciting things that are happening in NLP everyday. This is a very fast evolving field nowadays!

Some other resources on reinforcement learning:

- Sutton and Barton (2018) Reinforcement Learning: An Introduction, 2nd decision(Free textbook)
- Kaelbling, Littman and Moore (1996) Reinforcement Learning: A Survey (Survey paper)

Homework 6

Project 5: Text-Based Game

P5.

High level pseudocode of the provided functions in agent_tabular_ql.py:

- main():
  - load game constants
  - for NUM_RUNS:
    - run() --append–> epoch_rewards_test (list of list of floats):
      - q_func = 0
      - for NUM_EPOCHS:
        - run_epoch() --append–> single_run_epoch_rewards_test (list of floats):
          - for NUM_EPIS_TRAIN:
            - run_episode(for_training=True) # update q_func
          - for NUM_EPIS_TEST
            - run_episodes(for_training=False) --append–> reward
          - return mean(reward)
      - return single_run_epoch_rewards_test
  - transform list of list of float epoch_rewards_test in (NUM_RUNS,NUM_EPOCHS) matrix
  - plot NUM_EPOCHS agains mean(epoch_rewards_test) over NUM_RUNS

P5.7. Introduction to Q-Learning with Linear Approximation

Intuition for the approximation of the state/action spaces

MDQ and Q-learning are great tools, but they assume a categorical nature of the states/action spaces, i.e., each state/action is treated categorically, where each item is completely independent from the other. In learning, observing the result of state/action $(s1, a1)$ doesn't influence $(s2, a2)$. Indeed, there is no concept of "distance" between states or actions. The sets of states/action itself is not ordered.

In many problems however this is a suboptimal representation of the world. For example, if I am in a state `1000€ saving` and do action `100€ consumption`, intuitively I could borrow the learning from that state/action pair to the (state `1001€ saving`, action `101€ consumption`) pair, as the outcome shouldn't be too much different, or at least should be less different than (state `200€ saving`, action `199€ spending`).

And their (linear or non-linear) approximation try to exactly first learn, and then exploit, this relation between elements of the state/action spaces. In project 5, setting a feature vector based on bag of words allows to introduce a metric on the sentences, where sentences that change by just a word are closer than completely different sentences.

This is really cool, as it allows to map continuous spaces to discrete features and apply a MDP on top of it, well intended, if the nature of the problem admits a metric across these spaces.

Q-Learning with Linear Approximation

We need to create a vector representation for any possible states. In our case, being the states textual description, one immediate way is to use a bag of words approach, that, given a dictionary of size nV (i.e. of the vocabulary for the language of the sentences), returns for each concatenated description of room + description of quest, a fixed length representation as a vector of size nV, with `1` if the word in the dictionary is present in the concatenated sentence, `0` otherwise. We call this `state_vector`.

The idea is that nV should be much smaller of the possible number of original states (descriptions).

Given nA the number of actions, we then set a matrix θ of (nA, nV) that represents the weights we need to learn, such that the q_values associated to all the possible actions of a state s is $q\_value(s) = \theta * \text{state\_vector}_s$, that is the matrix multiplication between the weights θ and the (column) vector representation of the specific state. The resulting vector has length nA, where each individual element is the q_value for the specific (state,action) pair $q\_value(s, a_i) = \sum_{j=1}^{nV} \theta_{i,j} * \text{state\_vector}_s$.

Our tasks is then to learn the θs, and we can use a squared loss function applied to our flow of data coming from the game, where the value of the "estimated" y is the q_value so computed (from the thetas), and the one acting as the "real" one is those computed using the Bellman equation for the q_value: $y = q\_value(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a', \theta)$ where s' is the destination state observed as result of the action.

The loss function is then $L(\theta; s, a) = \frac{1}{2}(y - Q(s, a, \theta))^2$.

The gradient with respect to the thetas *of that specific action row* is then $g(\theta) = \frac{\partial}{\partial \theta} L(\theta) = (Q(s, a, \theta) - y) * \text{state\_vector}_s = (Q(s, a, \theta) - R(s, a) - \gamma \max_{a'} Q(s', a', \theta)) * \text{state\_vector}_s$ (the thetas in the max q_value for the destination state are here as given).

Finally the update rule of the gradient descent is then:

$$\theta \leftarrow \theta - \alpha g(\theta) = \theta - \alpha \left[Q(s, a, \theta) - R(s, a) - \gamma \max_{a'} Q(s', a', \theta)\right] * \text{state\_vector}_s$$

Note that the `state_vector` so described encodes only the states, but one can choose an other form where both states and actions are represented, making it a matrix rather than a vector, and still using a matrix multiplication with a weight matrix in order to get a linear approximation of it.

Q-Learning with non-linear Approximation

And we can now think of why do we even have a linear approximation for the q_value ? Can we not parametrize it by using a deep neural network, where we plug in the state and the action as a specific vector representation, this vector representation goes through a neural network, which gets not necessarily just an inner product with the (unknown) parameters, and the out comes as the corresponding q_value ? And at this point we can train the parameters associated with that deep neural network in the same way as we did for the linear case, with a loss function where we compare the q_value obtained by the neural network in evaluating (s,a) with those arising from the Bellman equation.

This is then why we don't want to store potentially all possible (state,action) pairs. Instead, we will just be storing a parameter theta, and every time we want to compute the q_value, we kind of come up with a vector that represents the (state,action) pair, and then feed it through either a linear function or a neural network, and get the q_value(s,a) that we're supposed to get.

[MITx 6.86x Notes Index]