# Artificial Intelligence

Augustin Cosse.


Fall 2020

December 9, 2020

# So far

- Simple reflex, random agents, Utility based, Goal based
- Improvement through Search Methods (uninformed (DFS, BFS), informed (BS, A$^*$)).
- Logical Reasoning, Propositional logic + First Order Logic, Inference

# This week

- Human cognition consists of three remarkable capabilities: perception, learning, and reasoning,
- Modern artificial intelligence (AI) systems exhibit all three of those abilities
- Machine learning techniques such as deep neural networks have achieved extraordinary performance in solving learning/perception tasks
- Meanwhile, logic-based AI systems have succeeded in demonstrating human-level reasoning abilities in theorem proving and inductive reasoning.

# Reasoning

- Reasoning is the generation or evaluation of claims in relation to their supporting arguments and evidence
- Reasoning skills are also crucial for being able to generate and maintain viewpoints or beliefs that are coherent with, and justified by, relevant knowledge
- What really separates learning from reasoning is the ability for the technology to become smarter over time. Not just from logical entailment, but from new, possibly unexpected pairs of observations and feedbacks.

# Learning vs Reasoning

- An agent is learning if it improves its performance on future tasks after making new observations about the world.
- There are several reasons why we might want our agent to learn.
  - First the designer cannot anticipate all possible situations that the agent might find itself in
  - Second, the designer might not be able to anticipate all changes over time. A program designed to predict tomorrow's stock market prices must learn to adapt when conditions changes from boom to burst
  - Finally, the designer often has no idea how to program a solution. In this case it is therefore better to let the agent learn by itself, and improve through the data it acquires.

# Forms of learning

- Any component of an agent can be improved by learning from data
- The improvements depend on four main factors:
  - Which component is to be improved
  - What prior knowledge the agent has
  - What representation is used for the data and the component (e.g. a neural network (component) is used to learn to recognize faces stored as images (data))
  - What feedback is available to learn from (e.g. is it possible to know whether a guess is correct or not ? –¿ supervised vs unsupervised)

# Components to be learned

- So far we have discussed several approaches at designing an agent. Among the components we discussed, several of them could be learned.
  - A first component that we might want the agent to learn is the mapping between its internal representation of the environment and the actions it can take
  - Another component that one might want the agent to learn is a means to infer relevant properties of the world from its percept sequence
  - Finally, besides learning the mapping between the representation of the environment and the optimal action to take, it can sometimes be useful to learn a more general mapping between each action and its value in the long run

# Representation and Prior knowledge

- We have already covered two knowledge representations, when learning to code logical agents: Propositional Logic and First Order Logic
- When discussing learning in uncertainty, we will introduce yet another knowledge representation approach known as Bayesian networks.
- In this third part, our interest will focus on mapping between inputs that are encoded through a factored representation (a vector of attribute values) and outputs that can be either continuous (regression) or discrete (classification).
- The approach of learning a rule form specific input/output pairs is called inductive learning

# Representation and Prior knowledge

- The alternative, deductive learning corresponds to going from from a known general rule to a new rule that is logically entailed but useful because it allows more efficient processing

# Feedback to learn from

- There are three types of feedbacks that determines the three types of learning.
    - In unsupervised learning, the agent learns patterns in the input even though no explicit feedback is provided. The most common unsupervised learning task is clustering: detecting potentially useful clusters of input examples. An autonomous car might for example be able to discriminate between good traffic days and bad traffic days without being explicitly told that the data it gets corresponds to good or bad days
    - In reinforcement learning, the agent learns from a series of reinforcements, rewards or punishments. For example the lack of tip at the end of a journey gives the taxi agent an indication that it did something wrong. The two points corresponding to a win at the end of a chess game tell the agent that it did something right.

# Feedback to learn from

- In supervised learning, the agent observes some example input-output pairs and learns a function that maps from input to output. An autonomous cab could receive images of traffic light. In a first step, it would be provided with the explicitly meaning of the images: red light, orange light or green ligtht. In a second time, it would have to determine the meaning of a new image from the mapping between the previous images and their associated meaning it was given.

# Feedback to learn from

- The distinction between those different forms of feedback is not always so crisp. In particular some hybrid approaches are also possible

- In semi-supervised learning, we are given a few labeled samples and we must make what we can out of a large collection of unlabeled examples. As an example, imagine that the agent is given a set of images and has to guess the age of the persons whose faces have been captured by the pictures. Let us say that the agent receives some feedback (i.e. some of the people tell it explictly their age). Some of them lie though and the agent therefore has to compare the picture and the associated age information in order to detect the liars. This second component thus requires some unsupervised learning component.

# Supervised learning

- The task of supervised learning is the following:

  Given a training set of $N$ example input-ouput pairs

  $$(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$$

  where each $y_j$ was generated by an unknown function $y = f(x)$, discover a function $h$ that approximates the true function $f$

  Here $x$ and $y$ can take any value. They need not be numbers. The function $h$ is called a hypothesis

- Supervised learning can be seen as a search through the space of all hypotheses for one that will perform well.

# Supervised learning

- To measure the accuracy of a hypthesis, we give it a test set of examples that are distinct from the training set. We say that a hypothesis generalizes well if it correctly predicts
- We say that a hypothesis generalizes well if it correctly predicts the value of $y$ for new examples. Sometimes the function is stochastic (it is not strictly a function of $x$) and what we have to learn is a conditional probability distribution $P(t|\boldsymbol{x})$.
- When the output $y$ is one of a finite set of values (such as 'sunny', 'cloudy' or 'rainy'), the learning problem is called Classification (we talk about boolean or binary classification if there are only two possible values. When $y$ is a number (such as tomorrow's temperature), the learning problem is called Regression)

# Supervised learning

- In both problems, we want to learn the mapping $f$ from $x$ to $y$. Since we cannot store all possible such mappings, we need to restrict to particular classes of mappings that can be efficiently stored by the agent.
- Consequently, we will usually select our mappings from a particular hypothesis space $\mathcal{H}$.
- A particular hypothesis is called consistent when it agrees with all the data (i.e. for all training examples $(x_i, y_i)$ from $\mathcal{D}$, $h_\beta(x_i) = y_i$).
- This illustrates a fundamental problem in inductive learning: *how do we choose from among multiple consistent hypotheses?*

# Supervised learning

- One answer is to prefer the simplest hypothesis consistent with the data
- This principle is known as Ockham's razor after the $14th$ century English philosopher William of Ockham who used to argue sharply against all sorts of complications.
- In general there is a tradeoff between complex hypotheses that seem to fit the data well and simpler hypotheses that may generalize better.
- We say that a learning problem is realizable if the hypothesis space contains the true function.

# Learning Decision Trees

- Decision Trees is one of the simplest yet most successful learning algorithm
- A decision tree is used to encode a function that takes as input a vector of attribute values $A = (A_1, \ldots, A_D)$ and returns a decision (i.e. a single ouput value)
- At first we will focus on examples where the input is represented by discrete values and the output has exactly has two possible values
- An input is classified either as a positive, or as a negative example

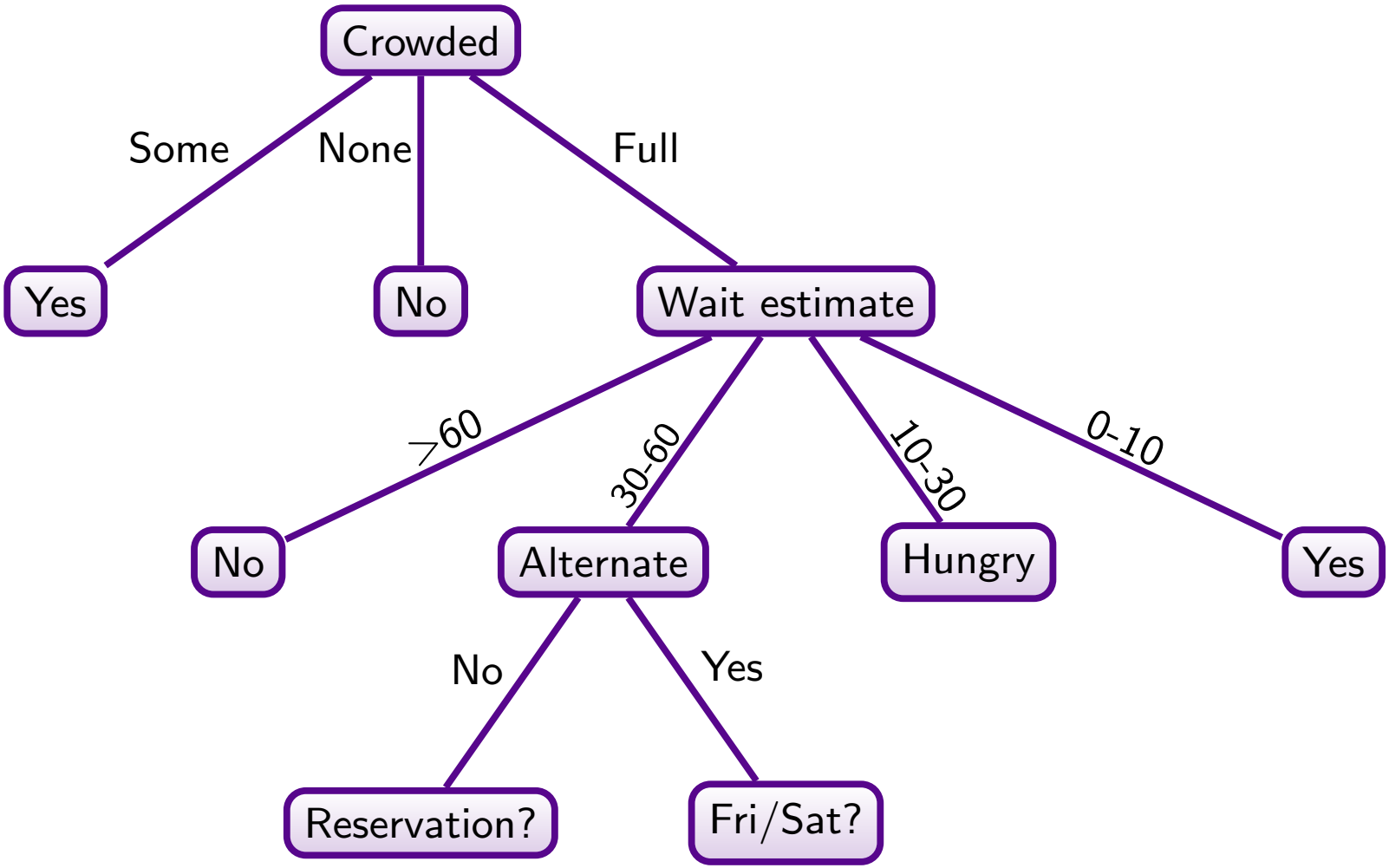
# Learning Decision Trees

- A decision tree reaches its decision by performing a sequence of tests
- Each internal node in the tree corresponds to a test of the value of one of the input attributes, let's say $A_i$, and the branches from that node are labeled with the possible values of this input attribute, $A_i = v_{i,k}$
- Each leaf node in the tree specifies a value to be returned by the function

# Learning Decision Trees

- As an example we could build a decision tree to decide whether to wait for a table at a restaurant.
- The goal here is to learn a representation for the goal predicate 'Will Wait'. We first list the attributes that will determine the output
  1. Alternate. Whether there is suitable alternative nearby
  2. Friday/Saturday. True if it is either Friday or Saturday
  3. Bar. Whether the restaurant has a comfortable bar area nearby
  4. Hungry Whether we are hungry
  5. Crowded. How many people are in the restaurant (possible values are *None*, *Some* and *Full*)
  6. ...

# Decision Tree (deciding whether or not to look for a table)



# Learning Decision Trees

- Note that a Boolean decision tree is logically equivalent to the assertion that the goal attribute is true if and only if the input attributes satisfy one of the path leading to a leaf with value true.

- Writing this in propositional logic gives

$$\text{Goal} \Leftrightarrow (\text{Path1} \vee \text{Path2} \vee \ldots)$$

  where each path is a conjunction of attributes-values test required to follow that path

- The whole expression is therefore equivalent to a disjunctive normal form.

- This also means that any function in proposional logic can be encoded as a decision tree.

# Learning Decision Trees

- Decision trees are learned on $(\boldsymbol{x}, t)$ pairs, where $\boldsymbol{x}$ is a vector of values for the input attributes and $t$ is a single Boolean output value.
- The decision tree learning algorithm adopts a greedy divide and conquer strategy. Always test the most important attribute first. By most important attribute, we mean the one that makes the most difference to the classification of an example.
- This way we hope to get the correct classification with a small number of tests (all paths in the tree should be short and and the tree as whole should be shallow)

# Learning the tree

**if** *Examples is empty* **then**
  return `most_common`(parent_examples)
**end**
**else if** *all examples have the same classification* **then**
  return the classification
**end**
**else if** *attributes is empty* **then**
  return `most_common(examples)`
**end**
**else**
  $A \leftarrow \underset{a \in \text{attributes}}{\text{argmax}}$ `Importance`$(a, \mathit{examples})$
  tree $\leftarrow$ a ne decision tree with root test $A$
  **for** *each value* $v_k$ *of* $A$ **do**
    exs $\leftarrow \{e \ : \ e \in \text{examples and } e.A = v_k\}$
    subtree $\leftarrow$
     `decision_tree_learning(exs, attributes - A, examples)`
    add a branch to `tree` with label $(A = v_k)$ and subtree `subtree`
  **end**
**end**

**Algorithm 1:** Learning a Decision Tree

# Choosing the attributes

- At each step, the construction of the tree considers the following four criteria:
  - If the remaining examples are all positive (or all negative), then we are done. We can answer yes or no.
  - If there are some positive and some negative examples, then choose the best attribute to split them.
  - If there are no examples left, it means that no example has been observed for this combination of attribute values. One approach is to return a guess based on the most common attribute among the parents
  - If there are no attributes left, but both positive and negative examples, it means that these examples have exactly the same description but different classifications. (can arise from errors, noise,..). One approach is then to return the most common decision among the examples as the final decision for the node.

# Choosing the attributes

- The gready learning algorithm just discussed is designed to minimize the depth of the tree.
- When possible, a perfect splitting should divide the tree into two sets, each of which only contains positive or ngative values
- To optimize the tree, we thus need a formal measure of how good a splitting can be.
- One approach is to use the notion of information gain, which can be represented by the notion of entropy, a fundamental quantity in information theory.
- Entropy is a measure of the uncertainty of a random variable. Acquisition of information leads to a reduction of entropy.

# Choosing the attributes

- A random variable with only one value (e.g. a coin that always comes up heads) has no uncertainty and its entropy thus takes the value 0
- That also means that we gain no information by observing its value
- A flip of a fair coin is equally likely to come up heads or tails, 0 or 1, and we will show that this counts as '1bit' of entropy.
- The roll of a fair four sided die has to bits of entropy because it takes two bits to describe one of four equally probable states.

# Choosing the attributes

- Consider an unfair coin that comes up heads 99% of the time. Intuitively that coin has less uncertainty than the fair coin (i.e if we guess heads we will be wrong only 1% of the time) so we would like it to have an entropy measure that is close to 0

- The Entropy of a variable $V$ with values $v_k$ each with probability $P(v_k)$ is defined as

$$\text{Entropy} \ : \ H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k)$$

- We can check that the entropy of a flipped fair coin is indeed 1 bit

$$H(\text{Fair}) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1$$

- If the coin is loaded to give 99% heads, we get

$$H(\text{Loaded}) = -(.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits}$$

# Choosing the attributes

- it will be helpful to use the short notation $B(q)$ to denote the entropy of a Boolean random variable that is true with probability $q$

$$B(q) = -(q \log_2 q + (1-q) \log_2 (1-q))$$

- Let us get back to the decision tree. If a training set contains $p$ positive examples and $n$ negative examples, then the entropy of the goal attribute on the whole set is

$$H(\text{Goal}) = B\left(\frac{p}{p+n}\right)$$

# Choosing the attributes

- An attribute $A$ with $d$ distinct values divides the training set into $E$ into subsets $E_1, \ldots, E_d$
- Assume that we want to split a note which has $p$ positive and $n$ negative examples by selecting a feature $E$ which can take one of $d$ distinct possible values
- Further assume that each of the $E_k$ siubset has $p_k$ positive and $n_k$ negative values.
- We can compute the average entropy after the split has

$$\text{remainder}(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right)$$

# Choosing the attributes

- Intuitively, one can understand the term $B(\frac{p_k}{p_k+n_k})$ as the number of bits of information we will need to answer the initial question if we go along that branch.
- To make this clear, we cna further define an notion of information gain representing the expected reduction in Entropy,

$$\text{Gain}(A) = B\left(\frac{p}{n+p} - \text{Remainder}(A)\right)$$

- Gain is what we will use in our `importance` function.

# Choosing the attributes

- Intuitively, one can understand the term $B(\frac{p_k}{p_k+n_k})$ as the number of bits of information we will need to answer the initial question if we go along that branch.
- To make this clear, we cna further define an notion of information gain representing the expected reduction in Entropy,

$$\text{Gain}(A) = B\left(\frac{p}{n+p} - \text{Remainder}(A)\right)$$

- Gain is what we will use in our `importance` function.

# Choosing the attributes

- A important risk in learning is to excessively rely on the training examples.
- As an illustration of this, consider building a tree that has to predict when a die will return a value of 6.
- The training set consists of examples that encoded though a set of features containing for example the color of the die, the fact that you crossed your fingers when the die was thrown, ...
- If it happens that there are two rolls of a blue die which you threw with your fingers crossed, that return the value 6, the decision tree will conclude that launching a blue die with your fingers crossed consistently returns a 6.
- This problem is known as overfitting.

# Choosing the attributes

- For decision trees, a technique called decision tree pruning can be used to combat overfitting.
- We start from a full tree. We then look for a node that only has leaf nodes as descendants
- If the node is irrelevant in terms of the classification (low information gain or proportion of positive samples equivalent to the total proportion), we eliminate the test (parent + children) and replace it with a leaf node.
- The test is then repeated until each node has either been removed or accepted as is.

# Decision tree pruning

- How large a gain should we consider before pruning a particular attribute?
- One approach is to consider a Statistical significance test
- Let $p$ denote the total number of positive and $n$ denote the total number of negative in the parent node. In the case of a meaningless split, the number of positive and negative samples in the $k^{th}$ child can be estimated approximately as

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n}, \quad \hat{n}_k = n \times \frac{p_k + n_k}{p + n}$$

# Decision tree pruning

- We can then estimate how much our split deviates from a meaningless split, by considering the $\chi^2$ test

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k}$$

  which is used to answer the question: Does the split matches the Null Hypothesis (= random splitting of the examples)?

- Under the null hypothesis, $\Delta$ follows a $\chi^2$ distribution with $v = n + p$ degrees of freedom

- Quantitatively it corresponds to the probability of rejecting the null hypothesis given that this null hypothesis was assumed to be true

# Decision tree pruning

- For example, a significance level of 0.05 indicates a 5% risk of concluding that a difference exists when there is no actual difference.
- E.g. For a $\chi^2$ with 3 degrees of freedom, a value of $\Delta = 11.35$ or more would reject the null hypothesis at a 1% level

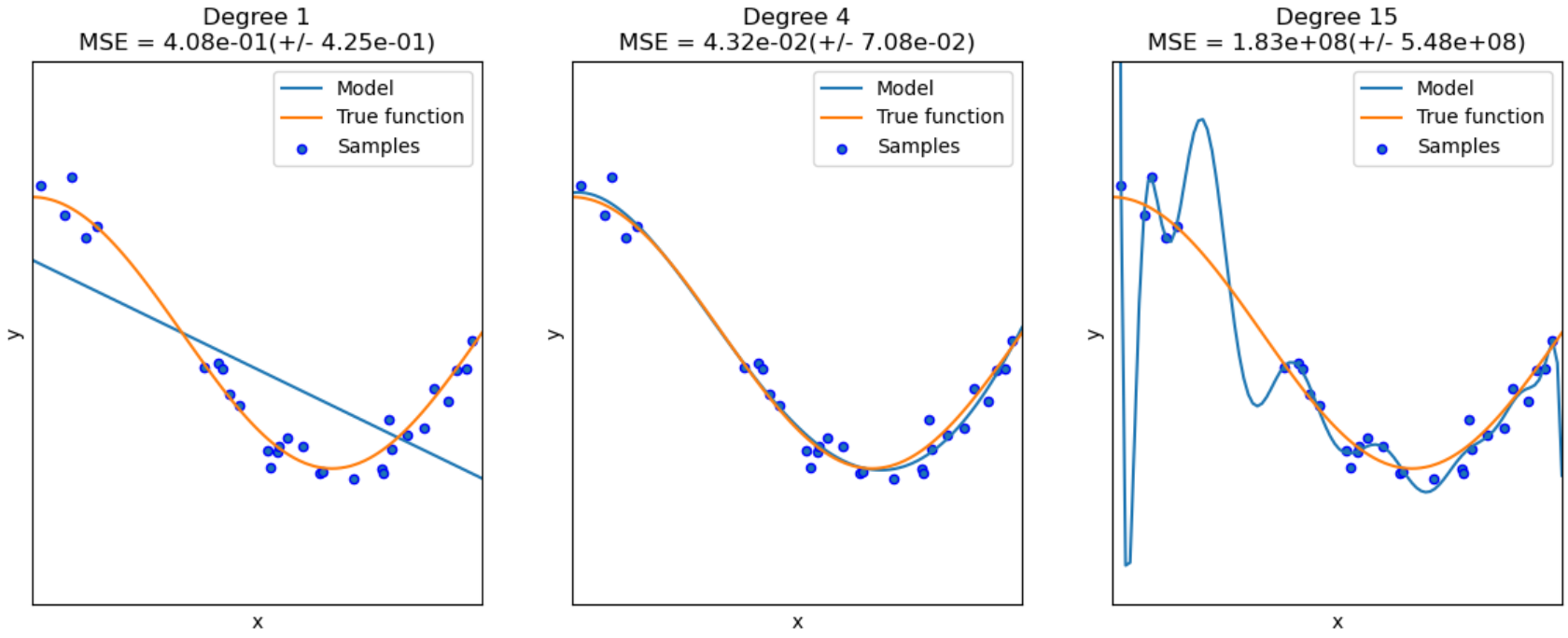
Chi-Square ($\chi^2$) Distribution

| Degrees of Freedom | Area to the Right of Critical Value | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0.99 | 0.975 | 0.95 | 0.90 | 0.10 | 0.05 | 0.025 | 0.01 |
| 1 | — | 0.001 | 0.004 | 0.016 | 2.706 | 3.841 | 5.024 | 6.635 |
| 2 | 0.020 | 0.051 | 0.103 | 0.211 | 4.605 | 5.991 | 7.378 | 9.210 |
| 3 | 0.115 | 0.216 | 0.352 | 0.584 | 6.251 | 7.815 | 9.348 | 11.345 |
| 4 | 0.297 | 0.484 | 0.711 | 1.064 | 7.779 | 9.488 | 11.143 | 13.277 |
| 5 | 0.554 | 0.831 | 1.145 | 1.610 | 9.236 | 11.071 | 12.833 | 15.086 |
| 6 | 0.872 | 1.237 | 1.635 | 2.204 | 10.645 | 12.592 | 14.449 | 16.812 |
| 7 | 1.239 | 1.690 | 2.167 | 2.833 | 12.017 | 14.067 | 16.013 | 18.475 |
| 8 | 1.646 | 2.180 | 2.733 | 3.490 | 13.362 | 15.507 | 17.535 | 20.090 |
| 9 | 2.088 | 2.700 | 3.325 | 4.168 | 14.684 | 16.919 | 19.023 | 21.666 |
| 10 | 2.558 | 3.247 | 3.940 | 4.865 | 15.987 | 18.307 | 20.483 | 23.209 |
| 11 | 3.053 | 3.816 | 4.575 | 5.578 | 17.275 | 19.675 | 21.920 | 24.725 |
| 12 | 3.571 | 4.404 | 5.226 | 6.304 | 18.549 | 21.026 | 23.337 | 26.217 |
| 13 | 4.107 | 5.009 | 5.892 | 7.042 | 19.812 | 22.362 | 24.736 | 27.688 |
| 14 | 4.660 | 5.629 | 6.571 | 7.790 | 21.064 | 23.685 | 26.119 | 29.141 |
| 15 | 5.229 | 6.262 | 7.261 | 8.547 | 22.307 | 24.996 | 27.488 | 30.578 |
| 16 | 5.812 | 6.908 | 7.962 | 9.312 | 23.542 | 26.296 | 28.845 | 32.000 |
| 17 | 6.408 | 7.564 | 8.672 | 10.085 | 24.769 | 27.587 | 30.191 | 33.409 |
| 18 | 7.015 | 8.231 | 9.390 | 10.865 | 25.989 | 28.869 | 31.526 | 34.805 |
| 19 | 7.633 | 8.907 | 10.117 | 11.651 | 27.204 | 30.144 | 32.852 | 36.191 |
| 20 | 8.260 | 9.591 | 10.851 | 12.443 | 28.412 | 31.410 | 34.170 | 37.566 |

# Another example: polynomial regression

- When we learn, we do not know the true mapping $f$ but we approximate it from the hypothesis space $\mathcal{H}$
- As an example we could take this space to be the space of polynomial with bounded degree.
- For any particular dataset $\left\{x^{(i)}, y^{(i)}\right\}_{i=1}^{N}$, polynomials of higher degree will always interpolate the data better than polynomials of low degree.

# Another example: polynomial regression



# Model selection: Complexity vs Goodness of fit

- When the agent will learn a model, we will also want it to learn a hypothesis that we will fit future data well
- A professor knows that an exam will not accurately evaluate the students if they have already seen the exam questions.
- Similarly, to get an accurate evaluation of a hypothesis, we need to test it on a set of examples it has not seen yet
- How then select the optimal degree in the regression problem ?

# Model selection: Complexity vs Goodness of fit

- A simple approach is to randomly split the data into a training set (from which the learning algorithm procduces $h$) and a test set on which the accuracy of $h$ is evaluated
- This approach known as holdout cross validation has the disadvantage that it fails to use all the available data. If we use half the data for the test set, then we are only training the agent on half the data and we may get a poor hypothesis

# Model selection: Complexity vs Goodness of fit

- We can squeeze more out of the data using a technique called $K$-fold cross validation.
- The idea of $K$-fold cross validation is that each sample serves double duty (both training and test).
- First we split the data into $K$ equal bins.
- We then perform $K$ rounds of learning. On each round, $1/k$ of the data is held out as a test set and the remaining examples are used as training data.
- The average test set score should provide a better estimate tha a single score.

# Model selection: Complexity vs Goodness of fit

- Machine learning models can also be represented by means of a utility function
- In machine learning, it is common to encode such a utility by means of a loss function
- The loss function $L(x, y, \hat{y})$ can be interpreted as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $y = f(x)$.

# Model selection: Complexity vs Goodness of fit

- Two popular functions for the loss are the absolute value of the difference (called $L_1$ loss) and the square of the difference (called $L_2$ loss). If what we want is to minimize error rates, a popular choice is the binary $L_{0/1}$ loss function,

$$\text{Absolute value loss : } L_1(y, \hat{y}) = |y - \hat{y}|$$
$$\text{Squared error loss : } L_2(y, \hat{y}) = (y - \hat{y})^2$$
$$0/1 \text{ loss : } L_{0/1}(y, \hat{y}) = 0 \text{ if } y = \hat{y}, \text{ else } 1$$

# Model selection: Complexity vs Goodness of fit

- A learning agent can theoretically maximize its expected utility by choosing the hypothesis that minimizes expected loss over all input-output pairs it will see.

- It does not make sense to talk about the expectation if we do not introduce the prior probability distribution $P(x, y)$ over the examples

- Let $\mathcal{E}$ denote the set of all possible input-output examples. Then the expected generalization loss for a hypothesis $h$ with respect to the loss $L$ is

$$\text{GenLoss}_L(h) = \sum_{(x,y)\in\mathcal{E}} L(y, h(x))P(x, y)$$

and the best hypothesis is the one with the minimum expected generalization loss

$$h^* = \underset{h\in\mathcal{H}}{\text{argmin}}\,\text{GenLoss}_L(h)$$

# Model selection: Complexity vs Goodness of fit

- Because $P(x, y)$ is not known, the learning agent can only estimate the generalization loss from the empirical loss on a particular subset of the examples

$$\text{EmpLoss}_{L,E}(h) = \frac{1}{N} \sum_{(x,y)\in E} L(y, h(x))$$

- The esimated best hypothesis $\hat{h}^*$ is then the one with minimum empirical loss

$$\hat{h}^* = \underset{h\in\mathcal{H}}{\text{argmin}} \text{EmpLoss}_{L,E}(h)$$

# Model selection: Complexity vs Goodness of fit

- There are four reasons why $\hat{h}^*$ may differ from the true model $f$:
  - First $f$ may not be realizable (may not be in $\mathcal{H}$)
  - Second, a learning algorithm will return different hypotheses for differnet sets of examples, even if those are drawn from the same function $f$
  - Third, $f$ may be non deterministic or noisy. It may return different values for $f(x)$ each time $x$ occurs
  - Finally when $\mathcal{H}$ is complex, it may be computationally intractable to systematically search the whole hypothesis space.

# Regularization

- As we saw, one approach at selecting our model is to do cross validation on the model complexity or size

- An alternative approach is to search for a hypothesis that directly minimizes the weighted sum of the empirical loss and the complexity of the hypothesis, which we call the total cost

$$\text{Cost}(h) = \text{EmpLoss}(h) + \lambda \text{Complexity}(h)$$

  The optimal hypothesis then being given by

$$\hat{h}^* = \underset{h \in \mathcal{H}}{\text{argmin}} \text{Cost}(h)$$

- Here $\lambda$ is a parameter, a positive number that serves as a converstion rate between loss and hypothesis complexity

- We however still need to do cross validation in order to determine the optimal value for $\lambda$ (i.e the value that gives the best validation set score)

# Regression and classification

- Let us now move away from trees and consider a different hypothesis space, one that has been used for hundred of years: the class of linear functions of continuous inputs
- A univariate linear function (a straight line) with input $x$ and output $y$ has the form $y = \beta_1 x + \beta_0$ where $\beta_0$ and $\beta_1$ are real coefficients to be learned.
- Let $\boldsymbol{\beta} = [\beta_1, \beta_0]$. The task of finding the hypothesis $h_{\boldsymbol{\beta}}$ that best fits the data is called linear regression
- To fit a line to the data, all we have to do is find the values of the weights $(\beta_1, \beta_0)$ that minimize the empirical loss

# Regression and classification

- It is traditional to use the squared loss function, $L_2$ summed over all training samples

$$\begin{aligned}\text{Loss}(h_{\boldsymbol{\beta}}) &= \sum_{j=1}^{N} L_2(y_j, h_{\boldsymbol{\beta}}(x_j)) \\ &= \sum_{j=1}^{N} (y_j - h_{\boldsymbol{\beta}}(x_j))^2 \\ &= \sum_{j=1}^{N} (y_j - (\beta_1 x_j + \beta_0))^2\end{aligned}$$

- We woud like to find the $\beta^* = \underset{\boldsymbol{\beta}}{\text{argmin}} \text{Loss}(h_{\boldsymbol{\beta}})$

# Regression and classification

- the sum

$$\text{Loss}(h_\beta) = \sum_{j=1}^{N} (y_j - (\beta_1 x_j + \beta_0))^2$$

is minimized when its partial derivatives with respect to $\beta_1$ and $\beta_0$ are zero

$$\frac{\partial}{\partial \beta_0} \sum_{j=1}^{N} (y_j - (\beta_1 x_j + \beta_0))^2 = 0$$

$$\frac{\partial}{\partial \beta_1} \sum_{j=1}^{N} (y_j - (\beta_1 x_j + \beta_0))^2 = 0$$

Those equations have a unique solution

# Regression and classification

- For univariate linear regression, the weight space (the space of all possible settings of the weights) is two dimensional and convex which implies that there are no local minimas.
- When consider more complex models (such as neural networks), finding the optimal value for the weights will involve a general optimization search problem in a continuous weight space
- Such problems can be addressed by a hill climbing algorithm that follows the gradient of the function to be optimized
- In this case, because we are trying to minimize the loss, we use gradient descent

# Gradient descent

$\boldsymbol{\beta} \leftarrow$ any point in the weight space
**while** `num_iter` $<$ `max_iter` **do**
  **for** *each* $\beta_i$ *in* $\boldsymbol{\beta}$ **do**
    $\beta_i \leftarrow \beta_i - \eta \frac{\partial}{\partial \beta_i} \text{Loss}(\boldsymbol{\beta})$
    num_iter $+ = 1$
  **end**
**end**

**Algorithm 2:** gradient descent

- The parameter $\eta$ is called the learning rate or step size
- It can be a constant or it can decay over time as the learning process proceeds

# Gradient descent

- in the univariate case, the partial derivatives are given by

$$\begin{aligned}\frac{\partial L}{\partial \beta_i}\text{Loss}(\boldsymbol{\beta}) &= \frac{\partial (y - h_{\boldsymbol{\beta}}(x))^2}{\partial \beta_i} \\ &= 2(y - h_{\boldsymbol{\beta}}(x))\frac{\partial}{\partial \beta_i}(y - h_{\boldsymbol{\beta}}(x)) \\ &= 2(y - h_{\boldsymbol{\beta}}(x))\frac{\partial}{\partial \beta_i}(y - (\beta_1 x + \beta_0))\end{aligned}$$

From which we get

$$\begin{aligned}\frac{\partial}{\partial \beta_0}\text{Loss}(\boldsymbol{\beta}) &= -2(y - h_{\boldsymbol{\beta}}(x)), \\ \frac{\partial}{\partial \beta_1}\text{Loss}(\boldsymbol{\beta}) &= -2(y - h_{\boldsymbol{\beta}}(x))x\end{aligned}$$

# Gradient descent

- The gradient iterations thus yield

$$\beta_0 \leftarrow \beta_0 + \eta \sum_{j=1}^{N} (y^{(j)} - h_{\boldsymbol{\beta}}(x^{(j)})),$$

$$\beta_1 \leftarrow \beta_1 + \eta \sum_{j=1}^{N} (y^{(j)} - h_{\boldsymbol{\beta}}(x^{(j)}))x^{(j)}$$

- Those iterates are referred to as Batch gradient descent
- An alternative consists in considering a single training example $\left\{\boldsymbol{x}^{(i)}, y^{(i)}\right\}$ at each iteration, cycling over the whole set of examples. This second approach is known as Stochastic gradient descent.

# Multivariate linear regression

- We can easily extend the previous discussion to the multivariate framework
- In this case, the hypothesis $h_{\boldsymbol{\beta}}$ for an example $\boldsymbol{x}^{(i)}$ reads as

$$h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}) = \beta_0 + \beta_1 x_1^{(i)} + \ldots + \beta_n x_n^{(i)} = \beta_0 + \sum_{j=1}^{D} \beta_j x_j^{(i)}$$

- The $\beta_0$ term is often called the intercept or the bias.
- If we introduce the weight vector $\tilde{\boldsymbol{\beta}} = [\beta_0, \ldots, \beta_D]$, and write the input vector as $\tilde{\boldsymbol{x}}^{(i)} = [1, x_1^{(i)}, \ldots, x_D^{(i)}]$, then we can simply write the model as the dot product of the weight vector and the input vector $h_{\boldsymbol{\beta}} = \tilde{\boldsymbol{\beta}}^T \tilde{\boldsymbol{x}}$

# Multivariate linear regression

- As in the univariate framework, we can find the vector of weights by minimizing the sum of squared errors

$$\boldsymbol{\beta}^* = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{N} \ell_2(y^{(i)}, \tilde{\boldsymbol{\beta}}^T \tilde{\boldsymbol{x}}^{(i)})$$

- The corresponding gradient descent iterates are then defined as

$$\beta_j \leftarrow \beta_j + \eta \sum_{i=1}^{N} x_j^{(i)} (y^{(i)} - h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}))$$

- The solution $\beta^*$ can also be found by setting the derivatives to 0, as

$$\tilde{\boldsymbol{\beta}}^* = (\tilde{\boldsymbol{X}}^T \tilde{\boldsymbol{X}})^{-1} \tilde{\boldsymbol{X}}^T \boldsymbol{y}$$

Where $\tilde{\boldsymbol{X}}$ is the data matrix (having the examples $\boldsymbol{x}^{(i)}$ as rows.)

# Multivariate linear regression

- When using multivariate linear, it is possible that some dimension that is actually irrelevant, appears by chance to be useful, resulting in a phenomenon known as overfitting
- A common approach to mitigate that risk is to rely on regularization
- The total cost of an hypothesis $h_{\boldsymbol{\beta}}$ being then given by

$$\text{Cost}(h_{\boldsymbol{\beta}}) = \text{EmpLoss}(h_{\boldsymbol{\beta}}) + \lambda \text{Complexity}(h_{\boldsymbol{\beta}})$$

- When the hypothesis is linear, such as in multilinear regression, the complexity can be specified as a function of the weights

$$\text{Complexity}(h_{\boldsymbol{\beta}}) = \ell_2(\boldsymbol{\beta}) = \sum_{j=1}^{D} |\beta_j|^2$$

# Multivariate linear regression

- Other possible regularizers include the $\ell_1$ norm,

$$\text{Complexity}(h_{\boldsymbol{\beta}}) = \ell_1(\boldsymbol{\beta}) = \sum_{j=1}^{D} |\beta_j|$$

- $\ell_1$ will usually favor sparser solutions (i.e solutions with more weights set to 0).

- $\ell_1$ can be seen to favor the axes while $\ell_2$ treats every direction similarly

# Multivariate linear regression

- This can be explained intuitively from the fact that for a generic loss, the corners of the $\ell_1$ ball will usually be located closer to the minimum of the loss

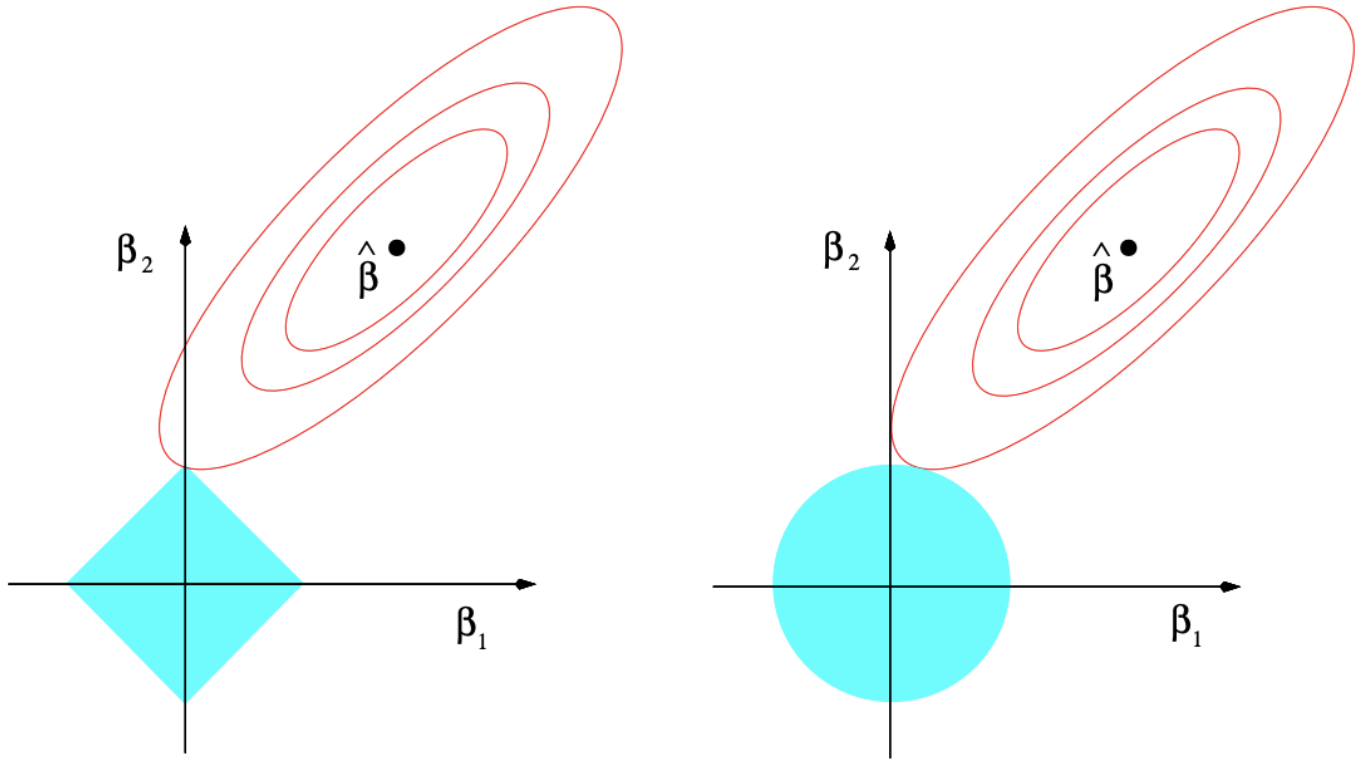


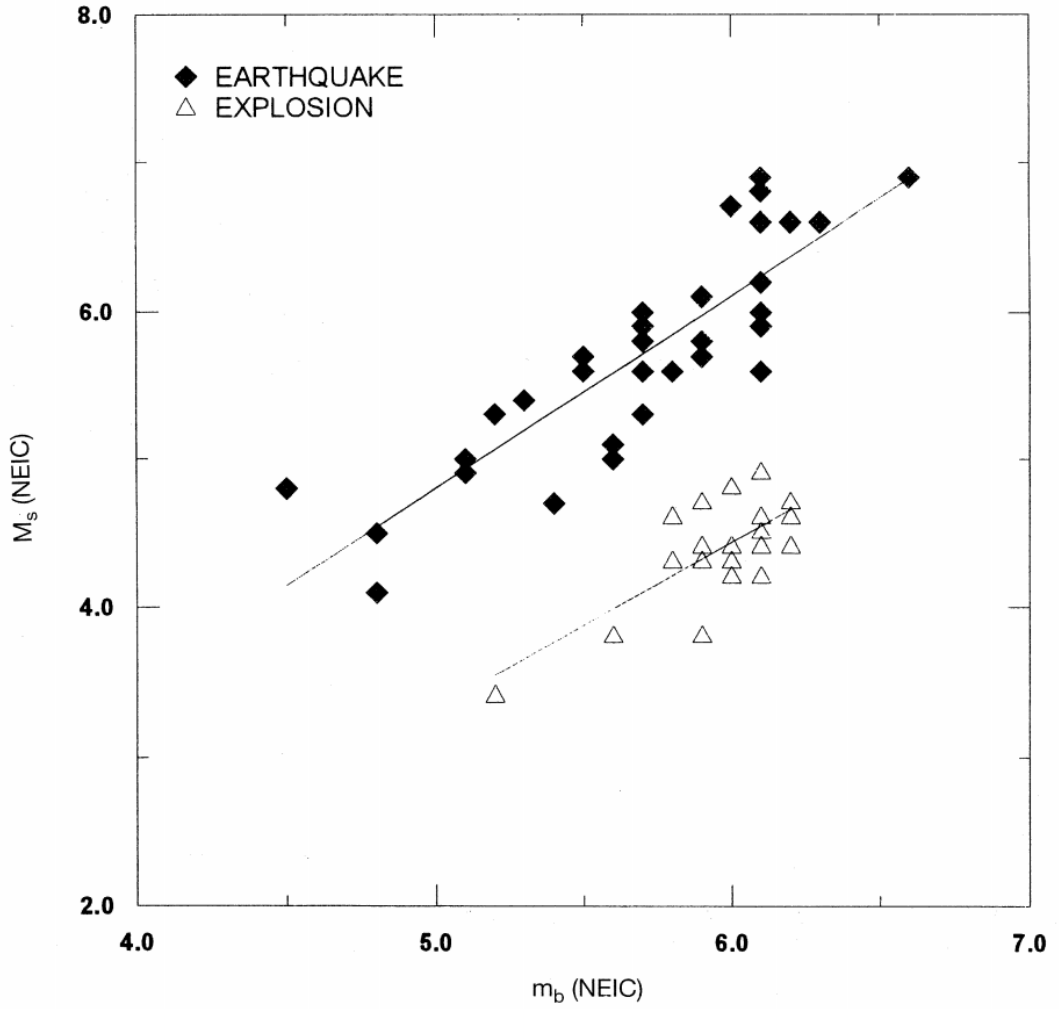

**FIGURE 3.11.** *Estimation picture for the lasso (left) and ridge regression (right). Shown are contours of the error and constraint functions. The solid blue areas are the constraint regions* $|\beta_1| + |\beta_2| \leq t$ *and* $\beta_1^2 + \beta_2^2 \leq t^2$*, respectively, while the red ellipses are the contours of the least squares error function.*

# Classification w/ hard threshold

- Consider for example a dataset that would consists of two classes: earthquakes (which are of interest to seismologist) and underground explosions (which are of interest to arms control experts). Each point is defined by two values $x_1$ and $x_2$ that refer to body and surface wave magnitudes.
- Given this training data, we want to learn a hypothesis $h_\beta$ that will take a new measurement $(x_1, x_2)$, and return either 0 for eartquakes and 1 for explosions.
- We call decision boundary the line that separates the two classes
- A dataset that admits a linear decision boundary is called linearly separable.

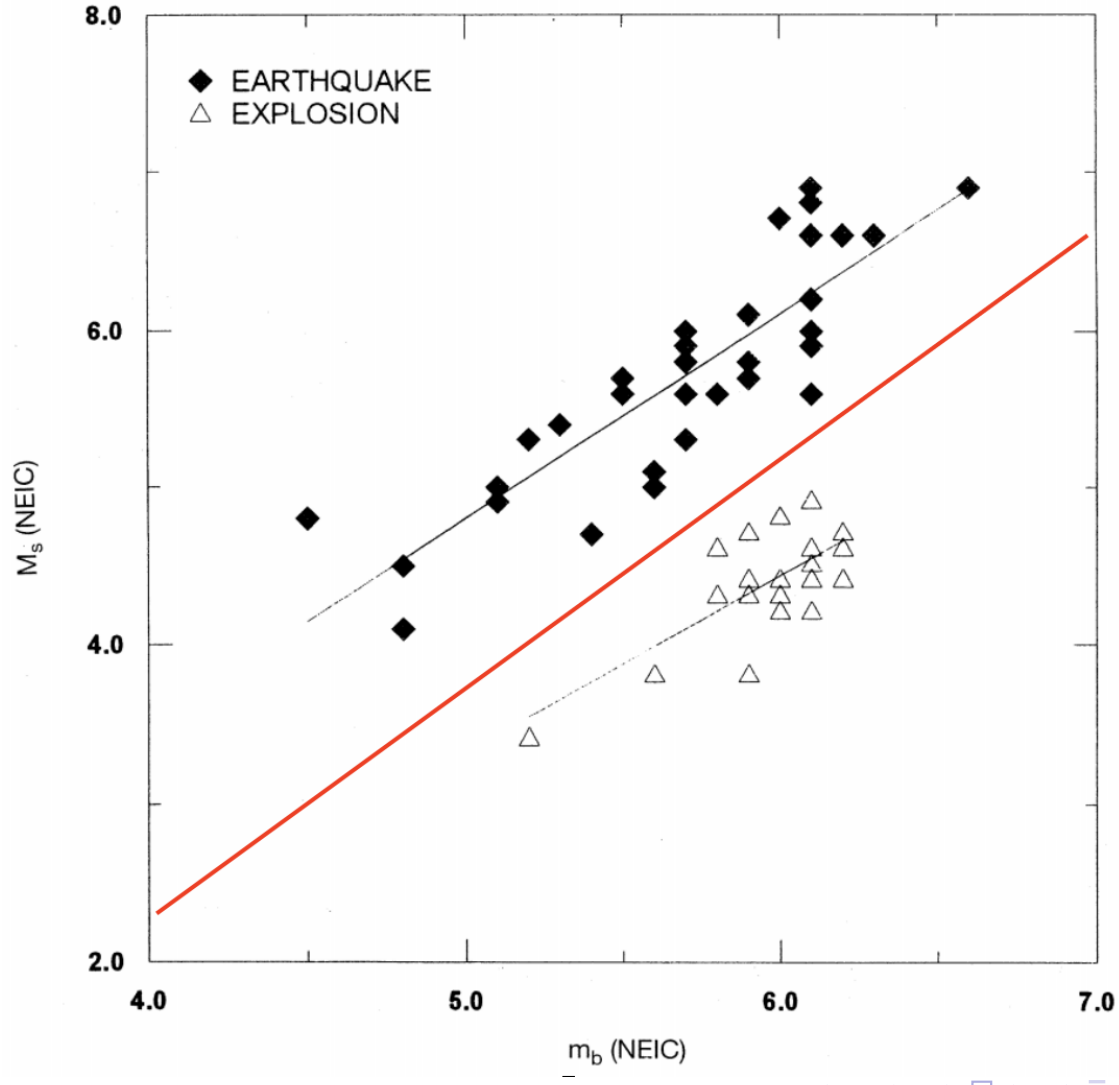
# Classification w/ hard threshold

- Linear function can be used to do classification as well as regression



# Classification w/ hard threshold

- Linear function can be used to do classification as well as regression



# Classification w/ hard threshold

- In the case of the seismic dataset, the linear separator is given by $x_2 = 1.7x_1 - 4.9$ or $-4.9 + 1.7x_1 - x_2 = 0$
- The explosions that we want to classify with value 1 are those on the right of the plane, i.e. they correspond to the pairs $(x_1, x_2)$ for which $-4.9 + 1.7x_1 - x_2 > 0$
- While earthquakes correspond to the points $-4.9 + 1.7x_1 - x_2 < 0$.
- From this, we can thus define the classification hypothesis as

$$h_\beta(\boldsymbol{x}) = 1 \text{ if } \boldsymbol{\beta}^T \boldsymbol{x} \geq 0, \quad \text{and 0 otherwise}$$

# Classification w/ hard threshold

- Alternatively, we can think of our model $h_{\boldsymbol{\beta}}$ as the result of passing a linear function $\tilde{\boldsymbol{\beta}}^T \tilde{\boldsymbol{x}}$ through a threshold function

$$h_{\boldsymbol{\beta}}(\boldsymbol{x}) = \text{Threshold}(\tilde{\boldsymbol{\beta}}^T \tilde{\boldsymbol{x}}),$$
$$\text{where Threshold}(z) = 1, \text{ if } z \geq 0, \text{ and 0 otherwise}$$

- The issue with this model is that the gradient is now 0 almost everywhere except at those points where $\tilde{\boldsymbol{\beta}}^T \tilde{\boldsymbol{x}} = 0$.

- There is however a simple weight update rule that converges to a solution. For a single example $(\boldsymbol{x}^{(i)}, y^{(i)})$, we can indeed take

$$\beta_j \leftarrow \beta_j + \eta(y^{(i)} - h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}))x_j^{(i)}$$

This rule is known as the percepton learning rule

# Classification w/ hard threshold

- The perceptron learning rule works as follows:
  - If the example $(\boldsymbol{x}^{(i)}, y^{(i)})$ is correctly classified by the model, i.e. if $y^{(i)} = h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)})$
  - If $y^{(i)}$ is 1 but $h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)})$ is 0, the weight $\beta_j$ is increase when the corresponding input $x_j^{(i)}$ is positive and decreased if this input is negative. This makes sense as we want to make $h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)})$ bigger so that it outputs a 1 instead of a 0
  - If $y^{(i)}$ is 0 but $h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)}) = 1$, then $\beta_j$ is decreased if $x_j^{(i)}$ is positive and increased when $x_j^{(i)}$ is negative. This again makes sense as we want to decrease the value of $h_{\boldsymbol{\beta}}(\boldsymbol{x}^{(i)})$ to zero.
- The Perceptron learning rule is guaranteed to find a separating boundary provided that the data is linearly separable