Story of Machine Learning

Syllabus:

Decision Tree 18 – 18.3.4
Evaluation 18.4
Model Selection 18.4.1
Regularization 18.4.3
Theory 18.5.0
Regression 18.6 – 18.6.2
Classification 18.6.3 – 18.6.4
Neural Network 18.7 – 18.7.4 (exclude exotic varieties of NN in my slides)
Non-parametric models 18.8 – 18.8.4
SVM basics 18.5
Clustering basics
…

INFORMATION ENTROPY:

If one were to transmit sequences comprising the 4 characters 'A', 'B', 'C', and 'D', a transmitted message might be 'ABADDCAB'. Information theory gives a way to calculate the smallest possible amount of information that will convey this.

If all 4 letters are equally likely (25%) in a text, one can't do better (over a binary channel) than to have 2 bits encoding for each letter: 'A' might code as '00', 'B' as '01', 'C' as '10', and 'D' as '11', i.e., 2 bits per letter.

If 'A' occurs with 70% probability, 'B' with 26%, and 'C' and 'D' with 2% each, and we are allowed to assign variable length codes, 'A' would be coded as '0' (one bit), 'B' as '10', and 'C' and 'D' as '110' and '111'. It is easy to see that 70% of the time only one bit needs to be sent, 26% of the time two bits, and only 4% of the time 3 bits. On an average, fewer than 2 bits will be required since the *entropy* is lower (owing to the high prevalence of 'A' followed by 'B' – together 96% of characters) than that with equal probability. Overhead of transmitting the encoding of letters is additional but minimal.

The calculation of the sum of *weighted log probabilities* measures and captures this effect.

https://en.wikipedia.org/wiki/Entropy_(information_theory)

Decision Tree: Choice of attribute at each level

Entropy of Target Examples (current level):

$H(Goal) = P(v_k) \sum_k \log_2 (1/P(v_k))$, *k may be {True, False}*

Say, 8 positive examples, and 4 negative examples

$H(Goal) = B(8/12) = (8/12) \log_2 (12/8) + (4/12) \log_2 (12/4)$

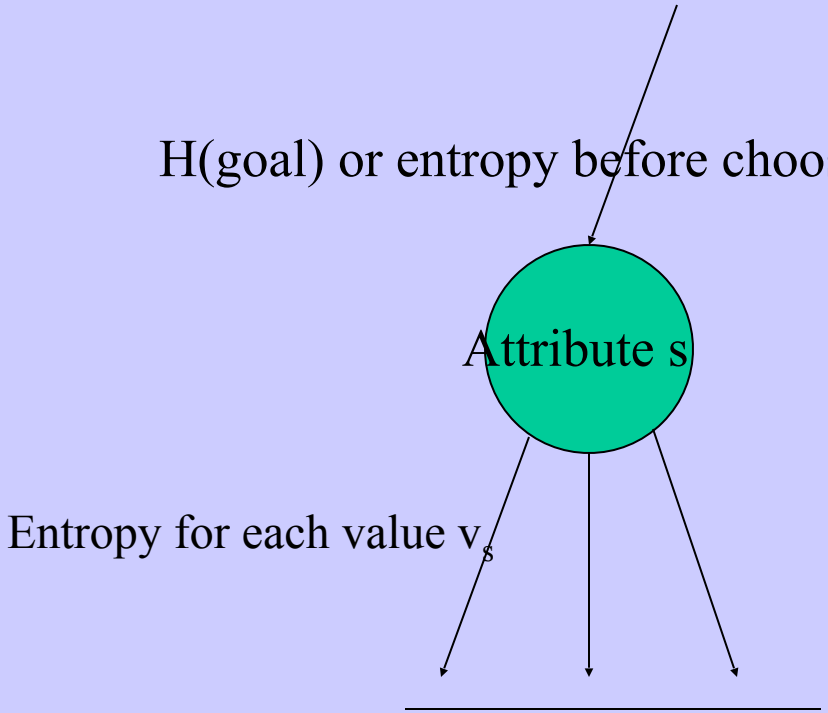
Now, for attribute A, calculate entropy:

but say, attribute A has three values {some, full, none}

Say, v_s, v_f, v_n *are number of examples for these three types, and*

(p_s, n_s*) are positive and negative examples for the Target-label attribute (to go to restaurant or not) corresponding to A=some*

such that $p_s + n_s = v_s$ *, and so are for* (p_f, n_f) , (p_n, n_n)


H(goal) or entropy before choosing attributes
Attribute s
Entropy for each value v_s
Aggregate entropy= *weighted* sum for all attribute values

Decision Tree: Choice of attribute at each level

Now, for attribute A, calculate the entropy:
but say, attribute A has three values {some, full, none}
Say, v_s, v_f, v_n are number of examples for these three types, and

(p_s, n_s) are positive and negative examples for the Target attribute (to go to restaurant or not) such that $p_s + n_s = v_s$, and so are (p_f, n_f) , (p_n, n_n)

$$\text{Entropy, } R(A) = (v_s/\text{total})[(p_s/v_s)\log_2(p_s/v_s) + (n_s/v_s)\log_2(n_s/v_s)] + (v_f/\text{total})[(p_f/v_f)\log_2(p_s/v_s) + (n_f/v_f)\log_2(n_f/v_f)] + \text{the same for A=“none”}$$

$$\text{total} = v_s + v_f + v_n$$

Decision Tree: Choice of attribute at each level

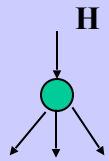
Now, for choosing attribute A, information gain will be

$$Gain(A) = H(Goal) - R(A)$$

Compute this gain for each attribute at the current level,

Best attribute provides highest information gain (or lowest entropy relative to Goal-entropy)

Decision Tree: Choice of attribute at top level in the Restaurant example in book



H(Goal), for 6 pos and 6 neg examples in total, =1 bit

*Gain(Patrons) = 1 – [(2/12)B(0/2) + (4/12)B(4/4) + (6/12)B(2/6)] = **0.541 bit***

*Gain(Type) = 1 – [(2/12)B(1/2) + (2/12)B(1/2) + (4/12)B(2/4) + (4/12)B(2/4)] = **0 bit***

B(q) is the entropy for a Boolean variable, with q=positives/total,
$B(q) = q\log_2(1/q) + (1-q)\log_2(1/(1-q))$

MISCELLENEOUS: Machine Learning has two stages:
Do not forget

Stage 1: Training, with "known" data (in supervised learning: input and known labels to generate model)

Stage 2: Inferencing (deploying trained ML model to its task: predict unknown label given input attribute values)

Stage 2.1: Validating with "unknown" data to quantify how good the trained model is

Evaluation of Algorithm

Training set = Sample of real world

Stationarity assumption: Real world has the same distribution as that of training data

Non-stationarity: data is changing over time, what you learned before is no longer useful

Independent and Identically distributed (iid): Each datum, training or in real world, has equal probability of appearing

Non-iid: Some data are more important than other

Evaluation of Algorithm

Cross-validation: divide data set into two groups - training and validation,
for computing the rate of successful classification of test data.

Measurement of validation: error rate on validation set

An ML algorithm has many parameters: e.g., model (order of polynomial), learning rate, etc.

Fine-tune those parameters using a WRAPPER algorithm, by repeated validation:
need to repeat cross-validation by randomly splitting the available data set.

Evaluation of Algorithm

k-fold Cross validation: 1/k part of data set is validation set, repeat x-number of times by randomly splitting 1/k

k=n, for data set size n, is leave-one-out cross-validation

Peeking: After k-fold cross-validation, ML algorithm may overfit known training data (if validation is over part of the training set) , but may not be as good for real life use (note: that may mean iid is not true)

ML competitions hold out real test data, but still groups "cheat" by repeatedly submitting fine-tuned code.

Model Selection

Finding best hypothesis: two step process

1)Find best hypothesis space
2)Optimization to find the best hypothesis

E.g., 1) Which order of polynomial, $y=ax+b$, or $y = ax^2 +bx +c$?
2) Find a, b, c parameters values

Regularization

Optimization function may embed simplicity of the model, or any other relevant knowledge

E.g., Cost(h) = Error + λ(complexity)*
Cost(h) = (y – [… $ax + b$*])² + λ*(number of parameters, e.g., for linear, #parameters=2),*
Or, Cost(h) = $(y - [ax^n + bx^{n-1} + cx^{n-2} + \ldots])^2 + \lambda*(a+b+..)$*, the parameter themselves*

- *h is hypothesis, the polynomial*
- *Cost(h) is the error to be minimized*
- *Polynomial embedded may be of arbitrary order*
- *λ is tunable regularization constant or hyper-parameter*
- *(a+b+...) regularization term, lower the better*

$h^* = \text{argmin}_{\{a,b,\ldots\}}$ *Cost(h), is to find parameters a, b, …*

PAC learning – quality of an algorithm:

- *Any seriously wrong hypothesis may be quickly found out with only a few examples*

- *Conversely, any hypothesis that survives after many examples is likely to be correct*

Provably Approximately Correct (PAC) learning algorithm

CLT provides a measure on PAC learning

How many examples can make the model - how accurate?

Sample complexity is studied here

If you want ε–accurate you need f(ε) number of samples, as CLT tries to find the function f(.)

Problem II: Linear Regression Output is Continuous valued

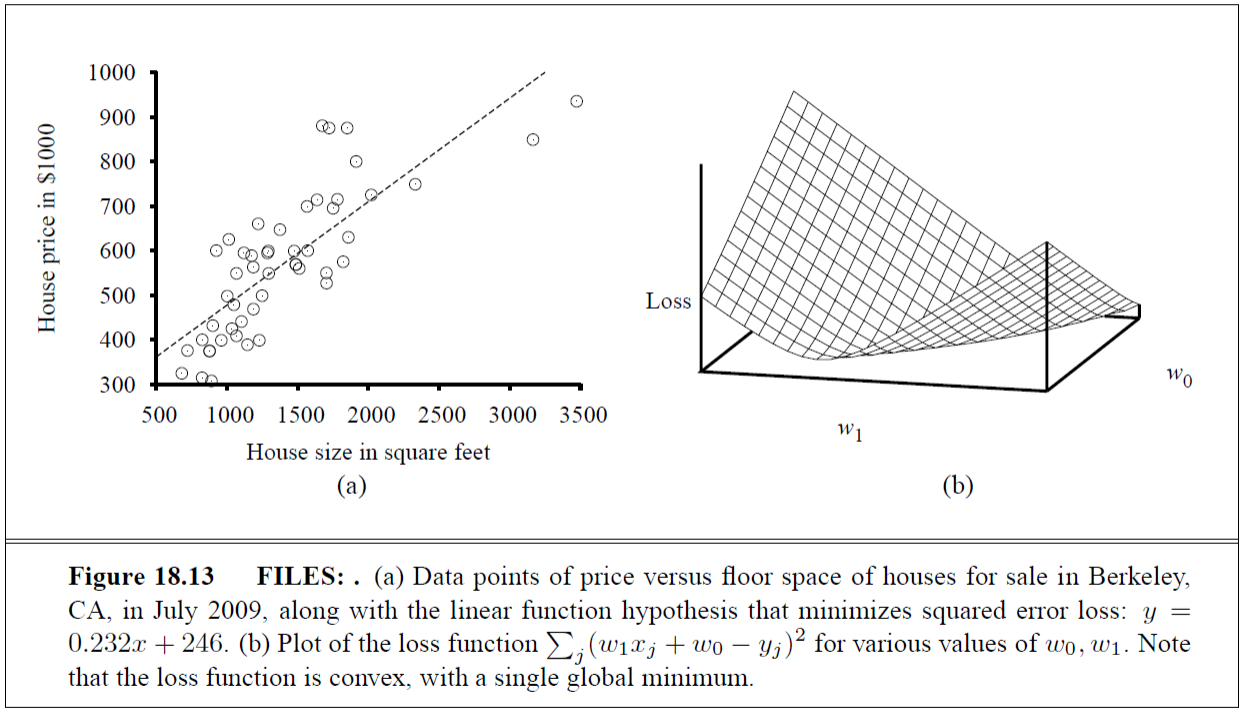


Figure 18.13 **FILES: .** (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

Linear Regression (Still supervised learning)

Regression: Predicting a *continuous value* (e.g., y) out of input x values

Least Square Error or L_2-norm minimization

Linear model, $y = mx + b$: Has closed form solution Eq 18.3

Linear Regression

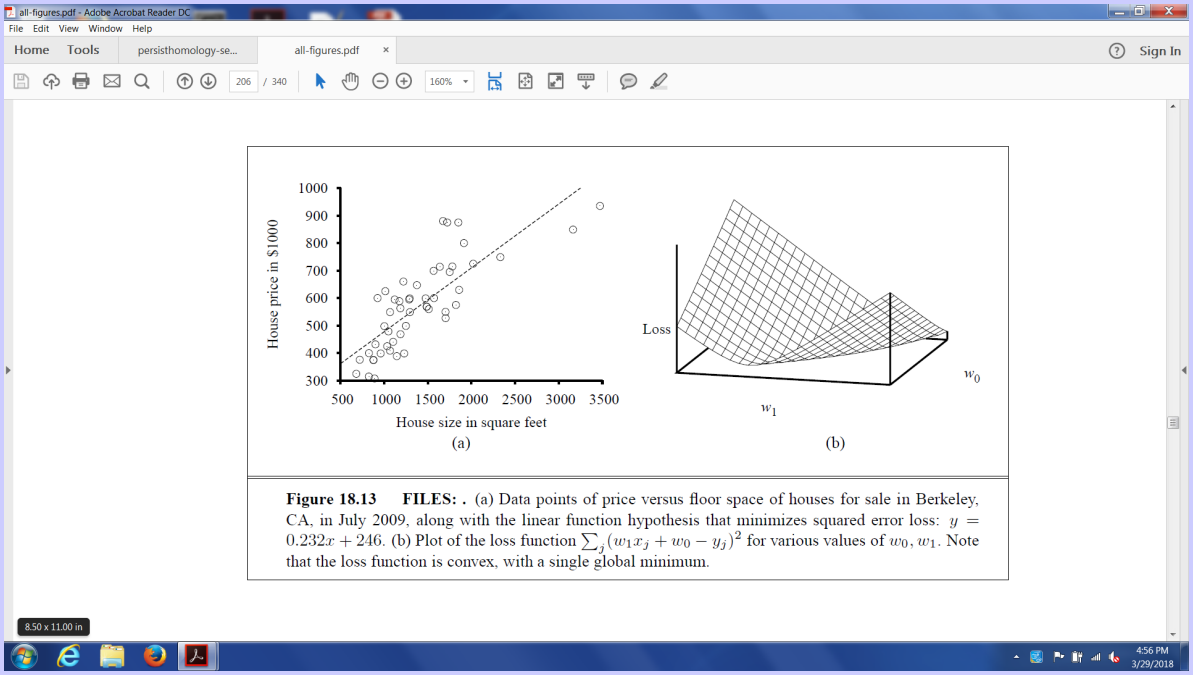


Figure 18.13 **FILES: .** (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

$\text{Loss}(h_{\underline{w}}) = \sum_j (y_j - (w_1 x_j + w_0))^2$, j goes over N data points

Take partial derivatives over w_i and equate each to zero, i runs over parameters.

w_is are parameters that the algorithm learns

$$w_1 = [N(\sum_j x_j y_j) - (\sum_j x_j)(\sum_j y_j)] \; / \; [N(\sum_j x_j^2) - (\sum_j x_j)^2]$$

$$w_0 = [\sum_j y_j - w_1(\sum_j x_j)]] \; / \; N$$

for two parameters

Linear Regression

Sometimes no solution is found in closed form (e.g., non-linear regression)

Gradient Descent gets iteratively closer to the solution:
determine the direction in each iteration and update *w* parameters above
step-size may be updated in each iteration, a constant, or a fixed schedule

Note: "direction" gets determined by the sign of error: *+ve* or *-ve*
(useful in understanding classifier later)

Multivariate Linear Regression

Hypothesis is: $y = w_0 + w_1 x_1 + w_2 x_2 + \ldots$, for $x_1, x_2, \ldots x_n$ variables in n-dimension

Closed form solution is a matrix formulation with partial differential equations equated to 0

Gradient descent is:

$\underline{w} \leftarrow$ start with arbitrary point in the parameter space ($\underline{w}$ vector);
loop until convergence
  for each parameter w_i in $\underline{w}$ do
    $w_i \leftarrow w_i - \alpha * \partial/\partial w_i$ (Loss($\underline{w}$));

α is the *step size* or *learning rate*

Optimizes on parameter-space

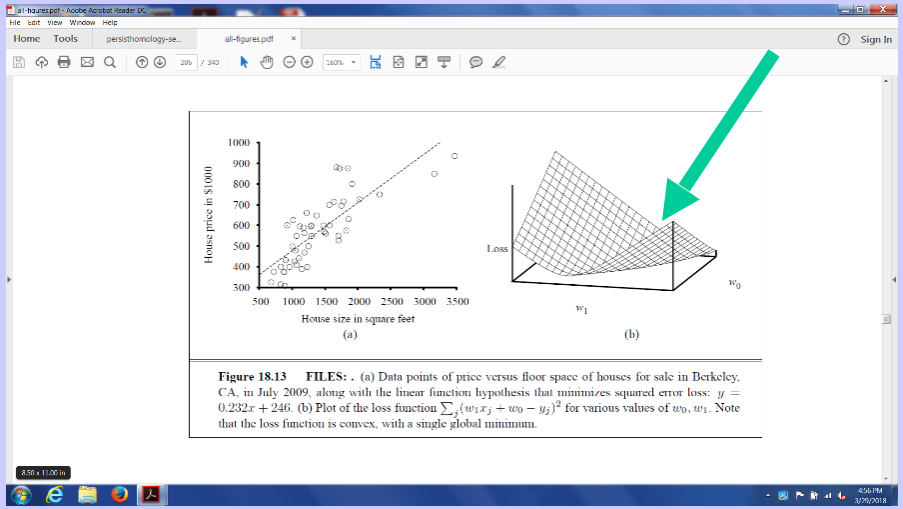


Figure 18.13 **FILES: .** (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (w_1 x_j + w_0 - y_j)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

Multivariate Linear Regression

Hypothesis: $y = w_0 + w_1 x_1 + w_2 x_2 + \ldots$, for $x_1, x_2, \ldots x_n$ variables

Gradient descent update is:
$w_i \leftarrow w_i - \alpha * \partial/\partial w_i \, (\text{Loss}(\underline{w}))$;

Loss function may be summed over all training examples

For example, with 2 parameters:
$w_0 \leftarrow w_0 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j))$; and
$w_1 \leftarrow w_1 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j)) * x_j$
… for all w_i's

where $h_{\underline{w}}(x_j)$ is the predicted value for y

Multivariate Linear Regression

Loss function may be summed over all training examples

For example, with 2 parameters:

$w_0 \leftarrow w_0 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j))$; and

$w_1 \leftarrow w_1 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j)) * x_j$

….

Above update procedure is called *batch-gradient descent:*

Update each w_i going over i's

For all training samples

……..

Stochastic-gradient descent:

For each training example *j*

update all w_is

. . . .

Typically, one uses a mix of the two: e.g., a fixed batch size

Multivariate Linear Regression

Loss function may be summed over all training examples

For example, with 2 parameters for one variable data ($y_j = w_0 + w_1 * x_j$):

$w_0 \leftarrow \; w_0 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j))$; and

$w_1 \leftarrow \; w_1 - \alpha * \sum_j (y_j - h_{\underline{w}}(x_j)) * x_j$

Note, in multi-variate case, element of $\underline{x}$ is x_{ij}

$y = w_0 + w_1 x_1 + w_2 x_2 + \ldots$, for *i* running over variables or parameters

and,

j running over training examples

Update rule is Eq 18.6

$\underline{w}^* = \text{argmin}_{\underline{w}} \sum_j L_2(y_j, \underline{w}.\underline{x_j})$, *as in L_2-norm*

or,

$w_i \leftarrow \; w_i + \alpha * \sum_j (y_j - h_{wi}(x_j)) * x_j$, *h is the hypothesis or model-formula, e.g. ax+b*

Multivariate Linear Regression

Note: some dimensions may be irrelevant or of low importance:

$w_i \sim 0$ for some x_i

Attempt to eliminate *irrelevant* dimensions or dimensions with low w values:

use a penalty term in error function for "complexity"

$\text{Loss}(h_{\underline{w}}) = L_2(h_{\underline{w}}) + \lambda \sum_i |w_i|$

L_1-norm (absolute sum) is better for this second term on complexity of the model:

"sparse model": minimizes #of "dimensions" (Fig 18.14 p722)

sometimes called *Lasso* regrerssion

Problem III: LINEAR CLASSIFIER 18.6.3

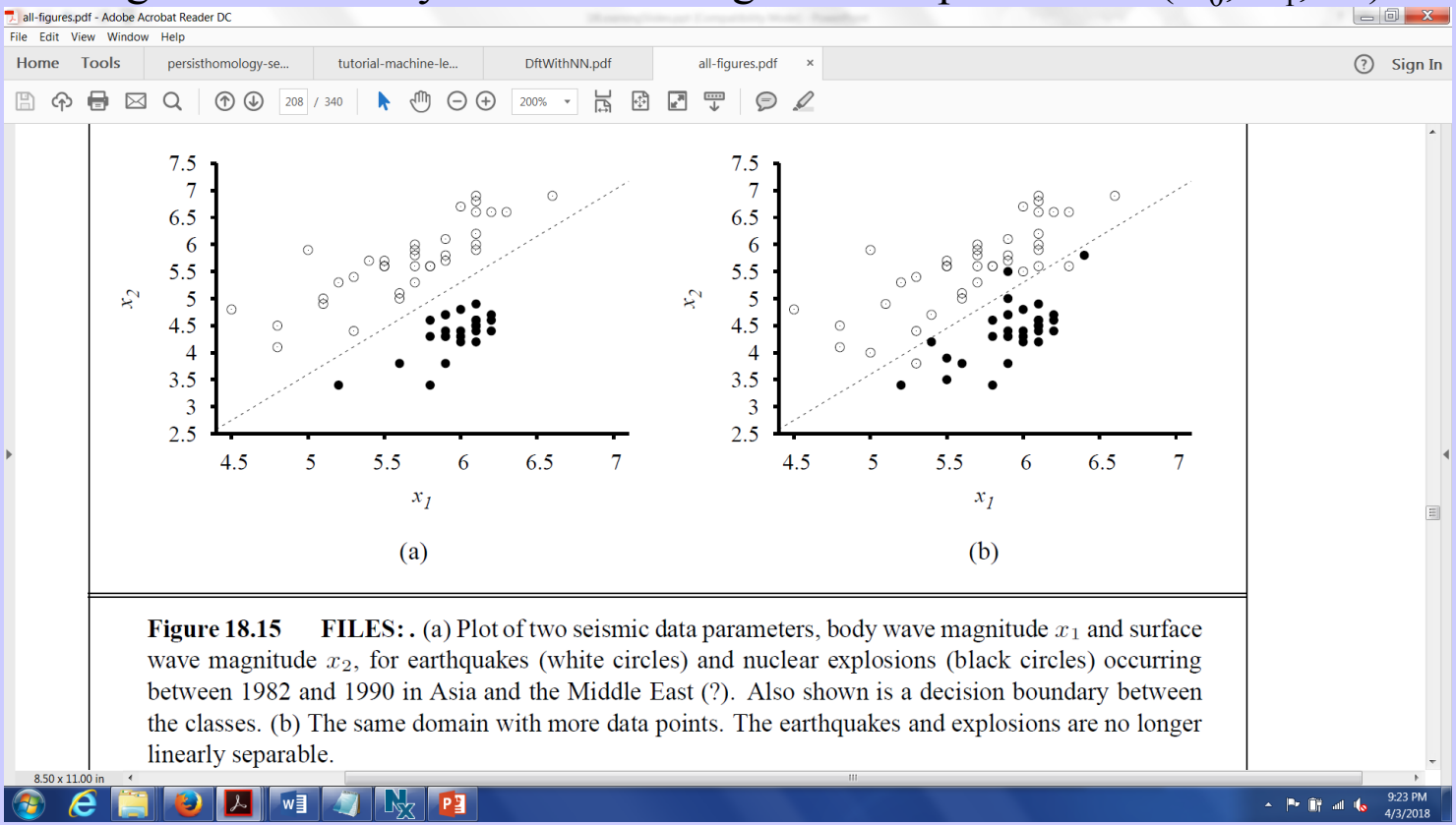
- Predicting *y* is the objective for regression, but classifiers predicts "type" or "class"
- Target function here is Boolean, y = 1 or 0 (as in Decision tree)
- The objective is to learn a Boolean function such that: $h_{\underline{w}}(x) = 1$ or 0:
  data point *x* is *in* the class or *not*
- Training problem:
  set of ($\underline{x}$, y) is given, *x* are data points and now, *y* =1 or 0,
  find $h_{\underline{w}}(x)$ that models *y*
- Test / purpose-of-learning / inference:
  a data point *x* is given, predict if it is in the class or not (compute $h_{\underline{w}}(x)$)

LINEAR CLASSIFIER 18.6.3

- No longer $h_{\underline{w}}(x)$ is the line expected to pass through (or close to) the data samples as in regression,
  but to separate or *classify* them into two sides of the line – in class or out of class

- Model: $w_0 + w_1x_1 + w_2x_2 + \ldots \geq 0$, and $h_{\underline{w}}(x) = 1$ if so, $=0$ otherwise

- Finding the line is very similar as in regression: optimize for $(w_0, w_1, \ldots)$



LINEAR CLASSIFIER 18.6.3

- Rewrite the model: $(w_0, w_1, w_2, ...)^T * (x_0, x_1, x_2, ...) \geq 0$, a vector product where (…) is a column vector, $(.)^T$ stands for matrix transpose, and $x_0 = 1$

- Consider two vectors, $w = (w_0, w_1, w_2, ...)^T$ and $x = (x_0, x_1, x_2, ...)^T$
- $h_{\underline{w}}(\underline{x}) = 1$ when $(\underline{w}.\underline{x}) \geq 0$, otherwise $h_{\underline{w}}(\underline{x}) = 0$

- Gradient descent (for linear separator) works as before. It is called:
- Perceptron Learning rule:
- $w_i \leftarrow w_i + \alpha*(y - h_{\underline{w}}(\underline{x})) * x_i, \quad 0 \leq i \leq n$, updates from iteration to iteration

- One can do Batch gradient descent (for-each *w*, inside for-each *datum-loop*) or
- Stochastic gradient descent (for-each datum, inside for-each w) here

- Note: *y* here is also Boolean: 1 or 0 in the "training" set

LINEAR CLASSIFIER 18.6.3

- Training:
- (1) $\underline{w}$ stays same for correct prediction y= $h_{\underline{w}}(\underline{x})$,
- (2) False negative: y=1, but $h_{\underline{w}}(\underline{x})$ =0, increase w_i for each positive?(x_i), decrease otherwise
- (3) False positive: y=0, but $h_{\underline{w}}(\underline{x})$ =1, decrease w_i for each positive?(x_i), increase otherwise

LINEAR CLASSIFIER 18.6.3

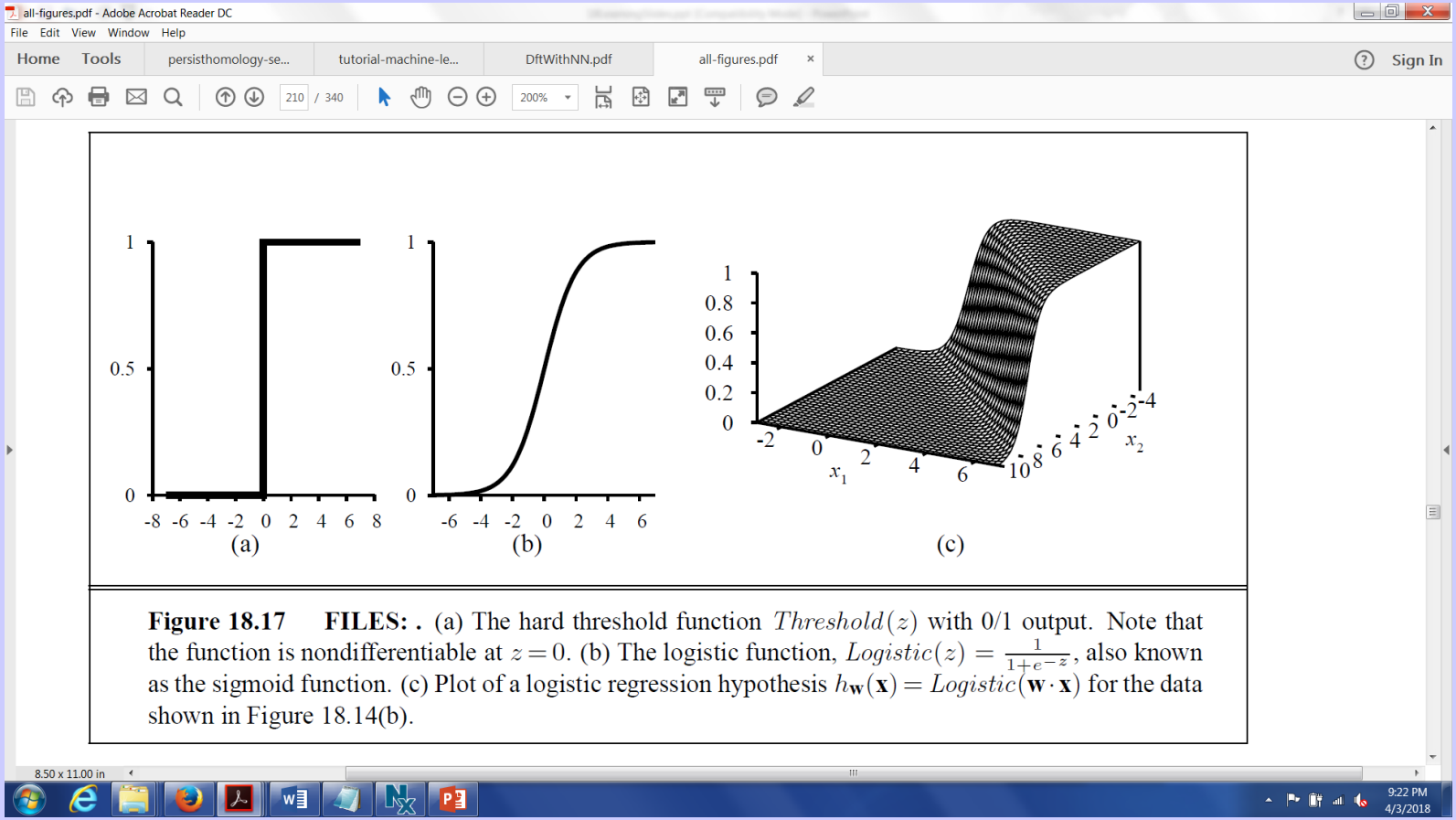


Figure 18.17 **FILES: .** (a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is nondifferentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 18.14(b).

- Logistic regression:
  use sigmoid $h_{\underline{w}}(\underline{x})$ rather than Boolean function (step function) as above
  $h_{\underline{w}}(\underline{x}) = \text{Logistic}(\underline{w}.\underline{x}) = 1 / [1 + e^{-\underline{w}.\underline{x}}]$

LINEAR CLASSIFIER 18.6.3

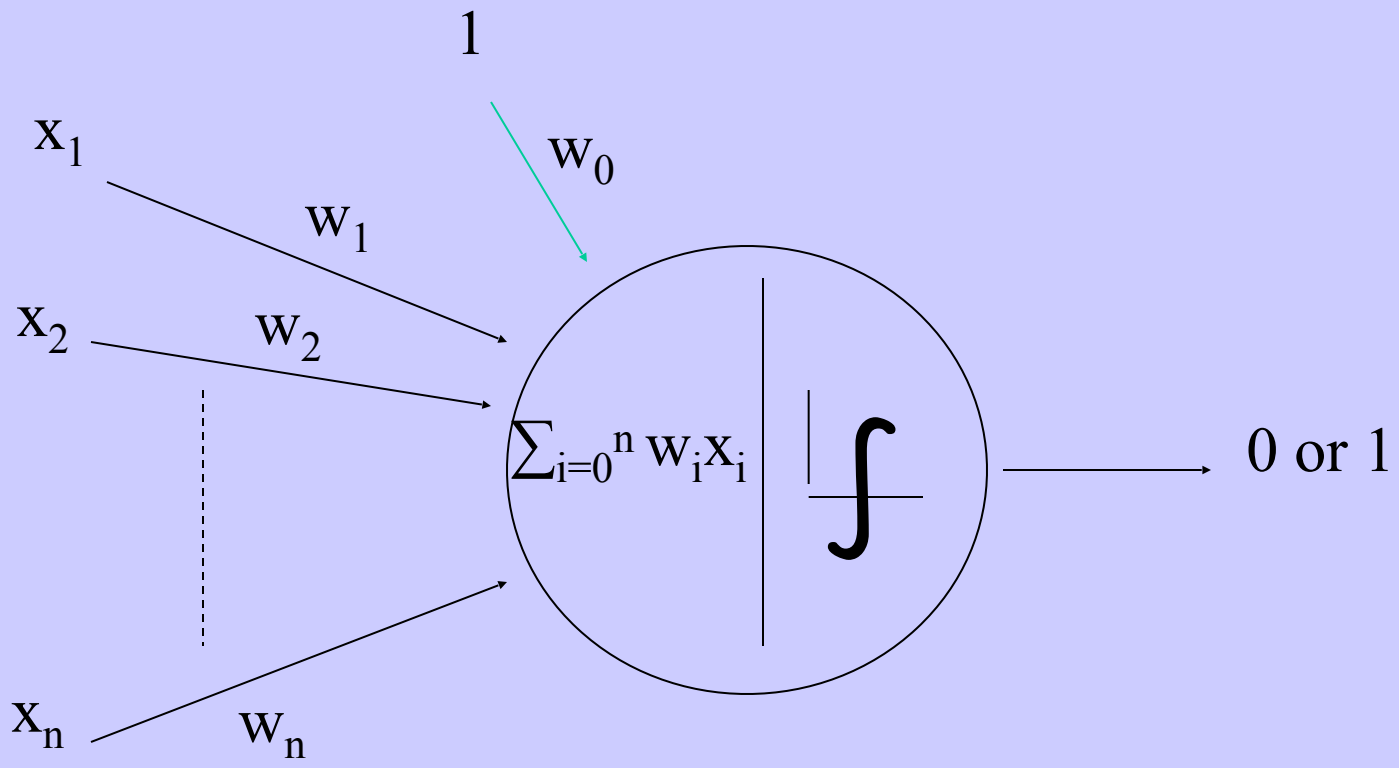
- Logistic regression:
  use sigmoid $h_{\underline{w}}(\underline{x})$ rather than Boolean function (step function) as above
  $h_{\underline{w}}(\underline{x}) = \text{Logistic}(\underline{w}.\underline{x}) = 1 / [1 + e^{-\underline{w}.\underline{x}}]$

- Update rule with above logistic regression model: Eq 18.8 p727
  $w_i \leftarrow w_i + \alpha*(y - h_{\underline{w}}(\underline{x})) * h_{\underline{w}}(\underline{x}) * (1- h_{\underline{w}}(\underline{x})) * x_i$

- Continuous valued $h_{\underline{w}}(\underline{x})$ may be interpreted as probability of being in the class

Problem IV: ARTIFICIAL NEURAL NETWORK Ch 18.7

- Single Perceptron, only a linear classifier, a neuron in the network



ARTIFICIAL NEURAL NETWORK Ch 18.7

- A single layer perceptron network cannot "learn" *xor* function or Boolean sum,
- Fig 18.21 p 730

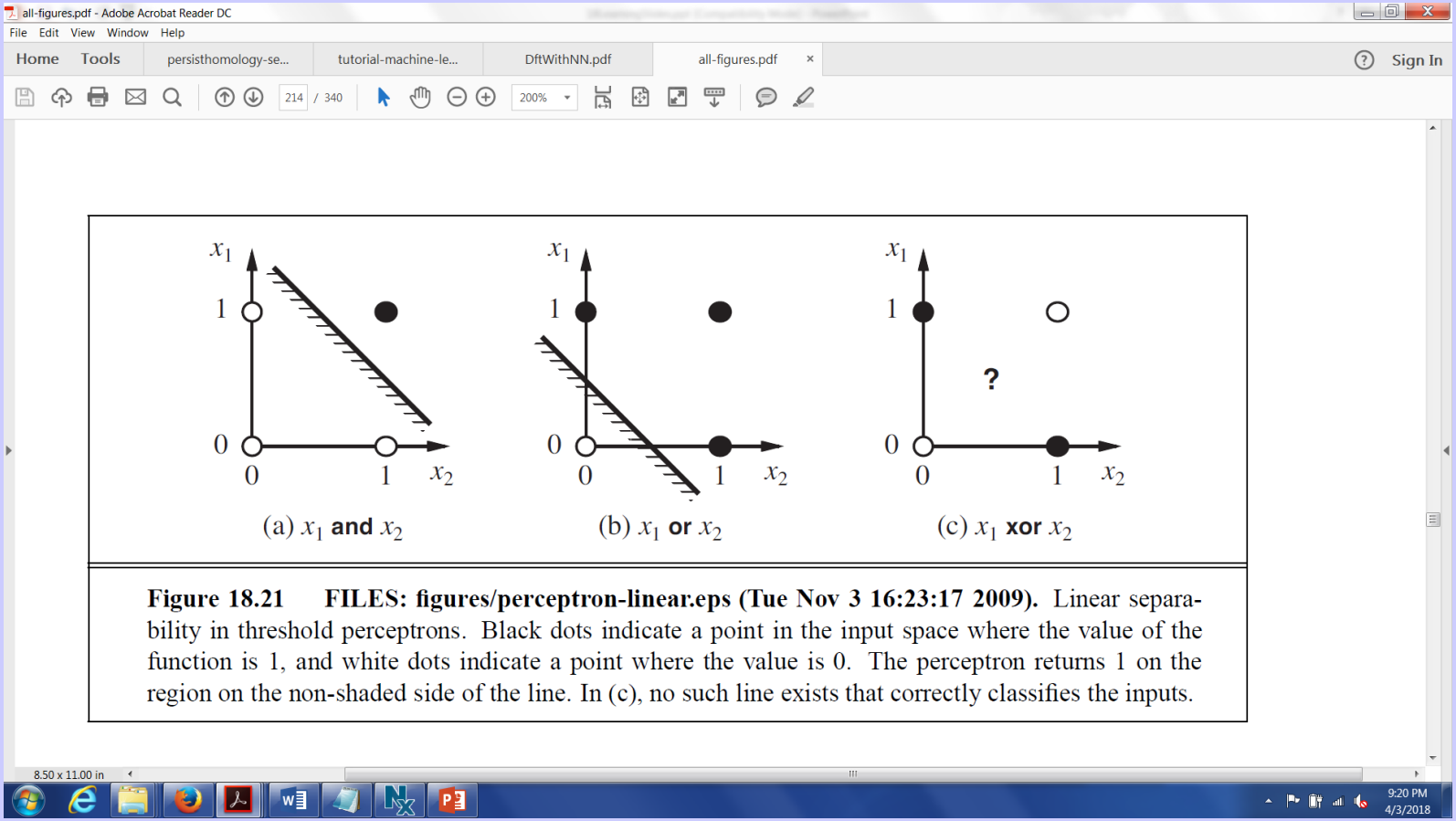


Figure 18.21 FILES: figures/perceptron-linear.eps (Tue Nov 3 16:23:17 2009). Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

ARTIFICIAL NEURAL NETWORK Ch 18.7

- Layers of perceptrons: *Input* → *Hidden* → *Hidden* → ... → *Output* classifying layer
- Two essential components: Architecture, and Weights updating algorithm

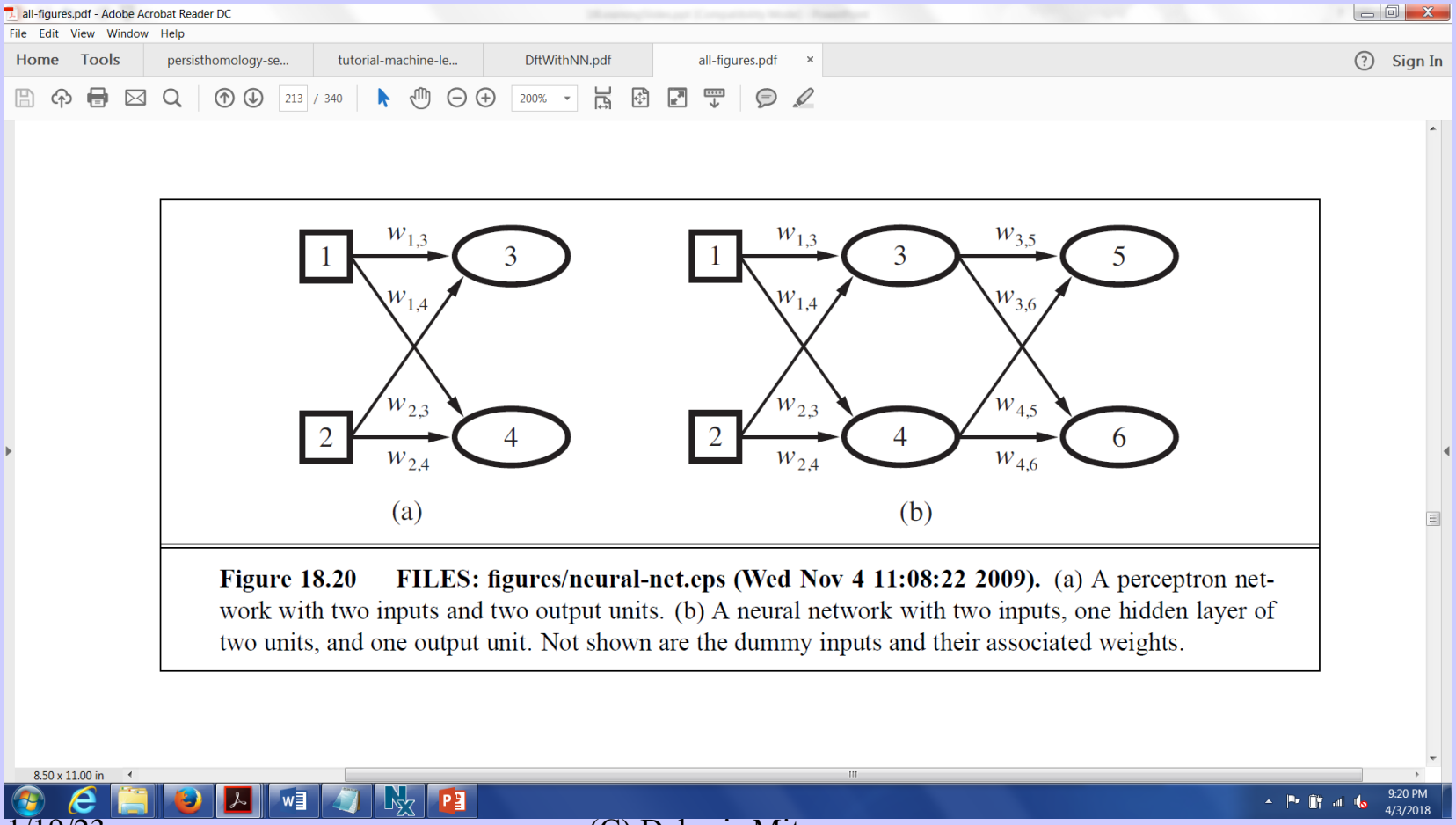


Figure 18.20 **FILES: figures/neural-net.eps (Wed Nov 4 11:08:22 2009).** (a) A perceptron network with two inputs and two output units. (b) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights.

ARTIFICIAL NEURAL NETWORK Ch 18.7

- Perceptron output fed to multiple other perceptrons, Fig 18.19 p728
- Different types of $h_{\underline{w}}(\underline{x})$ may be used, as *activation function*

ARTIFICIAL NEURAL NETWORK Ch 18.7

- A single layer perceptron network cannot "learn" *xor* function or Boolean sum
- Multi-layer Feed Forward Network:
- Multiple layers can coordinate to create complex multi-linear classification space,
- Fig 18.23 p732

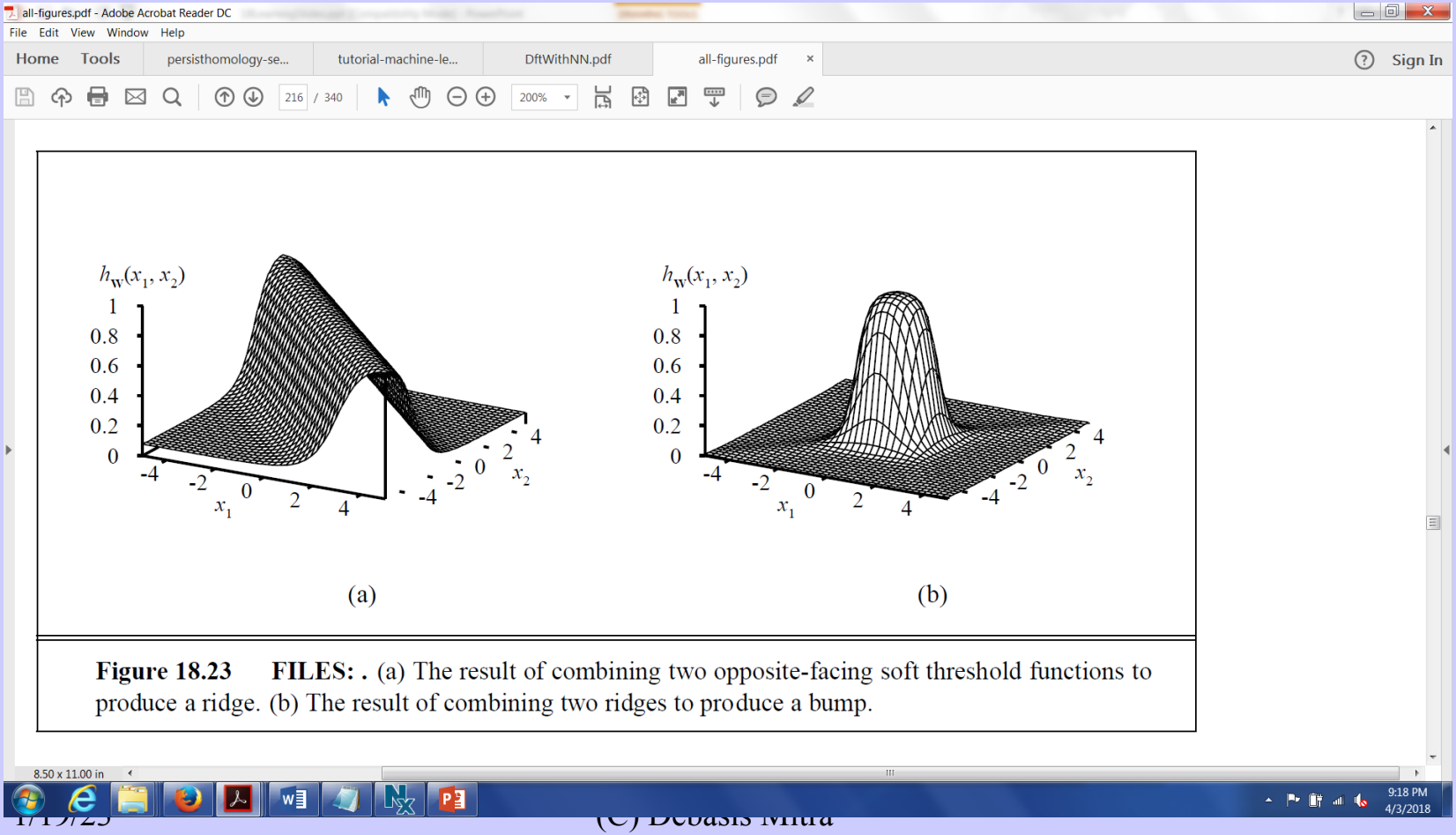


Figure 18.23 **FILES: .** (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

ARTIFICIAL NEURAL NETWORK Ch 18.7

- Types of architectures:
- *Feed-forward Network*: simple Directed Acyclic Graph
- *Recurrent Network*: feedback loop

ARTIFICIAL NEURAL NETWORK Ch 18.7.4

- Back-propagate, draw how error propagates backward
- get total error *del_E*, weighted distribution over each backward nodes,
- each node now knows its "errors“ or *E*’s, propagate that error recursively backward
- all the way through input layer,
- update weights or *w*’s

ARTIFICIAL NEURAL NETWORK Ch 18.7.4

- Total loss function at the output layer, *k* neurons:
  $Loss(\underline{\mathbf{w}}) = \sum_k (y_k - h_{\underline{\mathbf{w}}})^2 = \sum_k (y_k - a_k)^2$, a_k being the output produced

- This is to be minimized

- Gradient of this loss function is to iteratively lower:
  $\sum_k$ del/del_$\underline{w}$ $(y_k - a_k)^2$

- Weight updates:
  $w_{ij} \leftarrow w_{ij} + \alpha * a_j * Del_j$

- a_j is output of the neuron, Del_j weighted modified error incorporating
  the activation function's effect (see Eq 18.8)

- Error propagation

$Del_j = g'(in_j) \sum_k (w_{jk} Del_k)$, where g'() derivative of activation, *k* is over next layer neurons (previous layer in backward direction)

- An iteration of backprop learning:
  Propagate errors then update weight, layer by layer backwards:

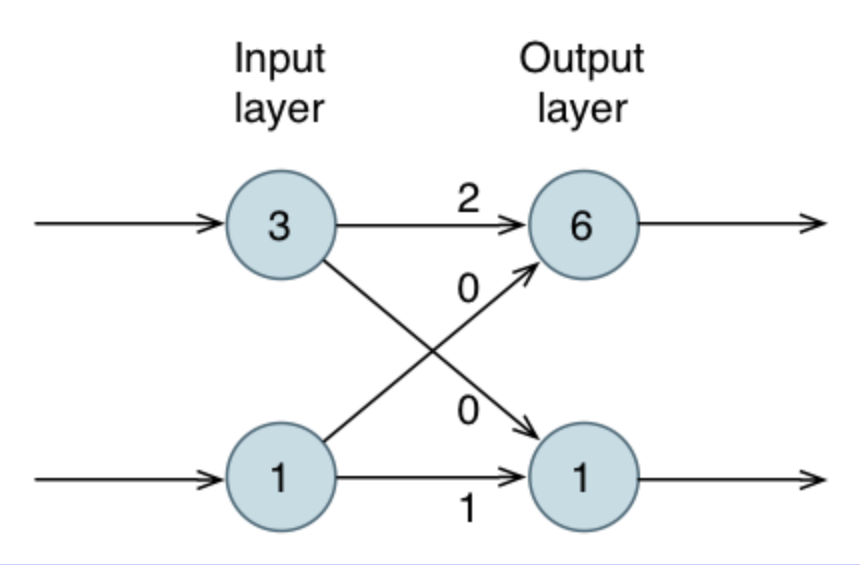
GENERATIVE NEURAL NETWORK

Not only classification…

Transformations: say, (x,y) goes to (2x,y) – a linear transformation that we want to learn

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 1 \end{bmatrix}$$

Neural Net:



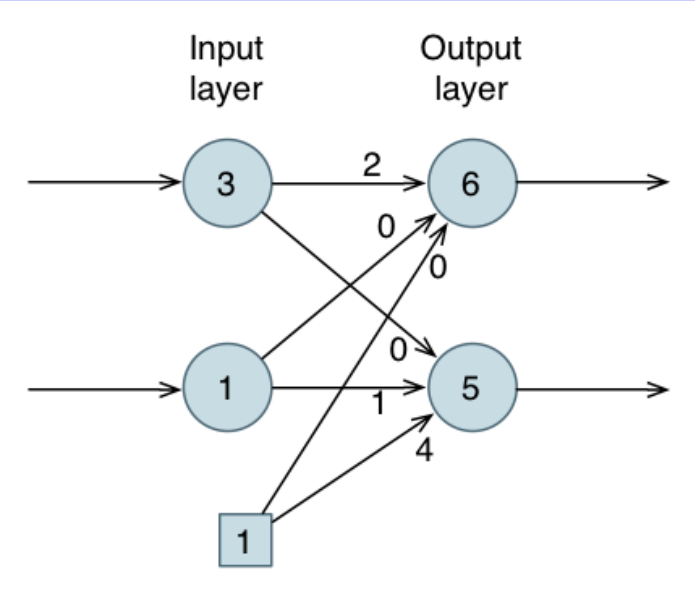
https://medium.com/wwblog/transformation-in-neural-networks-cdf74cbd8da8

GENERATIVE NEURAL NETWORK

- Add translations (Affine transformation):

$$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}$$

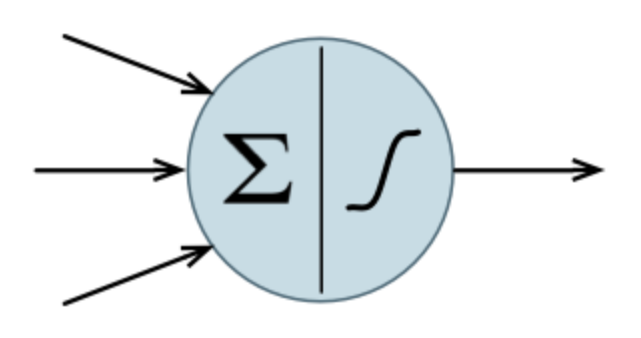
Neural Net:



https://medium.com/wwblog/transformation-in-neural-networks-cdf74cbd8da8

GENERATIVE NEURAL NETWORK

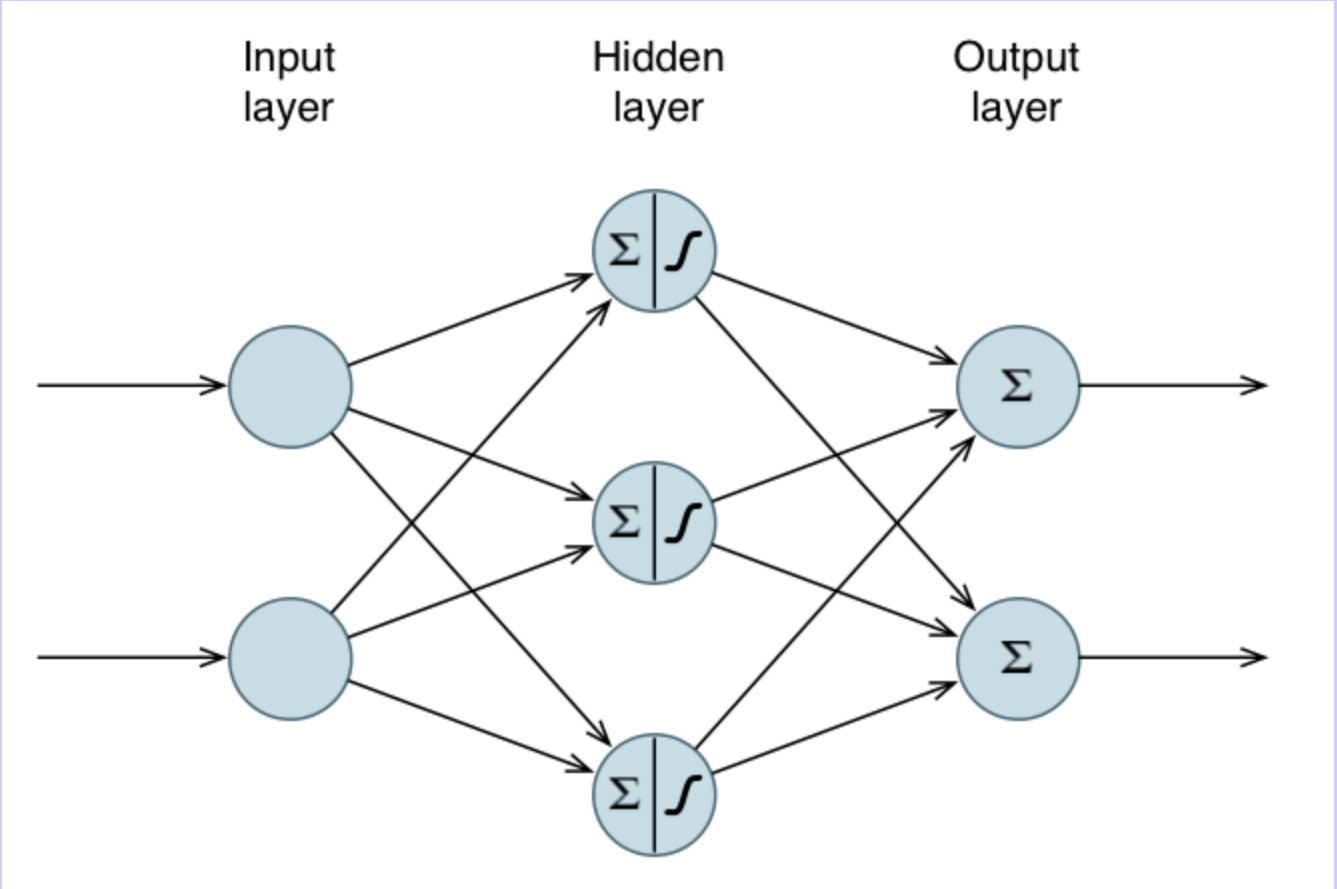
- For non-linear transformations use non-linear activation function:



https://medium.com/wwblog/transformation-in-neural-networks-cdf74cbd8da8

GENERATIVE NEURAL NETWORK

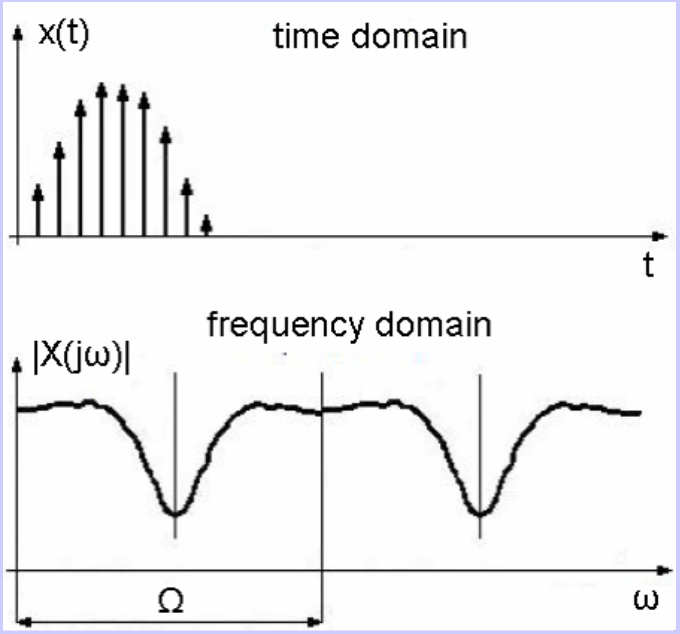
- Multi-layer generative network for non-linear transformation:

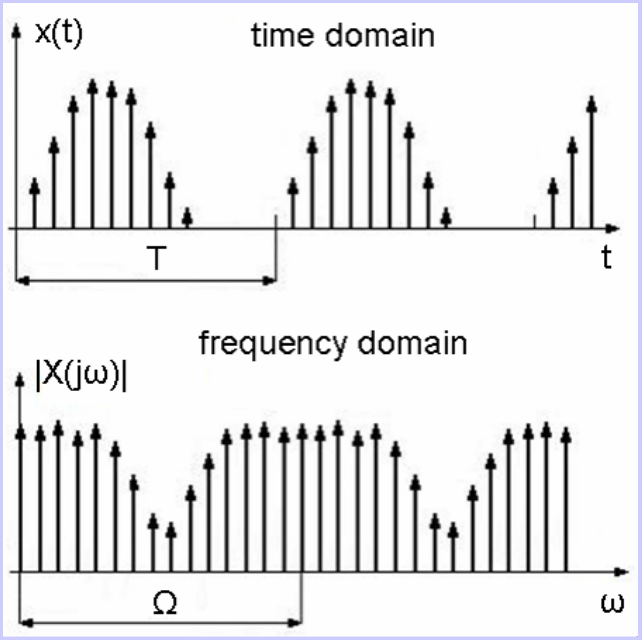


https://medium.com/wwblog/transformation-in-neural-networks-cdf74cbd8da8

GENERATIVE NEURAL NETWORK

- Fourier Transform (a linear transform):

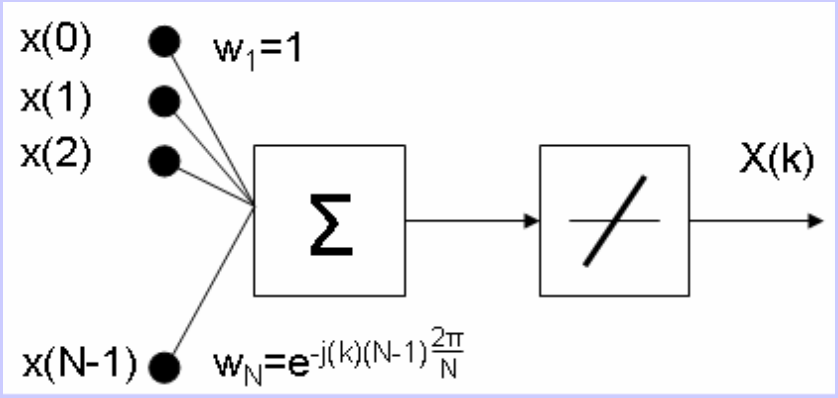


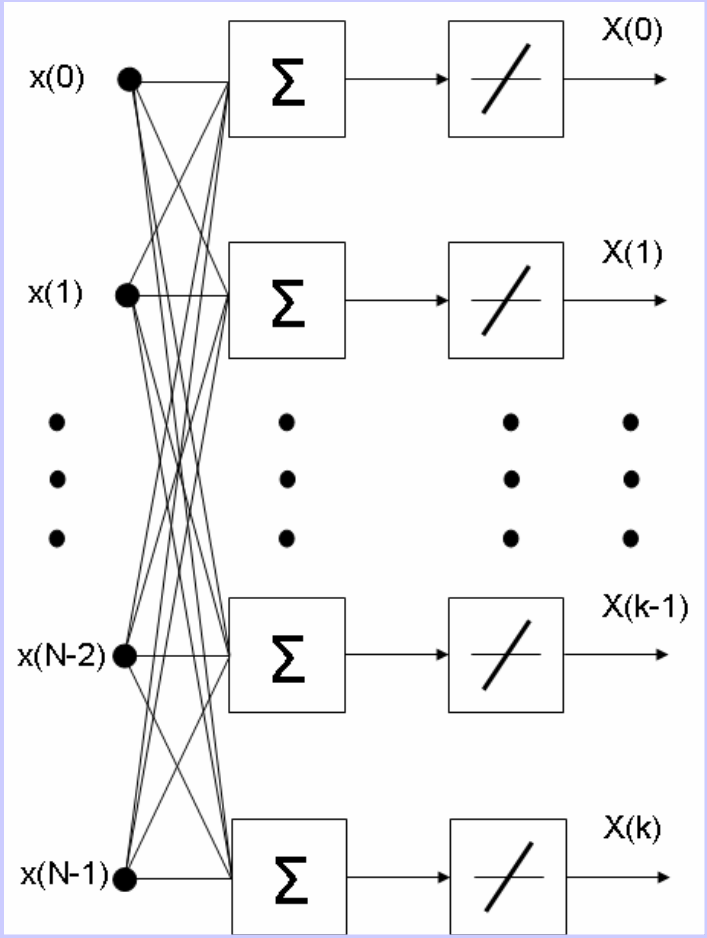


Discrete Fourier Transform Computation Using Neural Networks,
Rosemarie Velik, *Vienna University of Technology*

GENERATIVE NEURAL NETWORK

- NN for Fourier Transform:

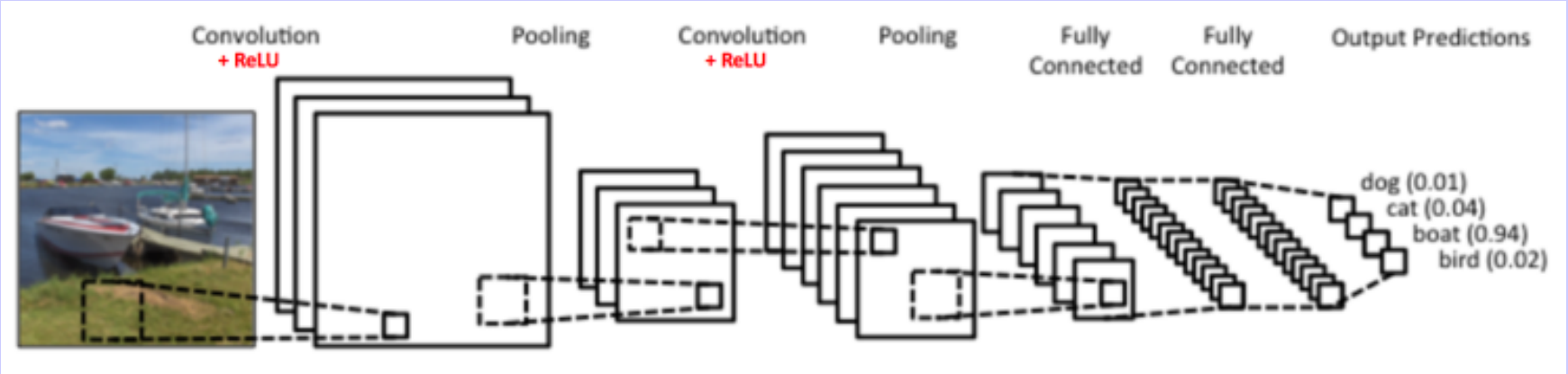




- Learning should attain weight values from training

CONVOLUTIONAL NEURAL NETWORK

- Running window weighted mean = convolution
- Weights are learned

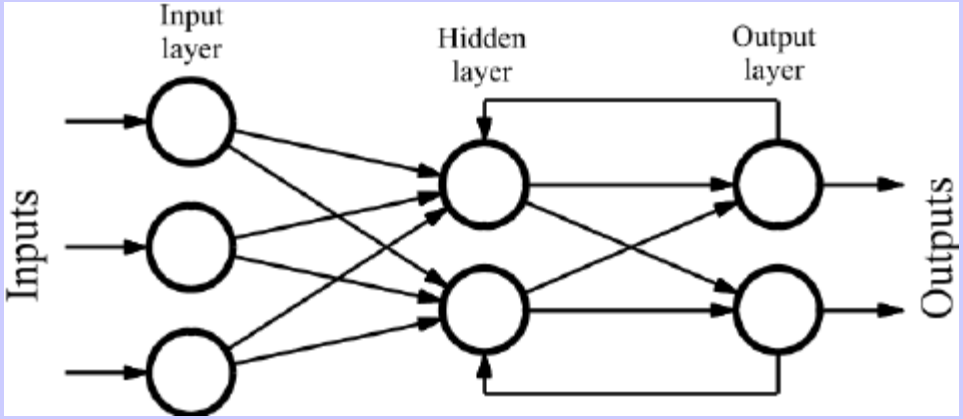


- Pooling: Pick up a value (e,g., max) from a window – reduce size

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

RECURRENT NEURAL NETWORK

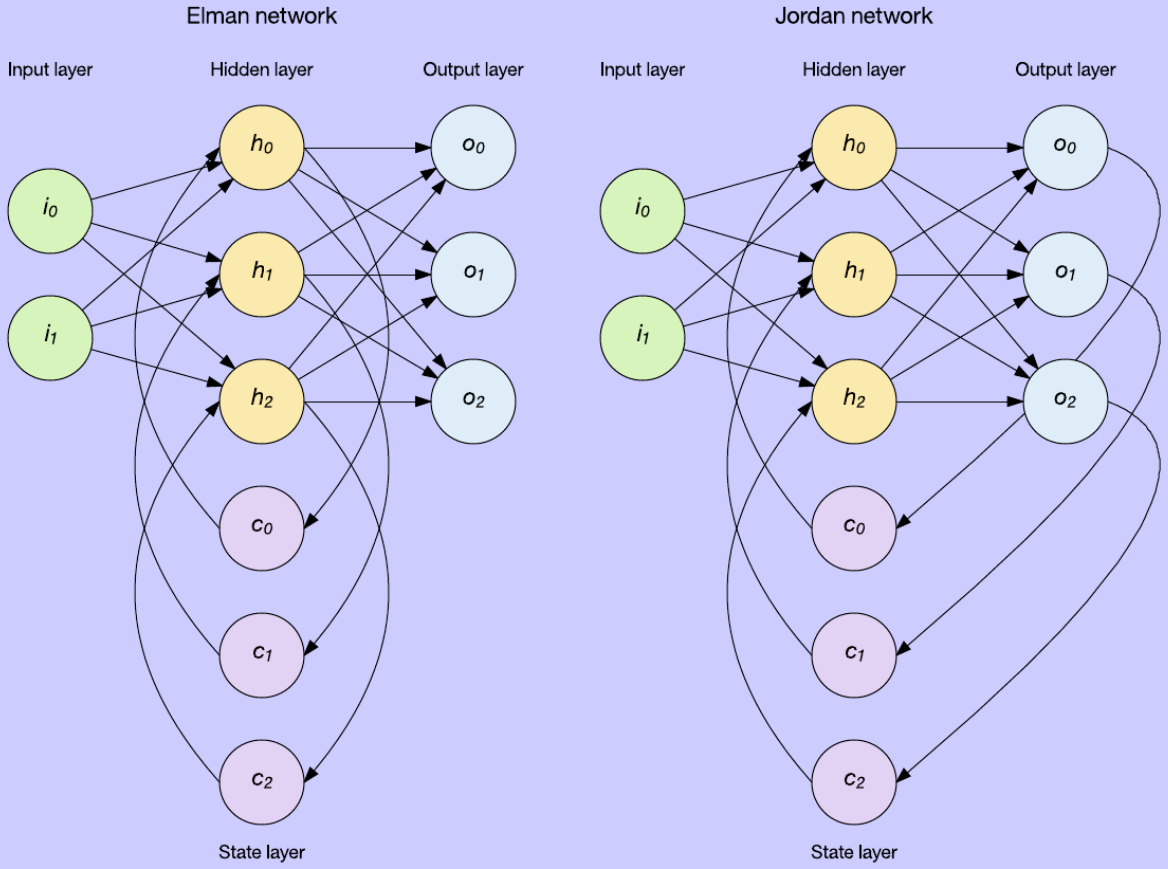
- Feed back loops are provided in the network
- Independent from error propagation (backprop learning, weight update) algorithm:



https://www.researchgate.net/figure/Graph-of-a-recurrent-neural-network_fig3_234055140

RECURRENT NEURAL NETWORK

- An exotic recurrent network:



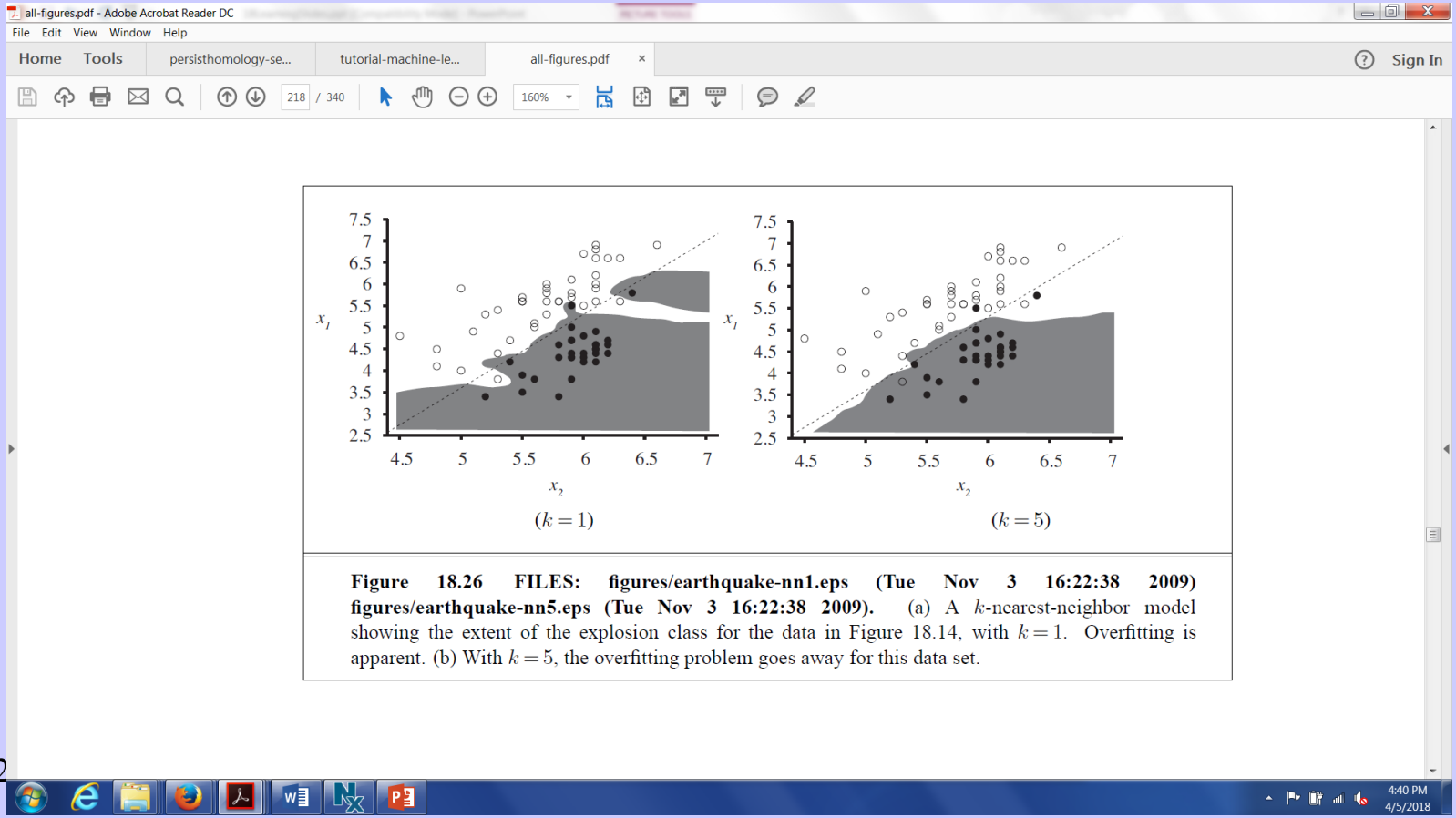
NON-PARAMETRIC *MODELS Ch 18.8*

- Parametric learning: $h_{\underline{w}}$ has $\underline{w}$ as parameters
- Non-parametric = no "global curve fitting"
- Query-time processing, minimal or no training
- *Simplest*: Table look up (Problem V)
  For a query find a closest data point and return
- We still need a sense of "distance" between data points, e.g., L^p-norm
  $L_p(x_j, x_q) = (\sum_{ki} (x_{ji} - x_{qi})^p)^{1/p}$, *i* runs over dimension of space,
  $\underline{x_j}$ and $\underline{x_q}$ are two points in that space

NON-PARAMETRIC MODELS Ch 18.8

- *K-nearest neighbor* look up (*kNN*): find k nearest neighboring example data instead of one,
- and vote by their attribute values (Pure counting for Boolean attributes)
- *k* is typically odd integer for this reason *(Problem VI)*

- *Fig. 18.26 p738, shows the "query-space",*
- *runs query for every point in the 2D space to check what the prediction will return for that point*
- *Gray areas indicate prediction=dark circle; and white areas indicate prediction=open circle*



NON-PARAMETRIC MODELS Ch 18.8

- (*Problem VII*) A different version of *kNN*: *fix a distance value d* and vote by all data points within *d*

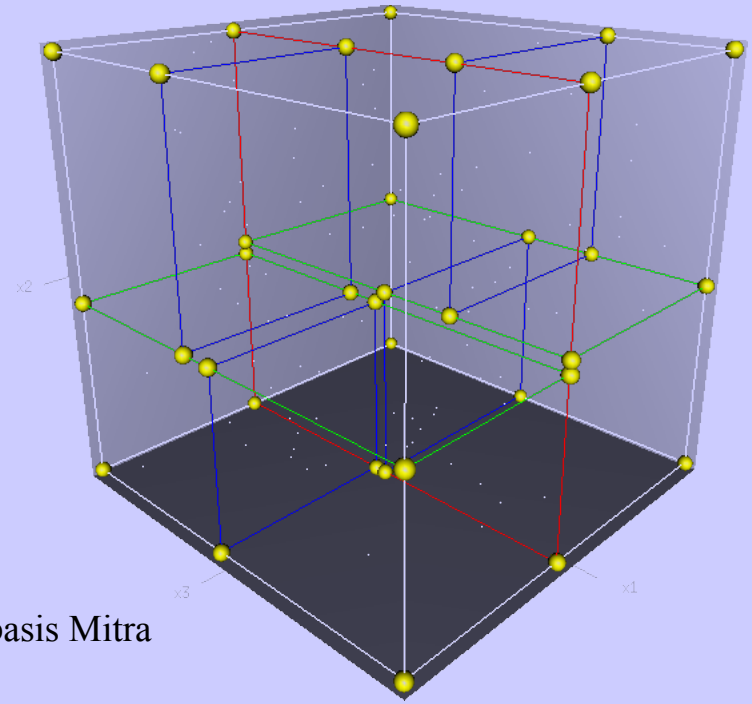
- *Advantage*: faster search, conventional *kNN* may need very expensive data organization
  (Note: this is a search problem, before query gets answered)

- *Disadvantage*: there may be too few data points within range *d*,
  e.g. zero data point or no datum

- *Curse of dimensionality:* number of dimensions (attributes) >> number of data
  - *Sparsely distributed data points*
  - *Search is slow*

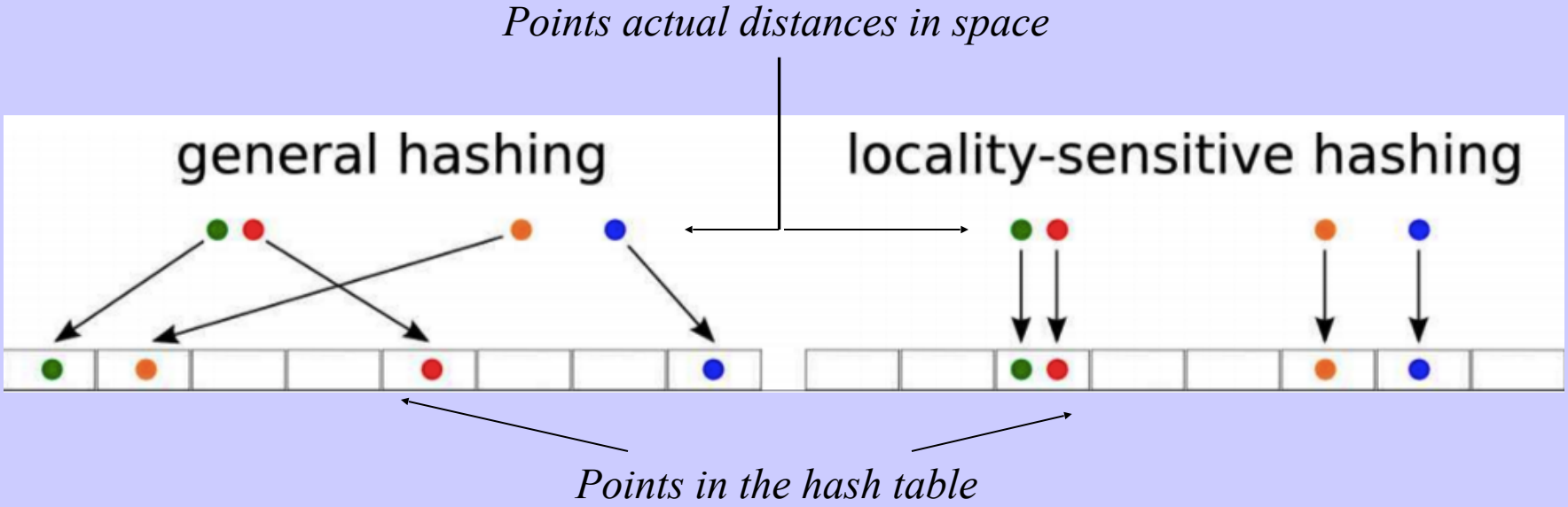
NON-PARAMETRIC MODELS Ch 18.8

- An efficient data organization: k-d tree
- *k* is dimension here
- Balanced binary search tree over *k*-dimension, with median (on each dimension) as the splitting boundary



NON-PARAMETRIC MODELS Ch 18.8

- Another efficient data organization: *LSH* or *locality sensitive hashing* (Problem VIII)
- Hashing typically distributes data randomly, but we want nearer points together in memory
- A few concepts are combined:
  - Approximate near-neighbors: find points that have "high" probability to be within distance *cr* , *radius r* is fixed, *c* and *high probability* are parameters
  - Two close points have always close projections on any dimension(s), but the reverse is unlikely to be true
  - Create multiple hash functions on multiple subset of dimensions (random),
    e.g., $x_1x_3x_4$, $x_5x_2x_9$,…
  - Retrieve all points close to the query point in any of the hash function (union of points with same hash value in each hash function)
  - Do full *kNN* search over those points only

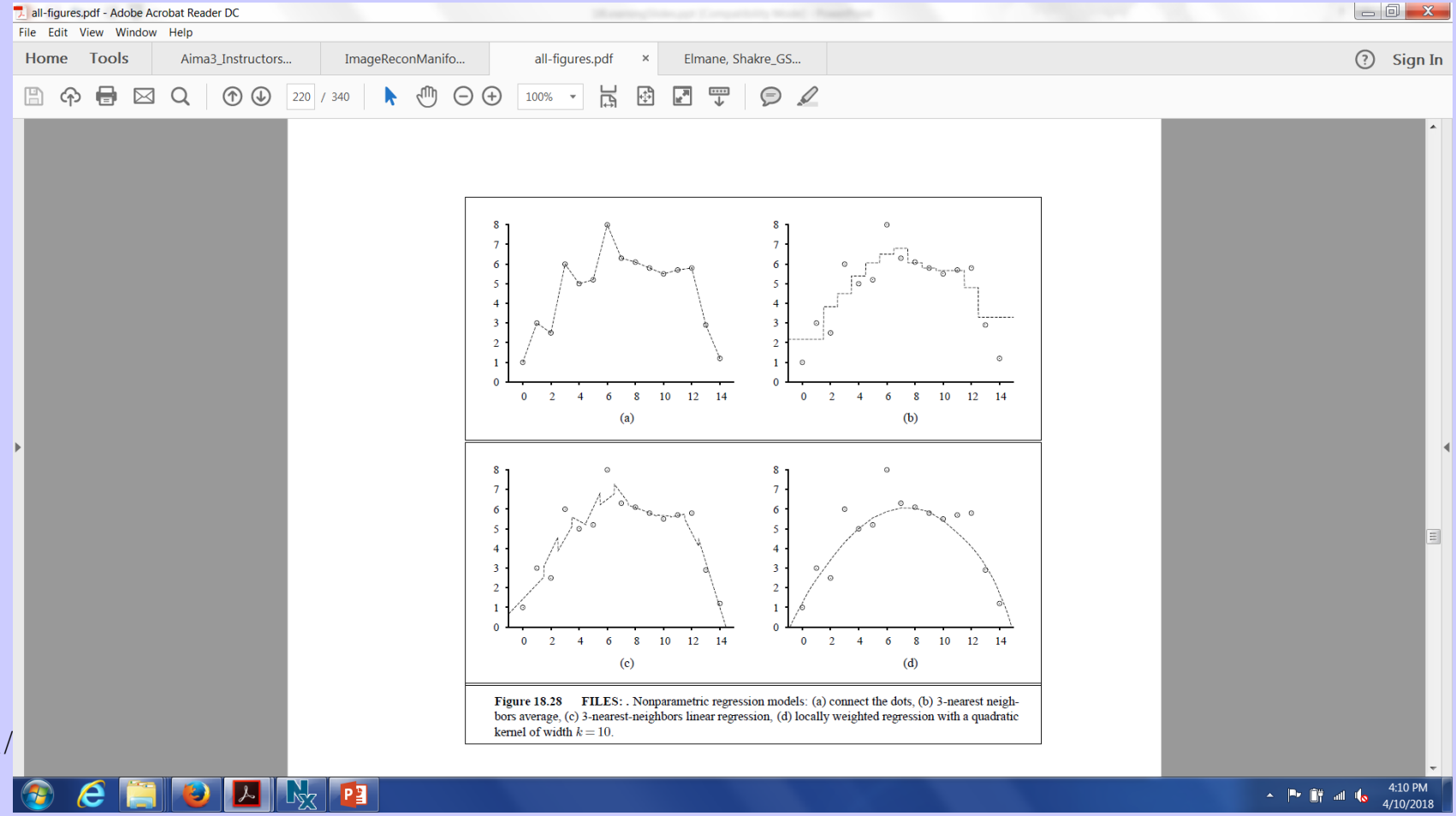


https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

NON-PARAMETRIC MODELS Ch 18.8

- *Back to regression*: NON-PARAMETRIC REGRESSION (Problem IX)

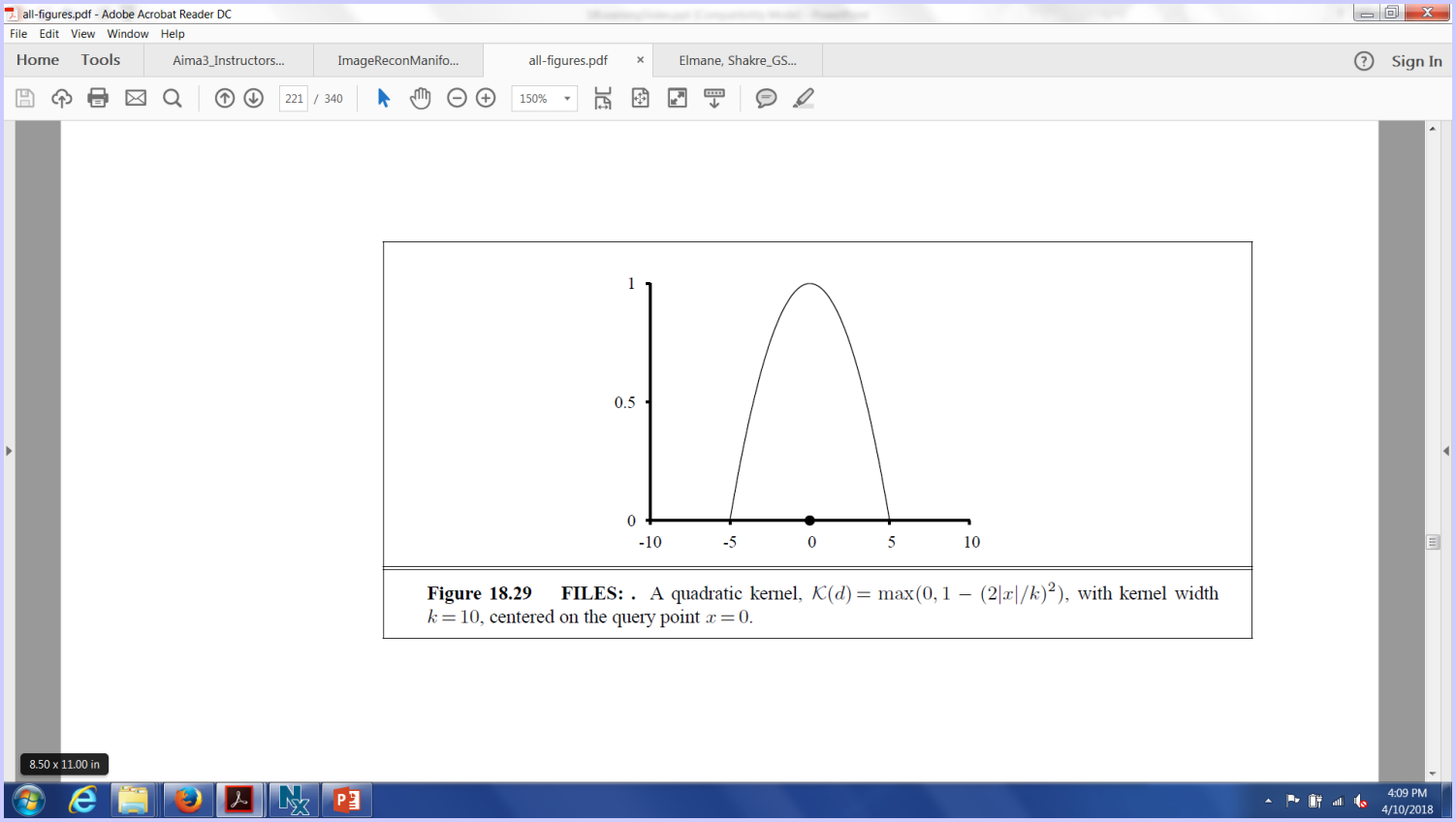
- *Philosophy:* Only *near-query data point* should influence regression result more than distance points
- Find *k*-nearest neighbors and perform regression on them
- Fig 18.28 p742: k=1, 3-average, 3 linear-regression, 10 with quadratic *Kernel*



1/

NON-PARAMETRIC MODELS Ch 18.8

- Find k-nearest neighbors and perform regression on them
- *Kernel-regression*: Locally weighted regression
  – provide more weights to closer points to the query



- Weights may be computed *("learned")*, given a parametrized function

NON-PARAMETRIC MODELS Ch 18.8

- *Kernel-regression*: Locally weighted regression
- Weights may be computed, given a parametrized function

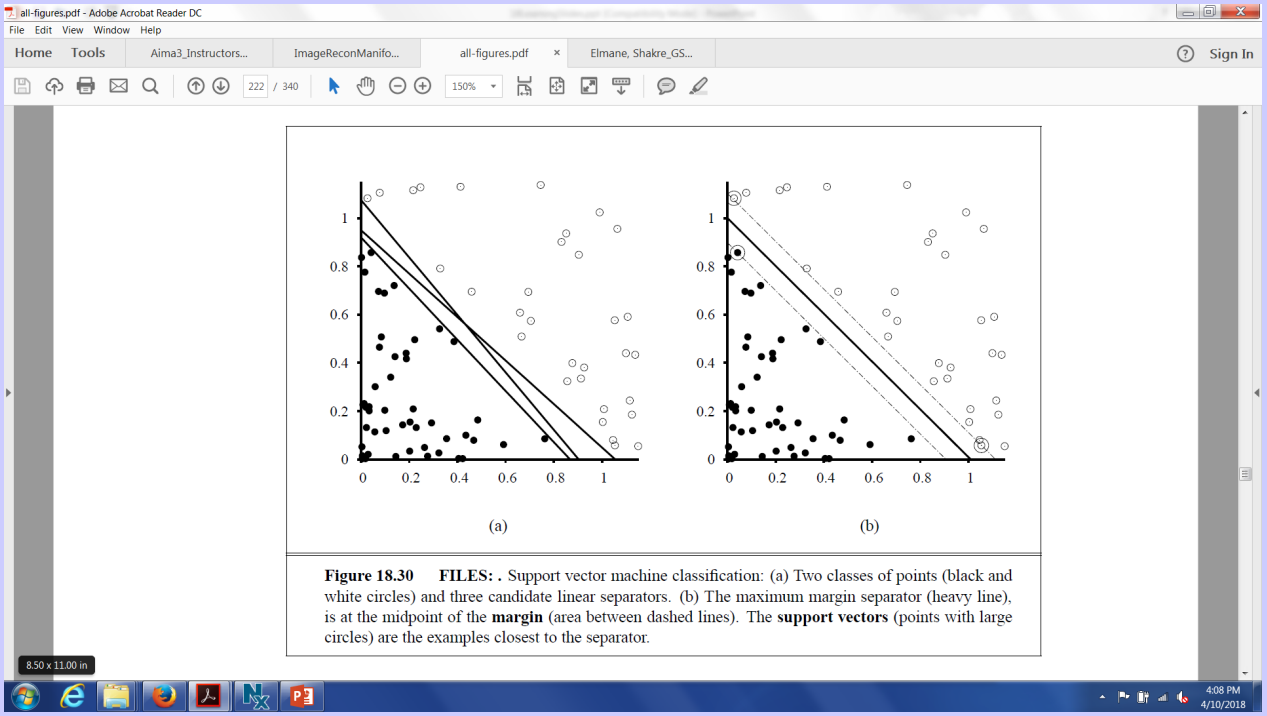
$\underline{w}^{\wedge} = \text{argmin}_w \sum_j K(\text{Distance}(\underline{x}_q, \underline{x}_j)) * (y_j - \underline{w}.\underline{x}_j)^2$

K is the kernel functional form with $\underline{w}$ as parameters,
x_q is the query point,
(remember) y_j is the actual output (*"training labels"*)

- Then the *regressed* output of query point $\underline{x}_q$ is, $h(\underline{x}_q) = \underline{w}^{\wedge} \cdot \underline{x}_q$

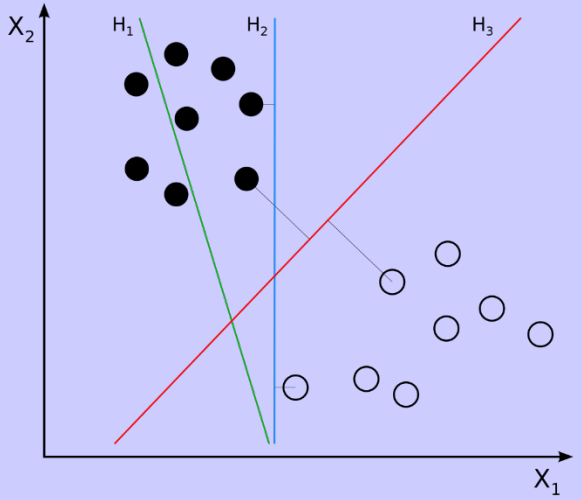
NON-PARAMETRIC MODELS Ch 18.9

- *SUPPORT VECTOR MACHINE* (SVM) – basics, *the best ML algorithm so far* (Problem X)
- A few concepts come together:
- Support vector: Data **points** separating the boundary between + and – labels
  (Classification or *decision boundary*)
- But with, *two parallel lines*, one passing through +ve support vectors, one through –ve ones
  The *gap* between these two *lines* that must be *maximized*, Fig 18.31 p747

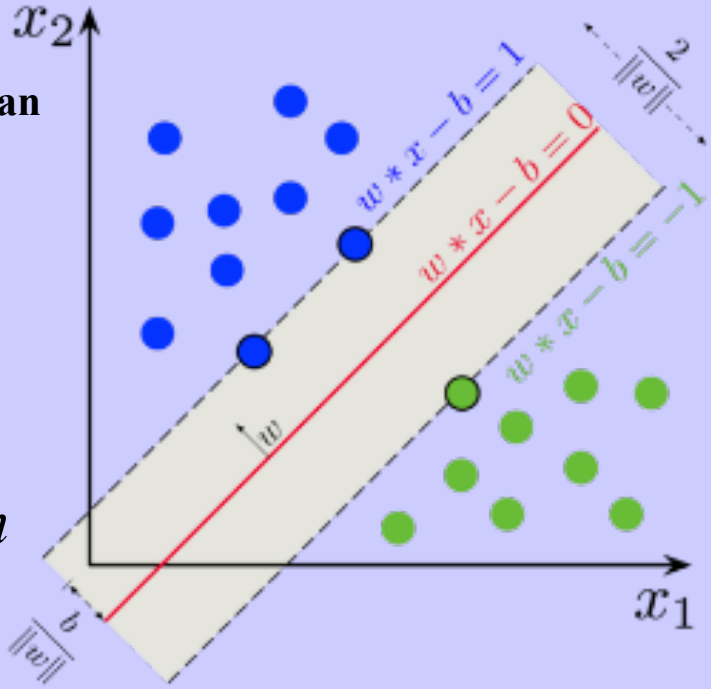


H_1 does not separate the classes.
H_2 does, but only with a small margin.
H_3 separates them with the ***maximal* margin.**

-- Wiki on SVM



Maximum-margin hyperplane and margins for an SVM trained with samples from two classes.
Samples on the margin are called the support vectors.



Note: expects clean, not noisy, separation

NON-PARAMETRIC MODELS Ch 18.9

- SUPPORT VECTOR MACHINE (SVM) – *basics*
- Concept-1. A few concepts come together, 1) Support vector
- 2) Non-linear space transformation to linearize decision plane: Kernelization
  - Resulting space may have more dimensions than the original space
- 3) Very fast primal-dual optimization

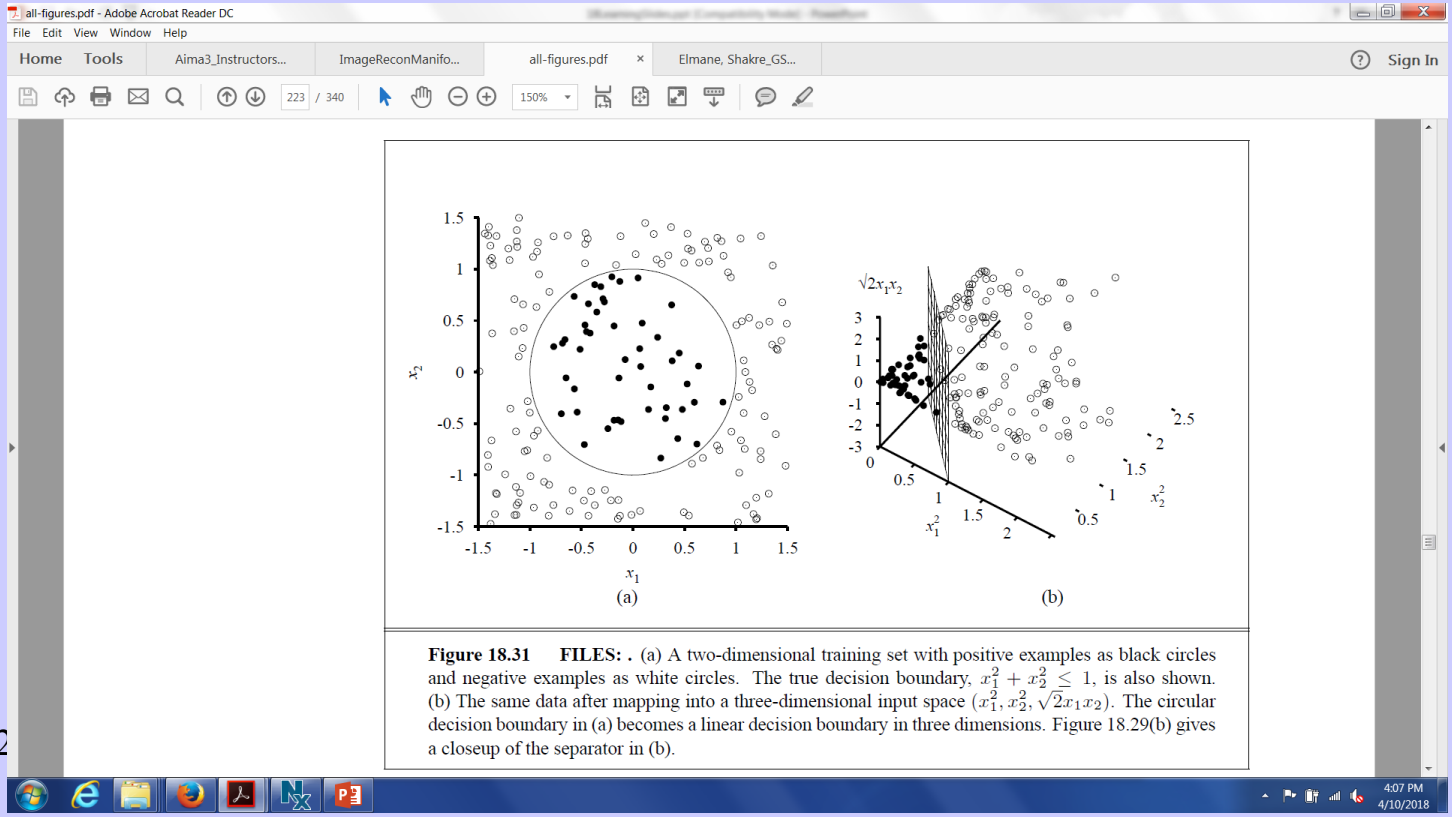


Figure 18.31 **FILES: .** (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 18.29(b) gives a closeup of the separator in (b).

NON-PARAMETRIC MODELS Ch 18.9

- SUPPORT VECTOR MACHINE (SVM) – basics
- Query on SVM runs very fast – like *kNN* algorithm
  - using only support vectors*, selected at training time*
- Stores *Only* Support Vectors – *huge space saving too*!

TYPES OF LEARNING

- Supervised (Regression or Classification)
- Unsupervised / Clustering:
  - Only data, no label to predict.
  - So, *group* or *cluster* data by their "proximity"
- Reinforcement Learning:
  - Occasional reward/penalty as the agent keeps behaving in real world (input data)
  - Online / Interactive / Robotics
- Semi-supervised learning:
  - Predict (hypothesis *h*) and include the prediction as training data (!!) if actual predicted value (*y*) was not available
  - Follows the "trajectory" of incoming data

UNSUPERVISED LEARNING / CLUSTERING

- No target output value to predict, i.e., **no label**, only a set of data
- Group them by their "proximity"
- Needs a distance function $d(x_1, x_2)$ between data x_1 and x_2

TYPES OF CLUSTERING

- Centroid models: for example, the *k-means* algorithm represents each cluster by a single mean vector.
- Connectivity models: for example, *hierarchical* clustering builds models based on distance connectivity or topology. *Mapper* algo creates maps of data space.
- Distribution models: clusters are modeled using statistical distributions, such as multivariate normal distributions used by the *expectation-maximization* algorithm.
- Density models: for example, DBSCAN and OPTICS defines clusters as connected dense regions in the data space. ToMato for *topological clustering* uses this.
- Subspace models: in *bi-clustering* (also known as co-clustering or two-mode-clustering), clusters are modeled with both cluster members and relevant attributes.
- Group models: some algorithms do not provide a refined model for their results and just provide the grouping information of data space.
- Graph-based models: a clique, that is, a subset of nodes in a graph such that every two nodes in the subset are connected by an edge that can be considered as a prototypical form of cluster.

- Neural models: the most well known unsupervised neural network is the self-organizing map (SOM) and these models can usually be characterized as similar to one or more of the above models: learns local manifolds in data space
- Topological data analysis (e.g., Mapper algorithm): visualize topology of data points on space

K-MEANS CLUSTERING
(Problem XII)

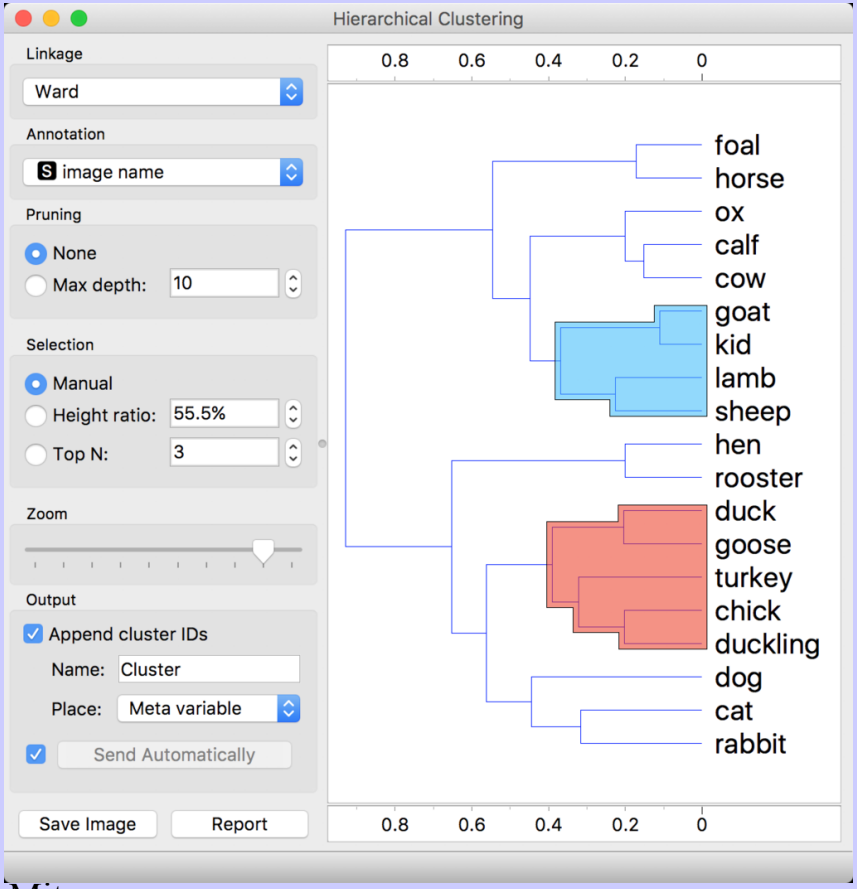
- Start with arbitrary *k* cluster-center points in data space

- Do two-steps while not converged:
- Group data points by their proximity to each of those *k* cluster-center points
- Find each group's mean (or median) and assign it as the new cluster-center

- https://en.wikipedia.org/wiki/K-means_clustering

- https://en.wikipedia.org/wiki/Hierarchical_clustering

- Two different ways to build:
  - Start with all points as one cluster and keep splitting *(top-down)*
  - Each point as a cluster and keep merging *(bottom-up)*



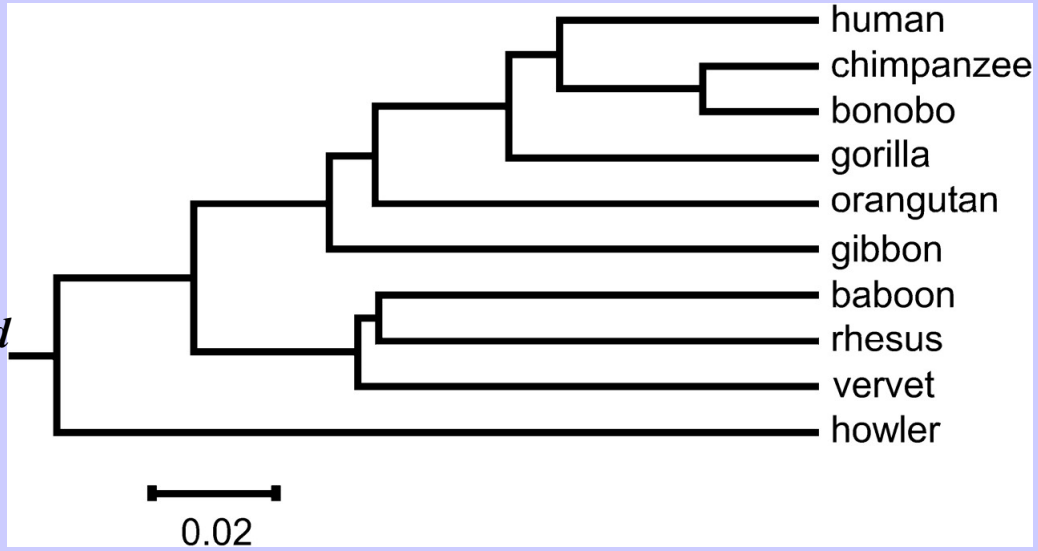
HIERARCHICAL CLUSTERING

- Needs a measure for inter-cluster distance, for splitting or merging
- UPGMA algorithm's (bottom up) inter cluster distance:

  $[1/|A|*|B|] * \sum_{x \in A} \sum_{x \in B} d(x,y)$

  where |A| is the size of cluster A, and so for B, and
  x and y are two points in clusters A and B, respectively,
  d(x,y) is the distance between those two points

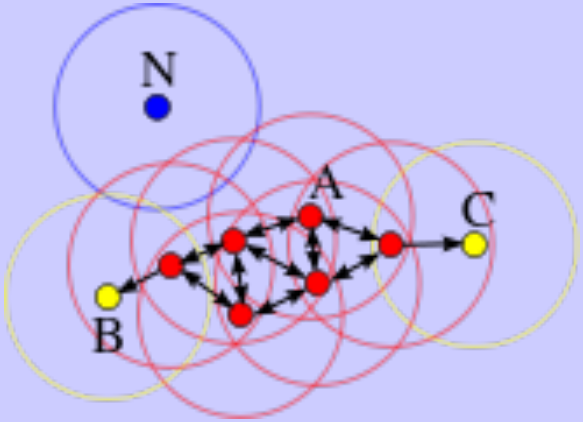
Genetic base pair distances translated to evolution time distance



- Given a set of points in some space, it groups together points that are closely packed together (points with many nearby neighbors), marking as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away).

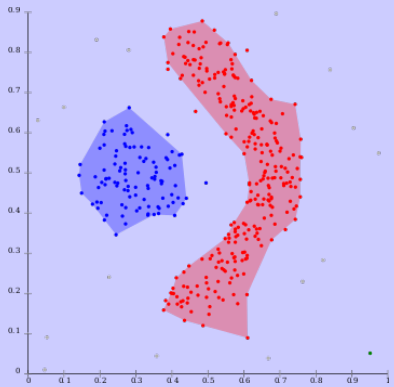
- https://en.wikipedia.org/wiki/DBSCAN

- *p* is a *core point* if at least *minPts* #points are within distance ε of it (including *p*). Those points are said to be *directly reachable* from *p*
- *q* is *directly reachable* from *p* if point *q* is within distance ε from point *p* where *p* is a core point.
- *q* is *reachable* from *p* if there is a path from *p* to *q*, via directly reachable points, with the possible exception of *q* itself.
- All points not reachable from any other point are *outliers*.
- Core points constitute a cluster core with reachable outliers as cluster edge



ADVANTAGES: DBSCAN

- No need for k as input (k = #clusters) as opposed to that in k-means clustering
- Arbitrarily shaped clusters.
  It can even find one cluster surrounded by a different cluster



- Understands noise, and is robust to outliers
- Requires two parameters *minPts* and ε, can be set by expert by pre-analyzing data
- It is mostly insensitive to the ordering of the points in the database.
  (However, points on an edge between two different clusters might swap cluster membership)

DISADVANTAGES: DBSCAN

- DBSCAN is non-deterministic: border points that are reachable from more than one cluster
- DBSCAN* is a variation that treats border points as noise, and not included in clusters
- Quality of DBSCAN depends on *minPts* and ε
- DBSCAN cannot cluster data sets well with large differences in densities, since the *minPts* and ε combination cannot then be chosen appropriately for all clusters
- If the data and scale are not well understood, choosing a meaningful ε can be difficult.