UNIVERSITÄT BERN

Machine Learning Review

Paolo Favaro

A prerequisite for this course is ML (and CV)

Contents

- Revision of basic concepts of Machine Learning
- Based on **Chapter 5** of Deep Learning by Goodfellow, Bengio, Courville

Context

- A more complete introduction to Machine Learning through the following courses
  - Machine Learning @ UniBe
  - Machine Learning and Data Mining @ UniNe
  - Pattern Recognition @ UniFr
  - Statistical Learning Methods @ UniNe

Resources

- Books and online material for further studies
  - Machine Learning @ Stanford (Andrew Ng)
  - **Pattern Recognition and Machine Learning**
    by Christopher M. Bishop
  - **Machine Learning: a Probabilistic Perspective**
    by Kevin P. Murphy

Learning Pillars

- Supervised learning
- Semi-supervised learning
- Self-taught learning (unsupervised feature learning)
- Unsupervised learning
- Reinforcement learning

Machine learning can be divided into three (or more) categories.
By far the category most commonly used is supervised learning.
It is directly applicable to many useful problems in the industry and in science.
Unsupervised and reinforcement learning are the areas with the most potential
and gain, as they scale well (no need for costly annotation)
Variations are semi-supervised learning (labeled data + unlabelled data restricted to the known categories)
and self-taught learning (labeled data + unlabelled data with any category)

Definition

- Mitchell (1997)
  A computer program is said to learn from ***experience E*** *with respect to some class of* ***tasks T*** *and* ***performance measure P****, if its performance at tasks in T, as measured by P, improves with experience E.*

The Task T

- Example: if we want a robot to be able to walk, then **walking** is the task
- Approaches
  1. We could directly input directives for how we think a robot should walk, or
  2. We could provide examples of successful and unsuccessful walking (this is machine learning)

Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings.

The Task T

- Given an input x (e.g., a vector) produce a function f, such that f(x) = y (e.g., an integer, a probability vector)
- Examples
  - Classification
  - Regression
  - Machine translation
  - Denoising
  - Probability density estimation

Classification: predict which of k categories some input belongs to. The learning algorithm is usually asked to produce a function f : R^n -> {1, . . . , k}. There are other variants of the classification task, for example, where f outputs a probability distribution p(y|x) over classes, where x is the input and y the class. An example of a classification task is object recognition, where the input is an image, and the output is a numeric code identifying the object in the image. Modern object recognition is best accomplished with deep learning (Krizhevsky et al., 2012; Ioffe and Szegedy, 2015). Object recognition is the same basic technology that allows computers to recognize faces (Taigman et al., 2014), which can be used to automatically tag people in photo collections and allow computers to interact more naturally with their users.

Regression: predict a numerical value given some input. The learning algorithm outputs a function, e.g., f : R^n -> R. This type of task is similar to classification, except that the output is continuous

Machine translation: the input consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. Example: translating from English to French

Denoising: the algorithm is given in input a corrupted example z obtained by an unknown corruption process from a clean example x. The learner must predict the clean example x from its corrupted version z, or more generally predict the conditional probability distribution p(x|z).

Density estimation or probability mass function estimation: the algorithm is asked to learn a function p : R^n -> R, where p(x) is a probability density function (if x is continuous) or a probability mass function (if x is discrete). The algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. In principle, we can then perform computations on that distribution in order to solve the other tasks as well. In practice, density estimation does not always allow us to solve all of these related tasks, because in many cases the required operations on p(x) are computationally intractable.

The Performance Measure P

- To evaluate a ML algorithm we need a way to measure how well it performs on the task
- It is measured on a separate set (**the test set**) from what we use to build the function f (**the training set**)
- Examples
  - Classification accuracy (portion of correct answers) or error rate (portion of incorrect answers)
  - Regression accuracy (e.g., least squares errors)

Error rate is typically the average 01 loss
For probability density estimation tasks we typically evaluate the average likelihood on a set of examples.

Choosing a good performance measure is non trivial and essential for the correct design of a machine learning algorithm.

The Experience E

- Specifies what data can be used to solve the task
- We can distinguish it based on the learning pillars
  - **Supervised**: data is composed of both the input x (e.g., features) and output y (e.g., labels/targets)
  - **Unsupervised**: data is composed of just x; here we typically aim for p(x) or a method to sample p(x)
  - **Reinforcement**: data is dynamically gathered based on previous experience

Please see Sutton and Barto (1998) or Bertsekas and Tsitsiklis (1996) for information about reinforcement learning, and Mnih et al. (2013) for the deep learning approach to reinforcement learning.

Relations

Supervised to unsupervised

- In unsupervised learning one learns p(x)

$$p(x) = \prod_{i=1}^{n} p(x_i|x_{i-1}, \ldots, x_1) \qquad x = \begin{bmatrix} x_1 \\ \ldots \\ x_n \end{bmatrix}$$

- Each conditional probability is a supervised learning problem (except the last one $p(x_1)$)

This strategy also connects to the newly developed method called self-supervised learning. In self-supervised learning one tries also to identify the conditional probability choice that then leads to the “best” representation (best in the sense that this should transfer well to other tasks)

Relations

Unsupervised to supervised

- Alternatively, given input x and output y one can use unsupervised learning to obtain

$$p(x, y)$$

and then solve the supervised learning task via

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

Data

- We assume that all collected data samples in all datasets:

  1. come from the same distribution → $p_{x^{(i)}}(x) = p_{x^{(j)}}(x)$

  2. are independent → $p\left(x^{(1)}, \ldots, x^{(m)}\right) = \prod_{i=1}^{m} p\left(x^{(i)}\right)$

- This assumption is denoted **IID** (independent and identically distributed)

whether we use supervised, unsupervised or reinforcement learning, we rely on collected samples. Some basic assumptions need to be made.

The Learning Problem

- **The learning problem**: Given a finite set of samples, find a set of rules that are probably correct about most data
- **Hume (1739–1740)**: Even after the observation of the frequent or constant conjunction of objects, we have no reason to draw any inference concerning any object beyond those of which we have had experience

The philosopher Hume is basically saying that we cannot learn anything about the data distribution given the data samples available to us
Does this mean that the machine learning problem is unsolvable?
This concept has been made formal by Wolpert (see next slide)

No Free-Lunch Theorem (Wolpert 1996)

- **Theorem**: Averaged over all possible data generating distributions, every set of rules has the same error rate when used on previously unobserved points
- **However, optimal algorithms (set of rules) exist for specific data distributions**

The theorem means that there is no universally good algorithm on any type of data (distribution).
However, we can look for algorithms for specific data distributions (and tasks).

Example: Linear Regression

- Given IID data inputs $x \in \mathbb{R}^n$ and outputs $y \in \mathbb{R}$
- **Task T**: predict y with the linear regressor $\hat{y} = w^\top x$
  need to find the weights w
- **Experience E**: training set $X^{\text{train}} \in \mathbb{R}^{m \times n}$, $Y^{\text{train}} \in \mathbb{R}^m$
  and test set $X^{\text{test}} \in \mathbb{R}^{m \times n}$, $Y^{\text{test}} \in \mathbb{R}^m$
- **Performance P**: Mean squared error

$$\text{MSE}^{\text{test}}(w) = \frac{1}{m} |X^{\text{test}} w - Y^{\text{test}}|^2$$

We choose test and training sets of the same size for simplicity. They usually differ.
The performance is computed on the test set. We want to do well on previously unseen data points. Overall, we want to minimize Bayes risk (that is the error over all data points weighted against the probability of the data points). Since we only have samples, we try to approximate the probability of data points with the test set.

Linear Regression

- Solve task T by minimizing the $\text{MSE}^{\text{train}}$

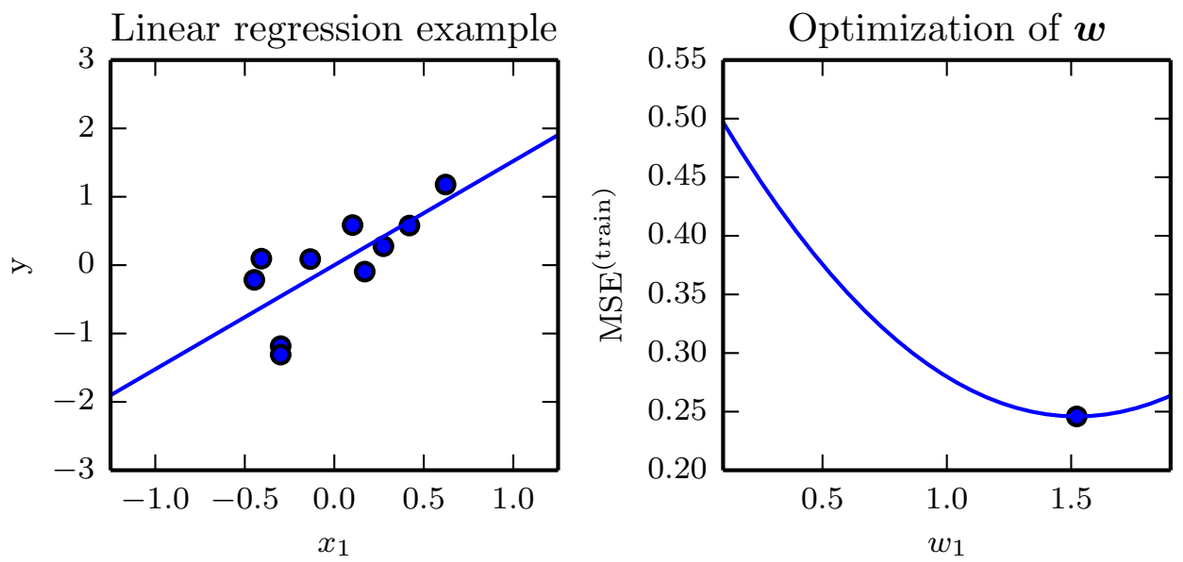
$$\text{MSE}^{\text{train}}(w) = \frac{1}{m}|X^{\text{train}}w - Y^{\text{train}}|^2$$

- Compute the gradient of $\text{MSE}^{\text{train}}$(w) with respect to w and set to 0 (normal equations)

- The solution is (pseudo-inverse)

$$w = \left(X^{\text{train}\top}X^{\text{train}}\right)^{-1} X^{\text{train}\top}Y^{\text{train}}$$

Linear Regression



The plot on the left shows the training set samples (1 dimensional x) and the blue line is the one satisfying y = w^T x after estimating w by minimizing MSEtrain.
The plot on the right shows the values of MSEtrain for several w values (w is also 1 dimensional — it must match the dimension of x). It is a quadratic function and has a unique minimum.

Overfitting and Underfitting

- Performance P captures how well the learned model predicts new unseen data
- Ideally we want to select the predictor with the best performance
- What happens when we use predictors of different complexity/capacity?

Validation and training sets introduce a regularization methodology.
We use both to estimate the optimal parameters, but we use them
in different ways. The training set is used to influence predictions in
a detailed and local way. The validation set is used to influence
predictions in a more global way.
The test set should be used only once and then never used again.
Otherwise it becomes a validation set.

We are interested in minimizing the
Bayes Risk = E_x,y [loss(f(x,w),y)]
with respect to the parameters w of the predictor f.
Since we only have m samples, we can use the empirical error
w* = arg min_w 1/m \sum_x,y loss(f(x,w),y)
However, this problem formulation tends to approximate
poorly the Bayes Risk. We know that the gap
between the Bayes Risk and the empirical error is bounded
by a factor that depends on the VC dimension of the predictor and
the dataset size. In principle, one would want to use as much data
as possible and instead limit the complexity of the predictor as much
as possible. In practice, however, people do not use all the data for training,

but save some of it (the validation set) for the selection of the predictor complexity/capacity.

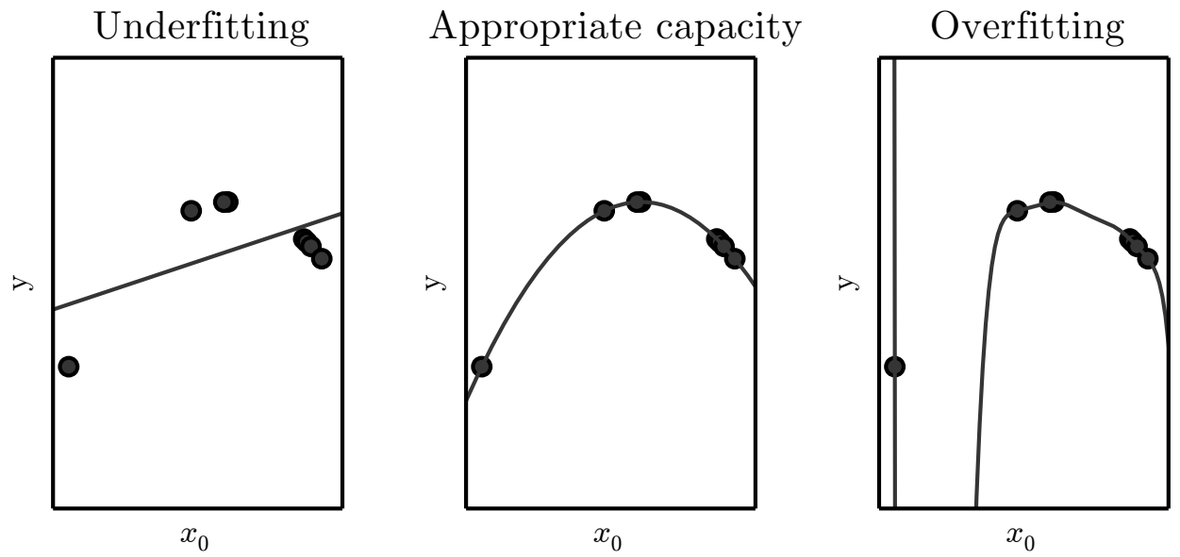
Instead, we compute
w~ = argmin_w 1/m2 \sum_x2,y2 [loss(f(x2,w), y2)] % training
and
performance = 1/m1 \sum_x1,y1 [loss(f(x1,w~), y1)] % testing

we need to solve
w* = argmin_w E_x,y [loss(f(x,w),y)]
  = argmin_w E_x1,y1|x2,y2 [E_x2,y2 [loss(f(x,w),y)]]
  = argmin_w E_x1,y1|x2,y2 [E_x2,y2 [loss(f(x,w),y)]]

Strictly speaking the use of a test set to evaluate the performance is appropriate
only if it is used once at the end. Any reiteration based on this feedback would
defeat the whole purpose of a test set.
If feedback is needed, then one should use a validation set. The validation set
influences the predictions more loosely (global choices rather than local/specific/detailed ones) than the training set. By doing so it avoids overfitting.

Overfitting and Underfitting

shown data is the training set



simple predictor | optimal predictor | complex predictor

Loss function

- Define a **predictor** function $f : \mathcal{X} \mapsto \mathcal{Y}$
- Define a **loss** function $l : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$ which measures how different the two inputs are
- Examples
  - 0-1 loss $$l(y, f(x)) = \begin{cases} 0 \text{ if } y = f(x) \\ 1 \text{ if } y \neq f(x) \end{cases}$$
  - Quadratic loss $$l(y, f(x)) = (y - f(x))^2$$

Bayes Risk

- **Bayes risk** is defined as (average loss)

$$R(f) = E_{x,y}[l(f(x), y)] = \int l(f(x), y)p(x, y)dxdy$$

- The optimal predictor function is

$$f^* = \arg\min_f R(f)$$

We cannot compute Bayes risk because we do not have the joint probability of input x and output y.

Empirical Risk

- Given (x_i,y_i) with i = 1,…,m the **empirical risk** is

$$\hat{R}(f) = \frac{1}{m} \sum_{i=1}^{m} l(f(x_i), y_i)$$

- The empirical predictor is

$$\hat{f} = \arg\min_{f \in \mathcal{F}} \hat{R}(f)$$

We can at most compute the empirical risk, which is a function of a finite set of samples.

Risks

- Bayes risk $R(f^*) = E_{x,y}[l(f^*(x), y)]$
- Empirical risk $\hat{R}(\hat{f}) = \frac{1}{m}\sum_{i=1}^{m} l(\hat{f}(x_i), y_i)$
- Bayes risk restricted to function family $\min_{f \in \mathcal{F}} R(f)$

So far we can consider the following set of risks: the ideal Bayes risk, the practical empirical risk and the Bayes risk restricted to a function family.

There are two main ways to choose the predictor function f: one is to use choose the set of samples and the other is to restrict f to a family F.

The risks above will allow us to calculate the error with respect to Bayes risk when playing with the samples dataset or F.

Estimation vs Approximation

- The **excess risk** is the gap between the empirical risk and the optimal Bayes risk

$$\hat{R}(\hat{f}) - R(f^*) = \underbrace{\hat{R}(\hat{f}) - \min_{f \in \mathcal{F}} R(f)}_{\text{estimation error}} + \underbrace{\min_{f \in \mathcal{F}} R(f) - R(f^*)}_{\text{approximation error}}$$

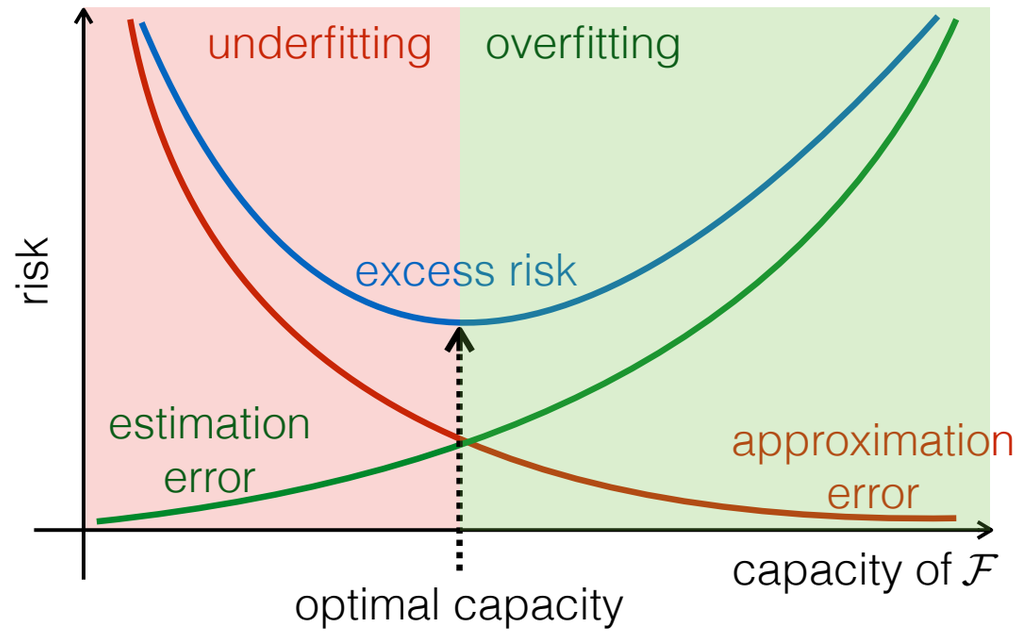
- **Estimation (variance)**: due to training set
- **Approximation (bias)**: due to function family $\mathcal{F}$

The estimation error quantifies how well we can use the training data to identify the predictor function (variance). The two risks in this term use the same family of functions, but one uses the training set and the other uses all the data.

The approximation error quantifies the error due to the choice of predictor function family (squared bias).
The two risks in this term use the same data distribution but the first one restricts f to a family and the other does not.

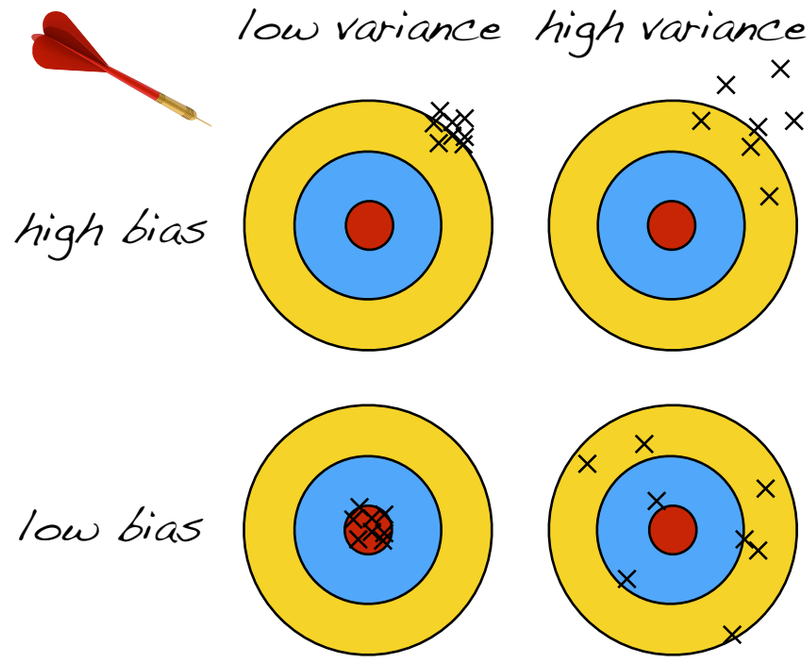
Extreme cases: if the prediction function family is extremely large, then approximation error is very low but the estimation error is high. If the prediction family is just one function then there is nothing to learn from data and the estimation error is very low (but the approximation error is very high).

Estimation vs Approximation



If the family F is too large, we might incur in overfitting, while if F is too small, we might incur in underfitting.

Bias and Variance



Concept by Pedro Domingos
University of Washington

Another visual example of bias and variance.
In this example we show darts throws.
bias is the learner tendency to learn always the wrong thing (the throwing strategy in the example).
variance is the tendency to learn random things with different observed data.

Regularization

- Define a parametric family $\mathcal{F}_\lambda$ of functions, where λ regulates the complexity/capacity of the predictors
- Given the optimal predictor from the empirical risk

$$\hat{f}_\lambda = \arg\min_{f \in \mathcal{F}_\lambda} \hat{R}(f)$$

we would like to choose the capacity based on Bayes risk

$$R(\hat{f}_\lambda)$$

Regularization

- Bayes risk is not available, thus we write

$$R(\hat{f}_\lambda) = \hat{R}(\hat{f}_\lambda) + \left(R(\hat{f}_\lambda) - \hat{R}(\hat{f}_\lambda) \right)$$

and approximate the second term with a regularization term

$$C(\hat{f}_\lambda) \simeq R(\hat{f}_\lambda) - \hat{R}(\hat{f}_\lambda)$$

then solve $\hat{\lambda} = \arg\min_\lambda \hat{R}(\hat{f}_\lambda) + C(\hat{f}_\lambda)$

The regularization function C is designed to measure the complexity of the function

Training, Validation and Test

- In alternative, collect samples into training set D_{train} validation set D_{val} and test set D_{test}
- Use the **training set** to define the optimal predictor

$$\hat{f}_\lambda = \arg\min_{f \in \mathcal{F}} \hat{R}_{D_{\text{train}}}(f)$$

- Use the **validation set** to choose the capacity

$$\hat{\lambda} = \arg\min_{\lambda} \hat{R}_{D_{\text{val}}}(\hat{f}_\lambda)$$

- Use the **test set** to evaluate the performance

$$\text{performance P} = R_{D_{\text{test}}}\left(\hat{f}_{\hat{\lambda}}\right)$$

This is the typical way of using these three sets.
One can also combine this method with the regularization term C.
Other variants are the cross-validation methods and in particular the leave one out method. In these cases there are several training and validation sets (all partitions of the same set of samples) and the evaluated risk is averaged over all choices.

Maximum Likelihood

- Given IID data samples $x^1, \ldots, x^m \sim p_{\text{data}}(x)$
- Let $p_{\text{model}}(x; \theta)$ be a parametric family of probability density functions
- The **maximum likelihood estimator** of θ is

$$\begin{aligned}\theta_{\text{ML}} &= \arg\max_\theta \prod_{i=1}^m p_{\text{model}}(x^i; \theta) \\ &= \arg\max_\theta \sum_{i=1}^m \log p_{\text{model}}(x^i; \theta) \\ &= \arg\max_\theta E_{x \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(x; \theta)]\end{aligned}$$

with the empirical data distribution $\hat{p}_{\text{data}}(x)$

Now the first step is to solve the optimization problem where we recover the parameters on the training set. One technique to do so is Maximum Likelihood. The empirical distribution \hat p_data is the sum of Dirac deltas centered at each sample.

Maximum Likelihood

- Maximum likelihood estimation can also be interpreted as fitting $p_{\text{model}}(x;\theta)$ to $p_{\text{data}}(x)$ via a Kullback-Leibler divergence minimization

$$
\begin{aligned}
&\arg\max_{\theta} -D_{\text{KL}}(\hat{p}_{\text{data}}|p_{\text{model}}) \\
=&\arg\max_{\theta} -E_{x\sim\hat{p}_{\text{data}}}[\log\hat{p}_{\text{data}}(x) - \log p_{\text{model}}(x;\theta)] \\
=&\arg\max_{\theta} E_{x\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(x;\theta)]
\end{aligned}
$$

Maximum Likelihood

- Given IID input/output samples $(x^i, y^i) \sim p_{\text{data}}(x, y)$

  the conditional maximum likelihood estimate is

$$\theta_{\text{ML}} = \arg\max_{\theta} \prod_{i=1}^{m} p_{\text{data}}(y^i | x^i; \theta)$$

$$= \arg\max_{\theta} \sum_{i=1}^{m} \log p_{\text{data}}(y^i | x^i; \theta)$$

The maximum likelihood estimator of a Gaussian model distribution p(y|x) yields the same previous linear regression (least squares solution)

Maximum a Posteriori

- Given IID data samples $x^1, \ldots, x^m \sim p_{\text{data}}(x)$
- Let $p(x^1, \ldots, x^m|\theta)$ be the conditional probability density function
- Maximum a Posteriori aims at recovering

$$p(\theta|x^1, \ldots, x^m) = \frac{p(x^1, \ldots, x^m|\theta)p(\theta)}{p(x^1, \ldots, x^m)}$$

Now we see the parameters \theta as a random variable

Maximum a Posteriori

- The prior $p(\theta)$ can encode a preference for simpler or smoother models (acts as regularizer)
- Predictions could be obtained by marginalizing over the parameters (this corresponds to a quadratic loss function in Bayes risk)

$$p(x^{m+1}|x^1,\ldots,x^m) = \int p(x^{m+1}|\theta)p(\theta|x^1,\ldots,x^m)d\theta$$

- However, this quickly becomes computationally unfeasible with large datasets

The prior p(\theta) controls models complexity.
The marginalization helps avoid overfitting.
The prediction is made by first fitting \theta to the x^1 to x^m values.
Then, given \theta there is no extra information in the x^1 to x^m
so we can write $p(x^{m+1}|\theta,x^1,\ldots,x^m) = p(x^{m+1}|\theta)$

Maximum a Posteriori

- If we minimize Bayes risk with a 0-1 loss function, we obtain a point estimate

$$\theta_{\mathrm{MAP}} = \arg\max_{\theta} p(\theta|x) = \arg\max_{\theta} \log p(x|\theta) + \log p(\theta)$$

  which is computationally feasible.

- This is the **Maximum a Posteriori (MAP)** estimate

Supervised Learning

- Make a prediction of an output y given an input x
- Boils down to determining the conditional probability

$$p(y|x)$$

- Formulate problem as that of finding θ for a parametric family (Maximum Likelihood)

$$p(y|x;\theta)$$

Supervised Learning

- **Example**: Binary classification $y \in \{0, 1\}$
- We aim at determining $p(y = 1|x; \theta) = \sigma(\theta^\top x)$

  where $\sigma(z) = \dfrac{1}{1 + e^{-z}}$ is the sigmoid function
- Class y=1 can be picked when

$$p(y = 1|x; \theta) > p(y = 0|x; \theta)$$

  which is equivalent to $\theta^\top x > 0$

This is logistic regression.
In this case we learn parameters \theta such that data is split into two half-spaces by a hyperplane.
One side is associated to class 1 and the other to class 0.
Such formulation is however quite limited: In the case of images we compute linear combinations of pixel intensities. As shown in the next slide, this might be not good enough.

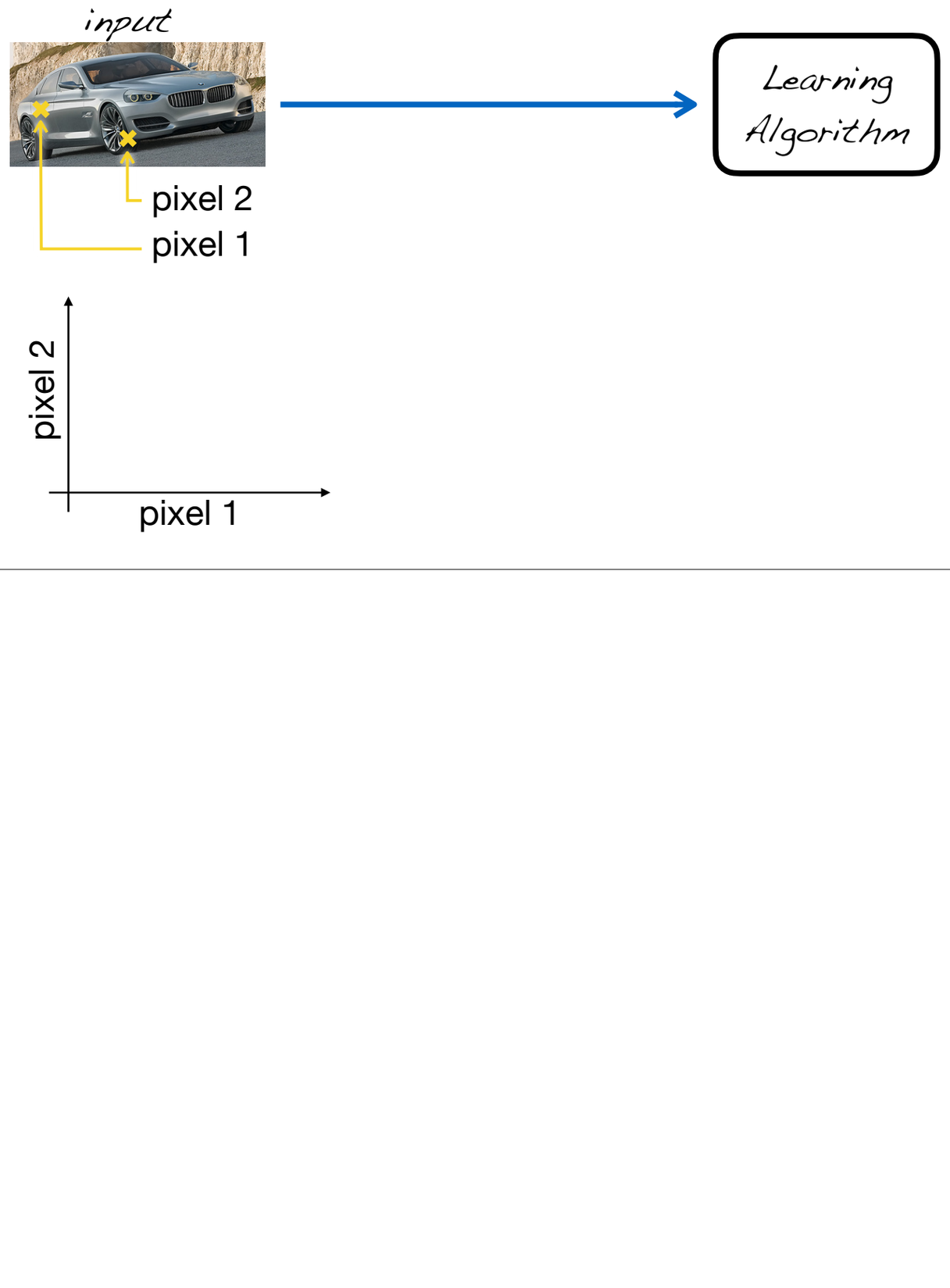
Features

Working directly on the raw data leads to a very tangled set of points, which is quite difficult to separate (into the two classes).

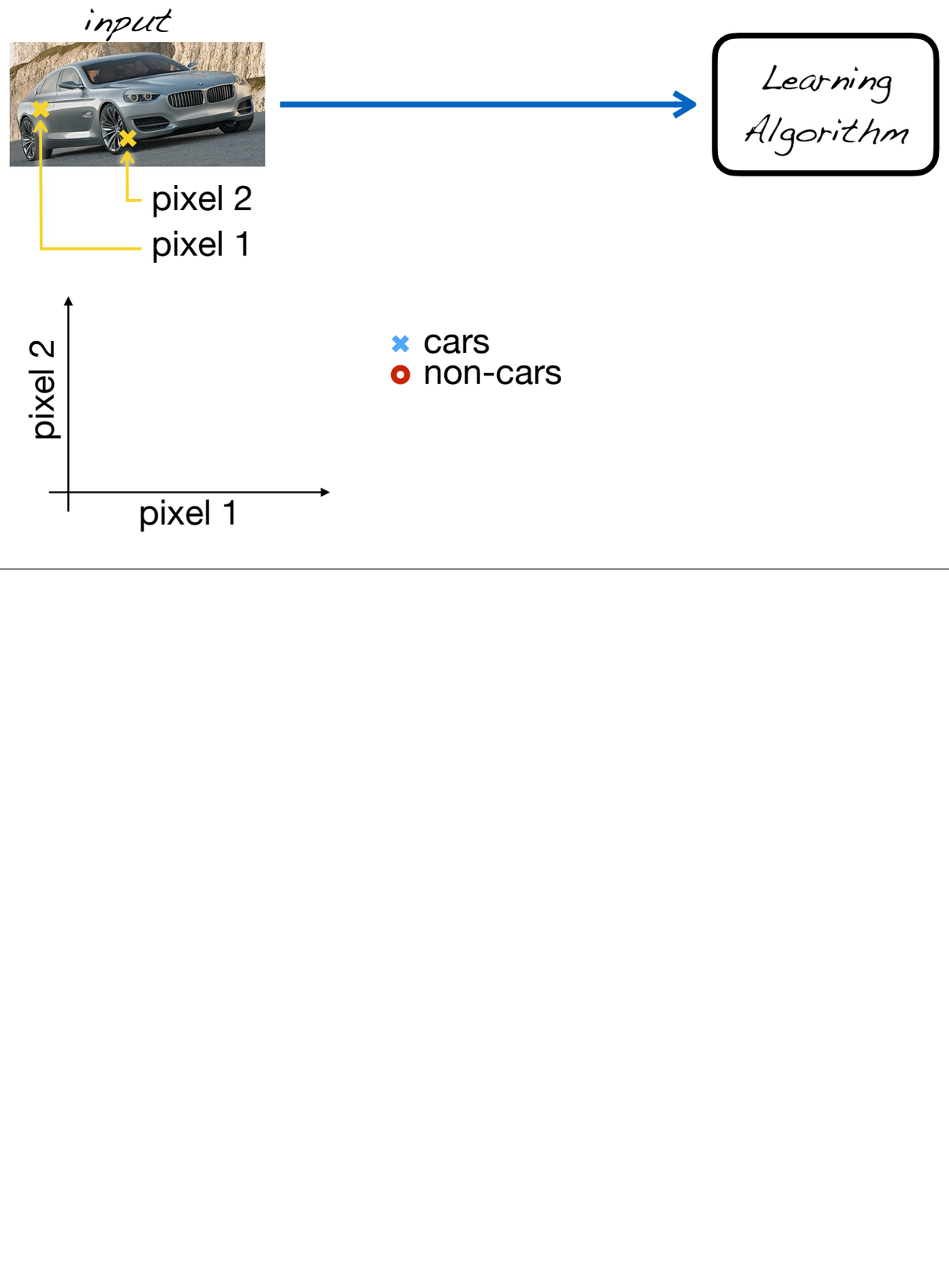
Features

Features

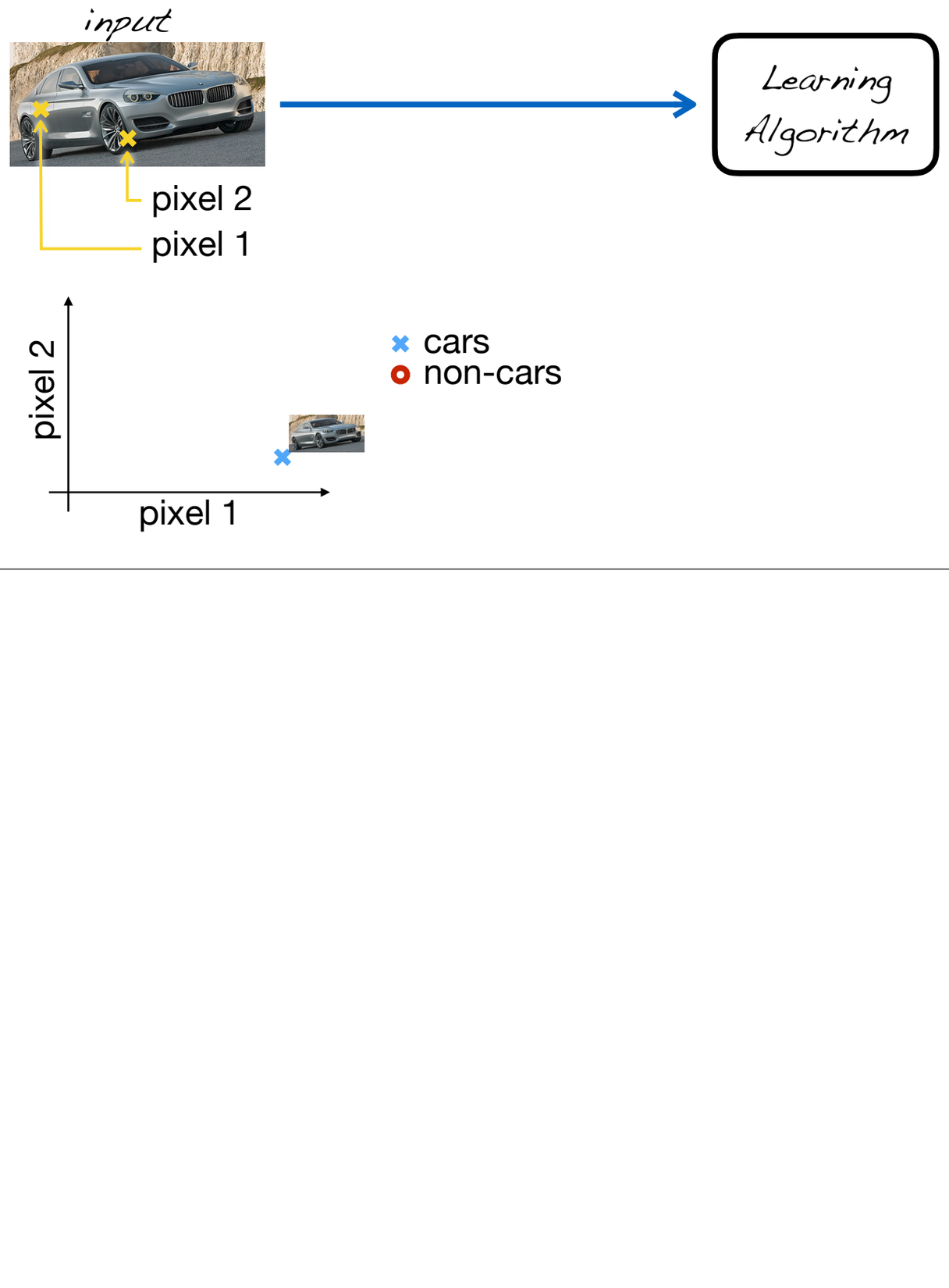
Features



Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

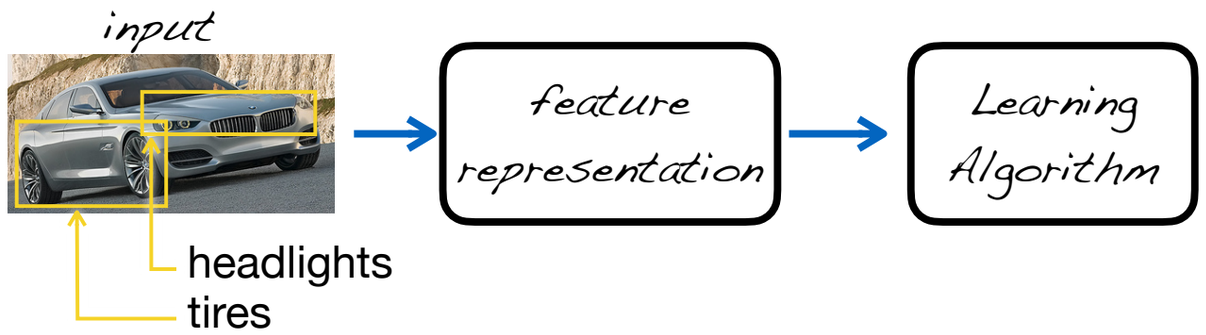
Features


input
pixel 2
pixel 1
Learning Algorithm
pixel 2
pixel 1
cars
non-cars

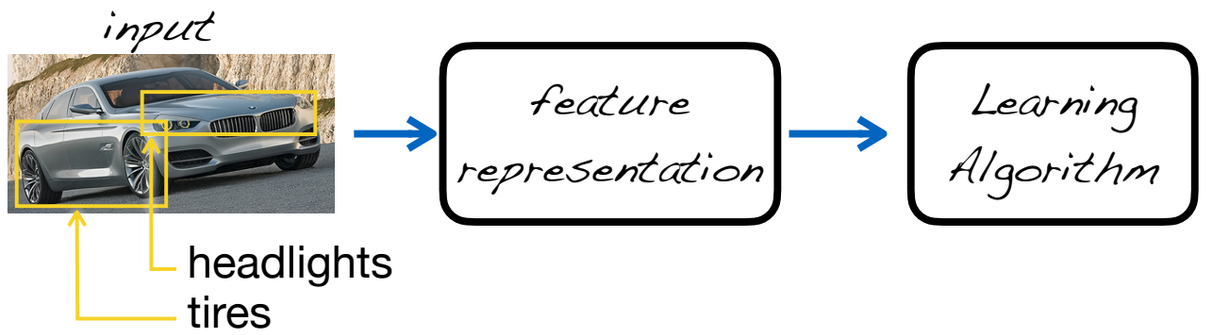
Features

The separation may become easier if we use some intermediate representation (features).
This shows then two problems: one is the identification of suitable features (so that important factors of variation are separated — in the example the category car) and the other is the classification of the features of the input data. The classification can be achieved, for example, with SVM (see next slide).
Unfortunately, the above features are as difficult to detect as much as the original problem (car detection). Therefore, typically the features are much less ambitious (low-level) statistics of the image.

Features


input
feature representation
Learning Algorithm
headlights
tires
pixel 2
pixel 1
cars
non-cars

Features


input
feature representation
Learning Algorithm
headlights
tires
pixel 2
pixel 1
cars
non-cars
tires
headlights

Support Vector Machines

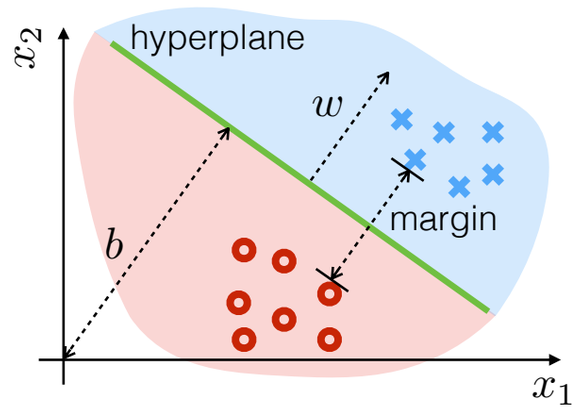
- Binary classification (can be extended to multiclass, regression and structured output)
- Aim is to find a separation between two classes with the largest gap (margin) possible

Support Vector Machines

- The linear classifier becomes an affine classifier

$$\theta^\top x \to w^\top x + b$$

- A geometric interpretation



Support Vector Machines

- A modern formulation is

$$\hat{w}, \hat{b} = \arg\min_{w,b} \lambda |w|^2 + \frac{1}{m} \sum_{i=1}^{m} \max\{0, 1 - y^i (w^\top x^i + b)\}$$

margin normalization

hinge loss

handles non separable sets

- **Formulations**: in the primal domain (see above) or in the dual domain (Lagrangian formulation)
- **Optimization methods**: sub-gradient descent (PEGASOS), stochastic gradient descent, coordinate descent (e.g., sequential minimal optimization, LIBLINEAR), interior point method

Support Vector Machines

- **Features** can be embedded via

$$\hat{w}, \hat{b} = \arg\min_{w,b} \lambda|w|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y^i(w^\top \phi(x^i) + b)\}$$

or via the **kernel trick** in the dual domain

$$\hat{\alpha} = \arg\min_{\alpha} \sum_{i=1}^{m} \alpha^i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha^i \alpha^j y^i y^j k(x^i, x^j)$$

$$\text{subject to } \sum_{i=0}^{m} \alpha^i y^i = 0 \text{ and } 0 \leq \alpha^i \leq \frac{1}{2m\lambda}$$

where we define the symmetric **kernel** $k(x^i, x^j) = \phi(x^i)^\top \phi(x^j)$

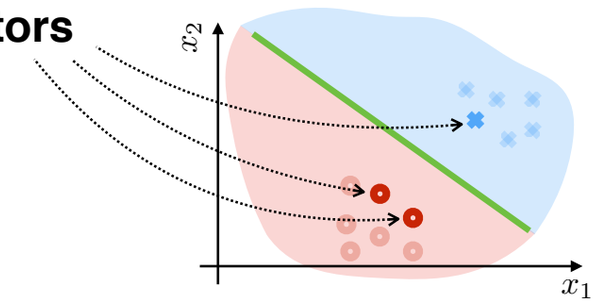
The direct definition and use of the kernel k may lead to a more efficient implementation of the inner product between the feature vectors. E.g., k(x,z) = (x^T z)^2 leads to a feature vector larger than x or z.

Support Vector Machines

- Final classifier can be written as

$$f(x) = w^\top \phi(x) + b = b + \sum_{i=1}^{m} \alpha^i k(x^i, x)$$

- The kernel allows a faster computation of $\phi(x^i)^\top \phi(x)$
- Most α^i are zero. Non-zero entries are called **support vectors**



The final classifier is then compared to zero. If positive then class y=1 is chosen and if negative then class y=0 is chosen.
The kernel allows faster computation of the inner product when it is possible
to write a symmetric function of the two inputs that results in an inner product
between very large feature vectors.
Support vectors correspond to the data points closest to the decision boundary

K-Nearest Neighbors

- Non-parametric method and no training phase
- Used for both classification and regression

in the example k = 3

K-Nearest Neighbors

- Algorithm
  - Search for the k nearest neighbors to x in the dataset
  - Define output y based on the **voting** of the k nearest neighbors (via their associated outputs)

Voting can be a also an averaging operation

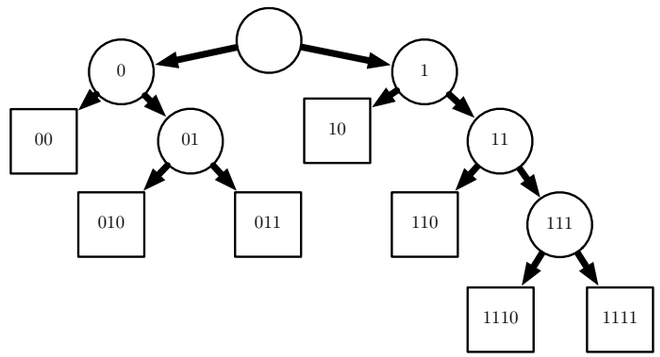
K-Nearest Neighbors

- Pros
  - Simple, no training required
- Cons
  - Memory and computationally inefficient (carry around and search through training set)
  - Poor generalization
  - Does not learn discriminability of features

To illustrate the poor discriminability problem we use an example
Take x\in R^100 from a Gaussian distribution
and suppose that y = x_1 (first component)
Given a new sample x* ideally we would like to take neighbors x that share a very similar x_1 and such that x_1 = x*_1, so that their prediction would be correct.
However, k-nearest neighbors does not learn that x_1 is the only parameter that needs to be used
because the neighbors are detected based on the distance between x* and x and the distance is based on x_1 as well as all other entries too.
Since all the components of x will be used, then k-nearest neighbors will pick neighbors with quite different x_1 components and thus make random assignments.

Decision Trees

- Break space into partition as k-nearest neighbors
- Each node is associated to a region in space
- Internal nodes split into children regions (axis -aligned)



Easy to define and execute.
May require many levels with decisions that are not axis-aligned.

Unsupervised Learning

- Aim is to find a suitable **data representation**
  - Probability density estimator
  - Sampling procedure
  - Data denoising
  - Manifold learning
  - Clustering

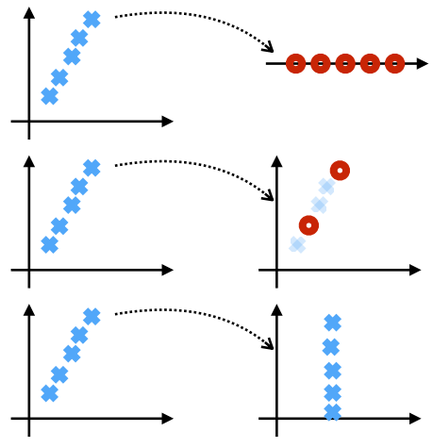
A classic unsupervised learning task is to find the “best” representation of the data. By ‘best’ we can mean different things, but generally speaking we are looking for a representation that preserves as much information about x as possible while obeying some penalty or constraint aimed at keeping the representation simpler or more accessible than x itself.

Unsupervised learning is associated with density estimation, learning to draw samples from a distribution, learning to denoise data from some distribution, finding a manifold that the data lies near to, or clustering the data into groups of related examples.

Often, UL refers to learning techniques that do not use human labour to annotate samples.

Data Representation

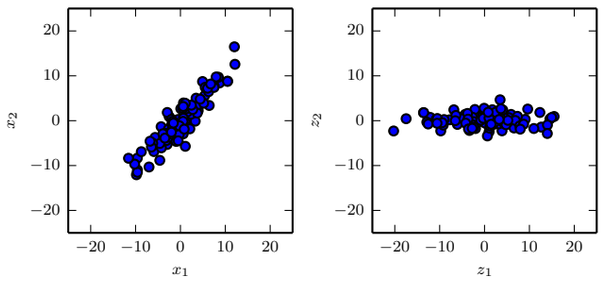
- The ideal data representation should:
  1. **Preserve** all task-relevant information
  2. Be **simpler** than the original data and **easier** to use
     - (i) low-dimensional
     - (ii) sparse
     - (iii) independent



meanings of a simple representation:
low-dim -> compress as much as possible data
sparse -> use as few components as possible to represent the data
independent -> disentangle sources of variation (generating the data) so that the dimensions of the representation are statistically independent

Principal Components Analysis

- **Definition**: Project data X so that the largest variation of the projected data $Z = U^\top X$ is axis-aligned



Any matrix X (rectangular non symmetric) can be decomposed as the product of three matrices U, S and V. Two matrices (U and V) are rotation matrices and S is a diagonal matrix where all the entries in the diagonal are positive and sorted from the largest to the smallest.

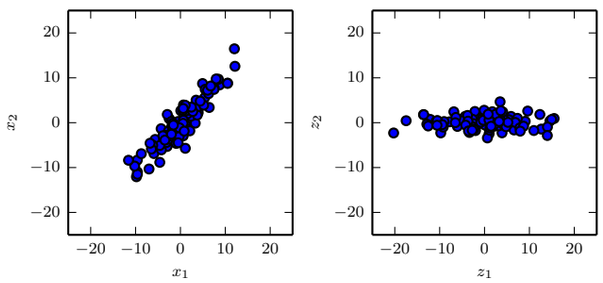
The matrices in U,S,V can be all rectangular (with matching dimensions).

The above factorization is called **singular value decomposition**.

Principal Components Analysis

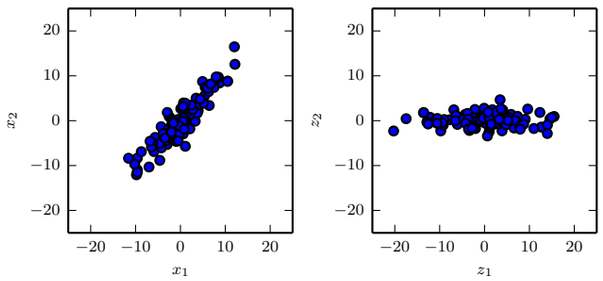
- **Definition**: Project data X so that the largest variation of the projected data $Z = U^\top X$ is axis-aligned

$$X = U\Sigma V^\top$$



Principal Components Analysis

- **Definition**: Project data X so that the largest variation of the projected data $Z = U^\top X$ is axis-aligned

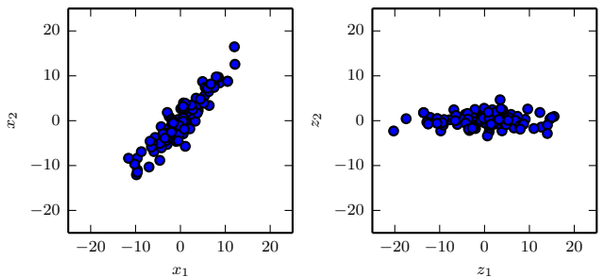


$$X = U\Sigma V^\top$$

$$U^\top U = I$$

Principal Components Analysis

- **Definition**: Project data X so that the largest variation of the projected data $Z = U^\top X$ is axis-aligned



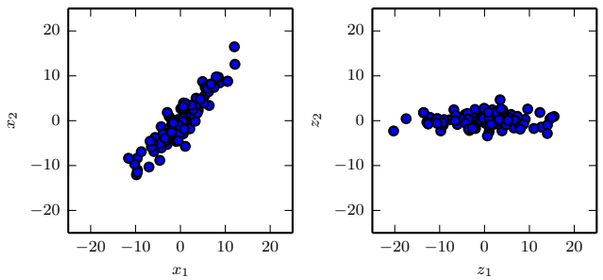
$$X = U\Sigma V^\top$$

$$U^\top U = I \qquad \Sigma = \begin{bmatrix} \sigma_0 & 0 & \dots & 0 \\ 0 & \sigma_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{bmatrix}$$

singular values $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_n \geq 0$

Principal Components Analysis

- **Definition**: Project data X so that the largest variation of the projected data $Z = U^\top X$ is axis-aligned



$$X = U\Sigma V^\top$$

$$U^\top U = I \qquad \Sigma = \begin{bmatrix} \sigma_0 & 0 & \dots & 0 \\ 0 & \sigma_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n \end{bmatrix} \qquad V^\top V = I$$

singular values $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_n \geq 0$

Principal Components Analysis

- Unsupervised learning method for **linearly** transformed data
- A low-dimensional representation (by thresholding the singular values)
- Yields independent (uncorrelated) components

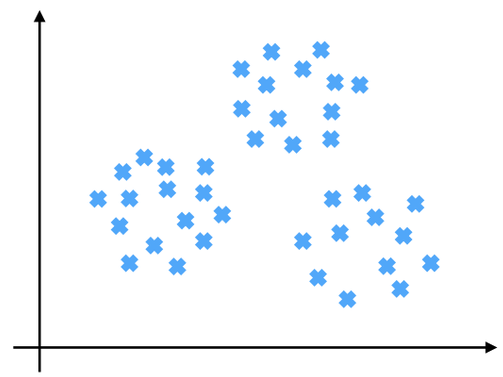
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



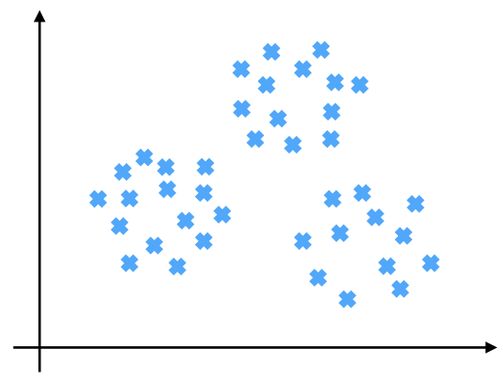
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



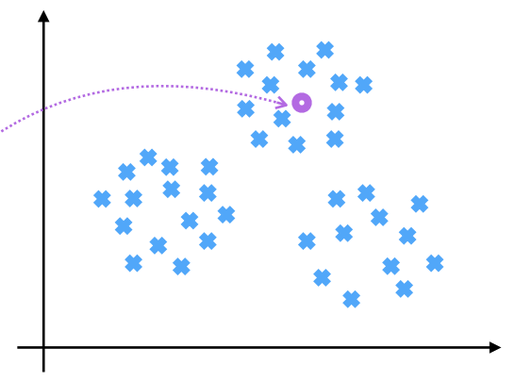
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



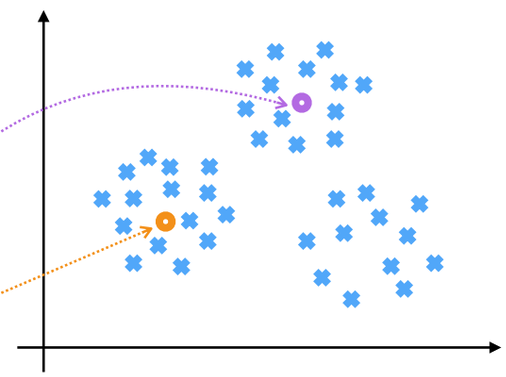
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



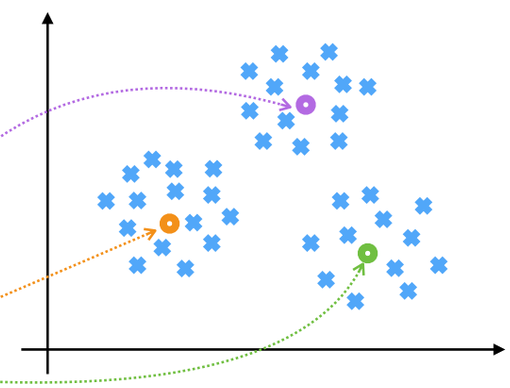
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



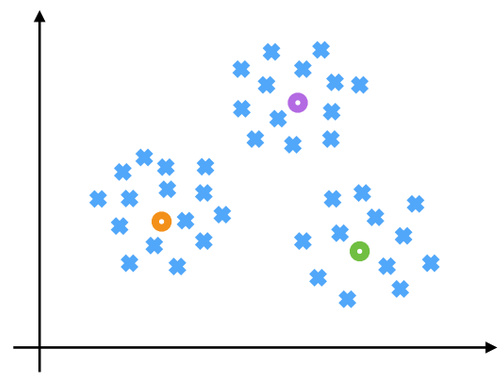
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



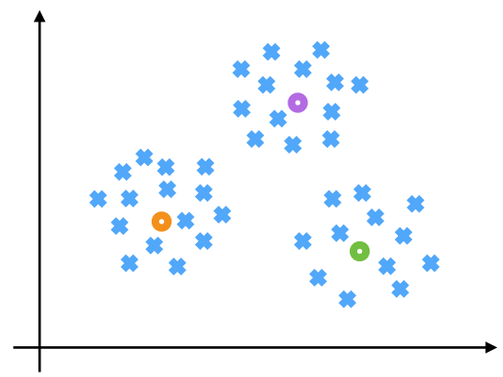
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



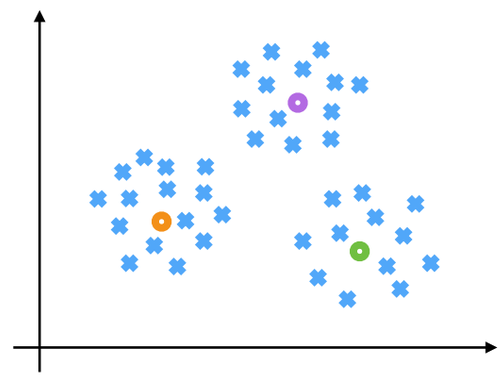
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



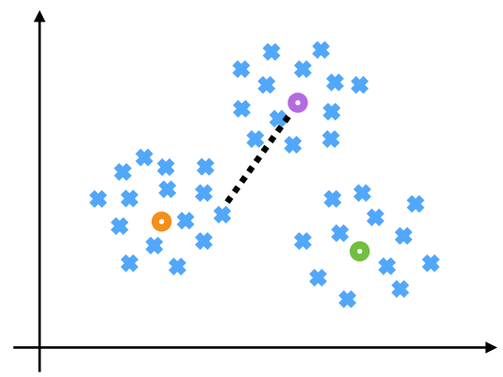
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



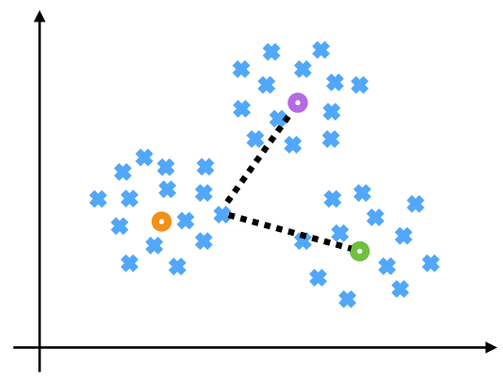
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



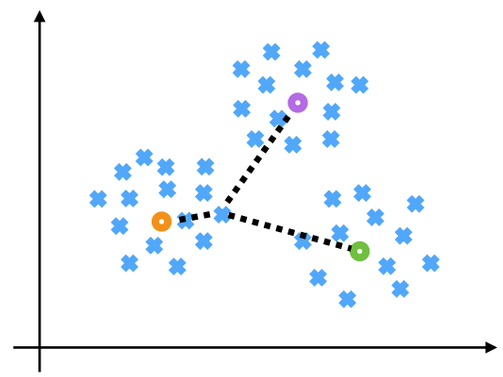
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



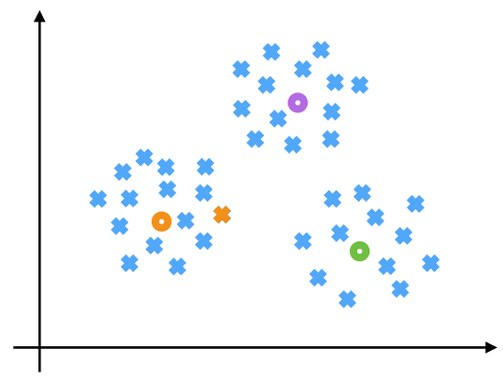
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



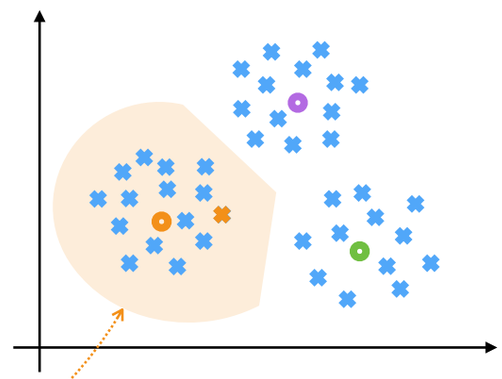
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



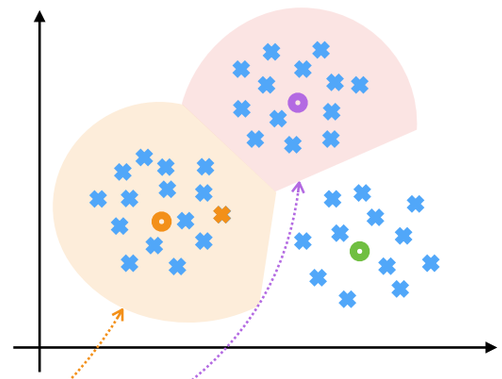
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



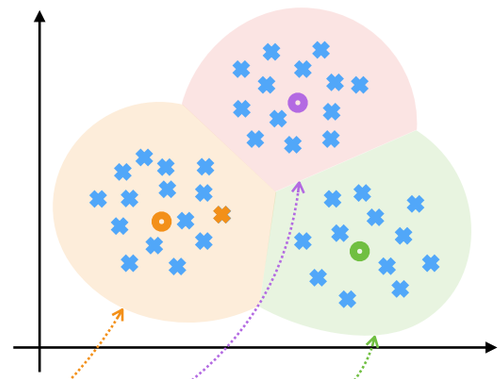
K-Means Clustering

- **Definition**: Find k clusters of data samples similar to each other

  Alternate between:

$$c_j = \frac{\sum_i \delta[w_i = j] x_i}{\sum_i \delta[w_i = j]}$$

$$w_i = \arg\min_j |x_i - c_j|^2$$



K-Means Clustering

- Unsupervised learning method (handles nonlinearly transformed data)
- A sparse representation (assignments w_i encode one sample with one of the cluster centers c_j)
- Depends on initialization
- Ill-posed (multiple solutions can be valid)
- Number of clusters is usually unknown

K-Means clustering is a useful and reasonably efficient method. Multiple initialisations might be needed.

Building a Machine Learning Algorithm

1. Build a dataset
2. Define a model
3. Define a cost function
4. Define an optimization procedure

The general recipe is: combine a specification of a dataset, a cost function, an optimization procedure and a model.
Most algorithms in machine learning can be formulated in terms of these components. Even unique or hand-designed algorithms can often be seen as a special case of the above scheme. The optimization procedure in fact may not be a gradient descent method but something analogous.

Example

- **Dataset** of cars and non-cars

x				
	$y = 1$ (cars)		$y = 0$ (non-cars)	

Example

- **Model**

$$p(y = 1|x; \theta) = \sigma(\theta^\top x) \quad \text{with} \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

logistic regression

Example

- **Cost function**

$$\sum_{i=1}^{m} \log p(y^i|x^i;\theta) = \sum_{i=1}^{m} \log \sigma(\theta^\top x^i)^{y^i} \left(1-\sigma(\theta^\top x^i)\right)^{1-y^i}$$
$$= \sum_{i=1}^{m} y^i \log \sigma(\theta^\top x^i) + (1-y^i) \log \left(1-\sigma(\theta^\top x^i)\right)$$
$$= \sum_{i=1}^{m} y^i \theta^\top x^i - \log \left[e^{-\theta^\top x^i}\left(1+e^{-\theta^\top x^i}\right)\right]$$

maximum likelihood

Example

- **Optimization procedure**

$$\theta_{t+1} = \theta_t + \alpha \sum_{i=1}^{m} \left(y^i - \sigma(\theta_t^\top x^i) \right) x^i$$

(batch) gradient ascent

Stochastic Gradient Descent

- A variation of (batch) gradient descent introduced to handle large training sets
- Applies to cost functions written as a sum of training samples (i.e., in the form of an expectation)

Stochastic Gradient Descent

- **Idea**: approximate gradient by using one (or a few) sample(s)
- Previous example: $\theta_{t+1} = \theta_t + \alpha \sum_{i \in Z_t \subset [1,m]} \left(y^i - \sigma(\theta_t^\top x^i)\right) x^i$
- **stochastic gradient descent** when Z_t singleton; **minibatch gradient descent** when Z_t larger; **incremental gradient method** (Bertsekas and Tsitsiklis 2000)

Bertsekas and Tsitsiklis refer to the stochastic GD as incremental gradient method
when the sum is limited to one sample and the set Zt cycles from the first sample to the last.

Stochastic Gradient Descent

- SGD will converge to a local minimum under some smoothness conditions on the cost
- Is the **fundamental optimization method** used in deep learning

Example (revisited)

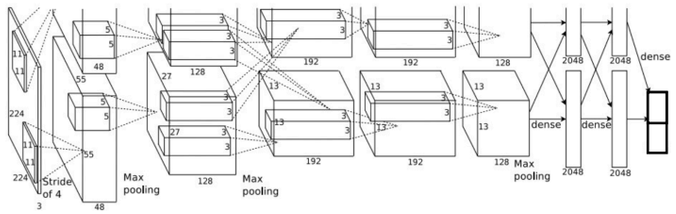
- **Dataset** of cars and non-cars (>1M samples)

x				
	$y = 1$ (cars)		$y = 0$ (non-cars)	

Example (revisited)

- **Model**

$$p(y|x;\theta) =$$



binary output

convolutional neural network

The output of the CNN is the probability p(y|x;\theta); so there are only two possible entries in the output vector and they must add up to 1 and be positive. \theta are all the CNN parameters (convolutional filters and weights in the fully connected layers)

Example (revisited)

- **Cost function**

$$\sum_{i=1}^{m} \log p(y^i | x^i; \theta)$$

negative cross entropy (maximum likelihood)

The cross entropy is defined as E_p[-log q] where p is the true p(y|x) and q = p(y|x;\theta) is the approximation we estimate

Example (revisited)

- **Optimization procedure**

$$\theta_{t+1} = \theta_t + \alpha_t \frac{\nabla_\theta p(y^i|x^i;\theta_t)}{p(y^i|x^i;\theta_t)} \qquad i \sim U[1,m]$$

stochastic gradient ascent

Challenges in Machine Learning

- Prior to deep learning, methods in ML could solve AI problems (e.g., object recognition) with limited success
- Difficulties
  - High-dimensional data
  - Generalization with high-capacity functions
  - High computational costs

Curse of Dimensionality

- Number of possible configurations grows exponentially with number of (data) dimensions
- We need an exponentially growing number of data samples to cover the configurations of interest

#samples~ $O(v^d)$
d dimensions
v values per dim.

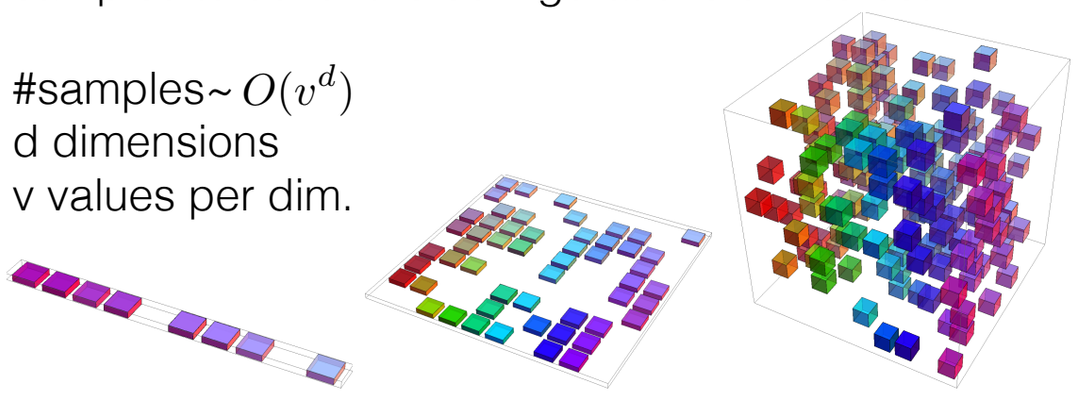


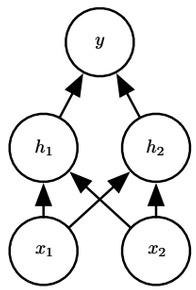
Illustration refers to the nearest neighbor method. As long as we can collect enough samples nearby any location, we can say something about that location. However, as the dimensions grow, the locations that do not get any sample grow very fast.

Locality and Smoothness

- Most ML algorithms make use of local constancy or smoothness
- This allows limited generalization
- For example, in the case of nearest neighbor, one needs at least one example per region of interest (the prediction function is as complex as the data)

Locality and Smoothness

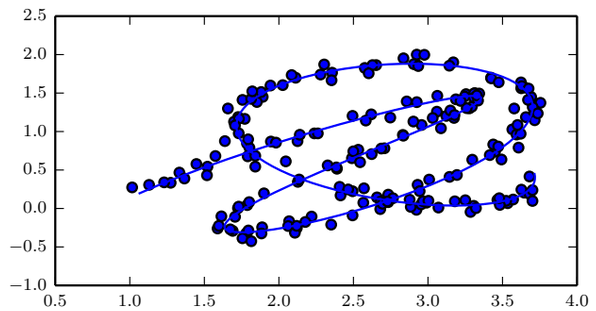
- We need to generalize well complex functions with few examples
- Deep learning does so by introducing assumptions on the data generating distribution
- The key assumption is that data is generated by a **composition of factors** (typically, in a hierarchical structure)



As in the illustration, the output y is obtained from a chain of compositions.
y depends only of h1 and h2 and each of them depends on only x1 and x2.
This controlled complexity limits the possible combinations.

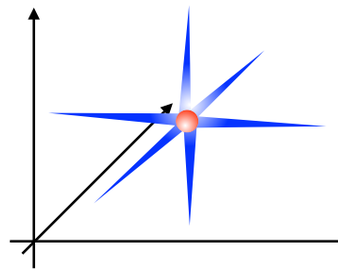
Manifold Learning

- In ML manifold learning aims at finding a low-dimensional embedding to represent data
- Example: a 1D curve in 2D space
- Assumes that data lies only near given samples



Manifold Learning

- Images occupy a very small portion of the pixel space (adding noise is not likely to generate another real image)



realistic images generated by simultaneously changing intensities of entire regions of pixels

- Similar argument for language (random sets of letters are not likely to generate English words)

Example

- Manifold is 2D (2 viewpoint angles)

Example: all face images of a person

1000x1000 pixels = 1,000,000 dimensions

But the face has 3 cartesian coordinates and 3 Euler angles

And humans have less than about 50 muscles in the face

Hence the manifold of face images for a person has <56 dimensions

Practical Methodology

- We have seen a recipe for building a ML algorithm
- In practice, is the implementation so straightforward?
- What can we do when we have a limited time and resource budget?

credits to Andrew Ng - University of Stanford

Strategy

1. **Look at/Plot the data**

2. Start with the simplest (but non trivial) implementation

3. Test it on validation data

   (i) **Diagnostics**: Plot learning curves to decide changes (e.g., more training data, better features etc)

   (ii) **Error/Ablative analysis**: Manually examine samples in the validation data where your algorithm makes mistakes; see if there are any patterns/trends; determine what works and what does not.

The simplest implementation is typically also the fastest and less likely to have coding bugs. Therefore, testing is more efficient and reliable.
The diagnostics will be explained more in details in the next slides.
The error analysis allows discovering limitations in our algorithm (model, dataset, optimization methods)
Ablative analysis allows determining what contributes to the result and what does not (and to what degree)

Debugging

- Two separate tasks
  - Debugging code: Code does not implement correctly the algorithm
  - Debugging algorithms: The algorithm does not perform as desired

We do not discuss the code debugging. For that part the best practices of software engineering should provide useful guidelines. Some of the concepts that will be discussed later on could also help code debugging. Also, some basic techniques like the use of toy examples should suffice to perform basic sanity checks.
In the next slides instead we focus on the more challenging task of determining if our algorithm (our idea) works as desired. This is a more challenging task and we provide a number of useful directions.

Debugging

- Two separate tasks
  - Debugging code: Code does not implement correctly the algorithm
  - Debugging algorithms: The algorithm does not perform as desired

Diagnostics

- **Machine Learning Diagnostics**
  A test that you can run to gain insight on what is and isn't working with a learning algorithm and to gain insight on how to best improve its performance

Debugging Learning Algorithms

- Example: Spam/Non-spam classification
- You carefully chose features (100 words out of 50K in English)
- Bayesian logistic regression implemented with gradient descent gets 20% error, which is too high

$$\max_{\theta} \sum_{i=1}^{m} \log p(y^i|x^i;\theta) - \lambda|\theta|^2$$

- What do we do next?

Debugging Learning Algorithms

- Common attempts (trial and error)

  (a) Try getting more training examples

  (b) Try a smaller set of features

  (c) Try a larger set of features

  (d) Try changing the features: Email header vs. email body features

  (e) Run gradient descent for more iterations

  (f) Try Newton's method

  (g) Use a different regularization parameter

  (h) Try using an SVM

- This approach might work, but it's **very time-consuming**, and largely a matter of luck whether you end up fixing what the problem really is

Debugging Learning Algorithms

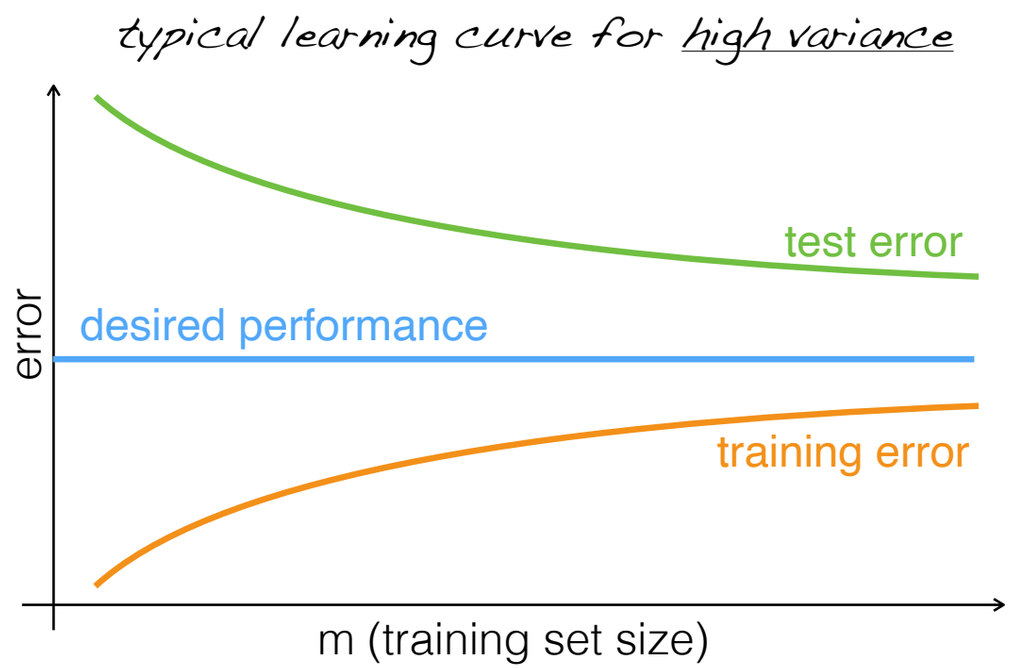
- Better approach
  1. Run **diagnostics** to find the problem
  2. Fix whatever the problem is

Debugging Learning Algorithms

1. Start with stating the problem: Bayesian logistic regression's test error is 20% (unacceptably high)
2. Make hypotheses for what the problem could be. For example,
   - Overfitting (high variance)
   - Too few features to classify spam (high bias)
3. Run diagnostics. For example,
   - Variance: Training error will be much lower than test error
   - Bias: Training error will also be high

In the next example we will see one of the most common issues with ML algorithms: underfitting and overfitting. This is also called variance vs bias. It is very important to be able to determine if any of the two conditions is occurring so that the correct countermeasures can be employed.

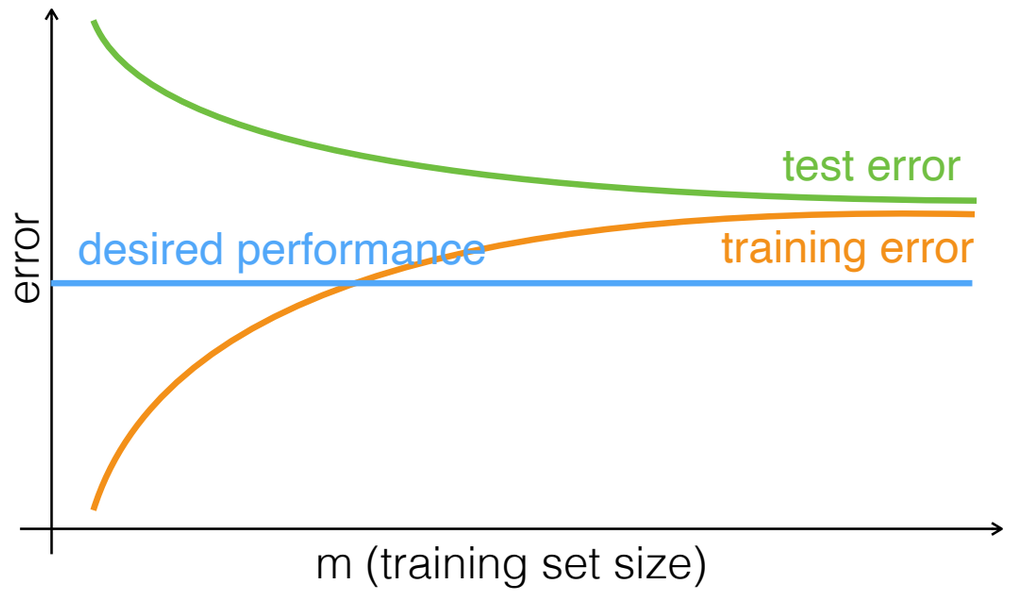
Bias vs Variance



Test error is still decreasing as m increases: a larger training set will help
There is a large gap between training and test error

Bias vs Variance

typical learning curve for high bias



Training error is too high

There is a small gap between training and test error

Fix Based on Diagnostics

- Fixes (and diagnostics)
  - (a) Try getting more training examples ⇢ fixes high variance
  - (b) Try a smaller set of features ⇢ fixes high variance
  - (c) Try a larger set of features ⇢ fixes high bias
  - (d) Try changing the features ⇢ fixes high bias
  - (e) Run gradient descent for more iterations
  - (f) Try Newton's method
  - (g) Reg. parameter (lower -> fixes high bias, higher -> fixes high var.)
  - (h) Try using an SVM

Other Diagnostics

- Bias vs variance is one common diagnostic
- In general, you need to construct your own diagnostics based on the specific problem at hand
- Other examples
  - Is the algorithm converging?
  - Are you optimizing the right function?
  - Is the model correct?
  - What is the best regularization value?

Hard to tell if an algorithm converges by looking at the objective function

Optimization Diagnostics

- Example
  - Bayesian logistic regression gets 2% error on spam and 2% error on non-spam (too high)
  - SVM with a linear kernel gets 10% error on spam and 0.01% error on non-spam (acceptable)
  - You want to use logistic regression because of efficiency
- What to do next?

Optimization Diagnostics

- SVM outperforms Bayesian logistic regression (BLR) but you want to deploy BLR
- You care about the weighted accuracy

$$a(\theta) = \max_{\theta} \sum_i w^i \delta[h_\theta(x^i) = y^i]$$

  and have $a(\theta_{SVM}) > a(\theta_{BLR})$

- BLR maximizes $J(\theta) = \sum_i \log p(y^i|x^i;\theta) - \lambda|\theta|^2$
- Diagnostic $J(\theta_{SVM}) > J(\theta_{BLR})$?

Optimization Diagnostics

- Case 1: $J(\theta_{SVM}) > J(\theta_{BLR})$

  BLR fails to maximize J (problem is with convergence/optimization algorithm)

- Case 2: $J(\theta_{SVM}) < J(\theta_{BLR})$

  BLR succeeds at maximizing J. Then, J is the wrong objective if we care about a

Fix Based on Diagnostics

- Fixes (and diagnostics)

(a) Try getting more training examples	⇢	fixes high variance
(b) Try a smaller set of features	⇢	fixes high variance
(c) Try a larger set of features	⇢	fixes high bias
(d) Try changing the features	⇢	fixes high bias
(e) Run gradient descent for more iterations	⇢	fixes opt. algorithm
(f) Try Newton's method	⇢	fixes opt. algorithm
(g) Reg. parameter	⇢	fixes opt. objective
(h) Try using an SVM	⇢	fixes opt. objective

Error analysis

- Explain the difference between current performance and perfect performance
- Understand what the sources of error are
- How much error is attributable to each component in the algorithm?
- Plug in ground truth in each component (if possible) and see how accuracy changes

Ablation Analysis

- Explain the difference between current performance and some baseline performance (much poorer)
- Which component of the algorithm really helps?
- Remove components one at a time and see how the algorithm breaks

More Tools

- Consider
  - Toy examples
  - Simple data first
  - Conditions where the result is known
  - Cases where analytic solutions are available
  - One source (input or intermediate) at a time

More Tools

- Consider
  - Set most items to known working settings
  - Simulating mentally the behavior of the algorithm
  - Worst-case scenario and conditions leading to it
  - Input data and settings that break the algorithm
  - Extreme conditions (meaningful behavior?)