Machine Learning

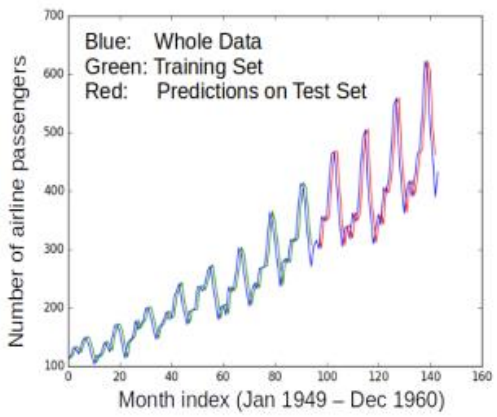
Aminur Hossain

Topics Covered in Machine Learning (ML)

- **ML Overview: What is ML?**
- **Why to Use ML?**
- **ML Taxonomy: Supervised, Unsupervised**
- **Data and Features**
- **Regression vs Classification**
- **Supervised learning by computing Distance**
  - **Learning with Prototype**
  - **Nearest Neighbors**
  - **KNN**
- **Cross validation**
- **Decision Tree**
- Linear Models
  - Linear Regression
  - Least Square : Regularized, Ridge vs Lasso
- Underfit and Overfit
- Ml Optimization: Gradient Descent, SGD
- Evaluation of Classification and Regression
- Introduction to Bayesian Learning: Probabilistic Learning
  - Bayes Theorem, MLE, MAP
  - Naïve Bayes Classifier
- Logistic Regression, SoftMax Regression
- Support Vector Machine(SVM)
- Kernel SVM
- Unsupervised Learning:
  - K-mean clustering
  - Dimension Reduction : PCA
  - Kernel PCA

Machine Learning (ML)

- Designing algorithms that ingest data and learn a model of the data
- The learned model can be used to
  - Detect patterns/structures/themes/trends etc. in the data
  - Make predictions about future data and make decisions

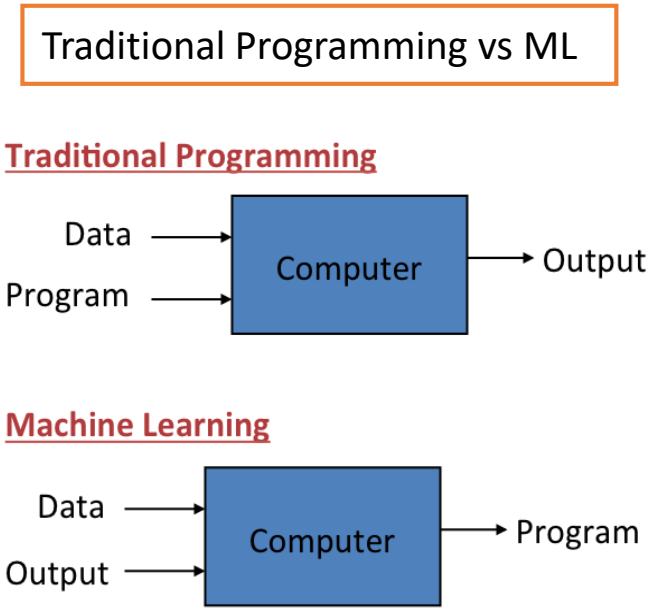


- Modern ML algorithms are heavily "data-driven"
  - No need to pre-define and hard-code all the rules (infeasible/impossible anyway)
  - The rules are not "static"; can adapt as the ML algo ingests more and more data

Formal Definition of ML

- Here is a slightly more general definition:
- Machine Learning *is the field of study that gives computers the ability to learn without being explicitly programmed.* —Arthur Samuel, 1959
- The name machine learning was coined in 1959 by Arthur Samuel.

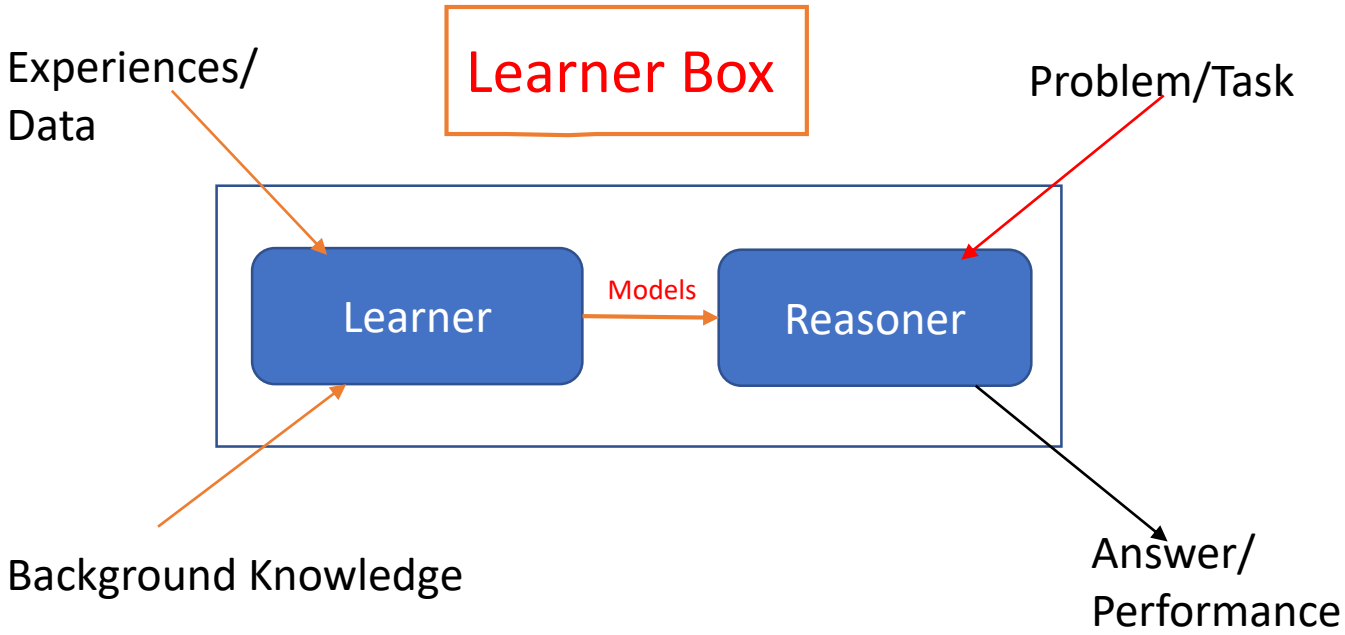
- And a more engineering-oriented one:
- *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.* —Tom Mitchell, 1997
- Ability of computers to "learn" from "data" or "past experience"
  - data: Comes from various sources such as sensors, domain knowledge, experimental runs, etc.
  - learn: Make *intelligent* predictions or decisions based on data by optimizing a model



ML learner system as a Box

Schematic diagram Learner system.

- Two main components, learner(L) and Reasoner(R).
- L takes experience/data and background knowledge and learns a Model.
- Reasoner works with the models and for given an new problem and a task it can come up with the solutions and the performance measures

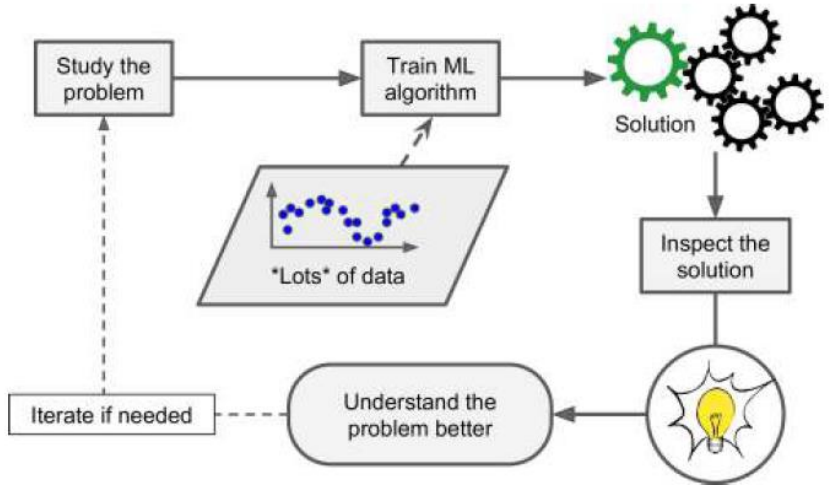


Steps for Creating a Learner:

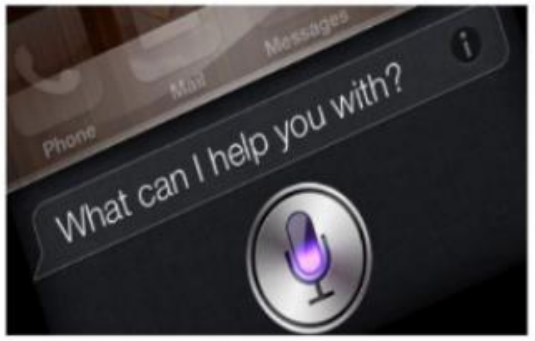
1) Choose the training experience/data(features)
2) Choose target function or how we want to represent a model that is to be learn
3) Choose how to represent the target function(class of functions)
4) Finally Choose a learning algorithms to infer the target function

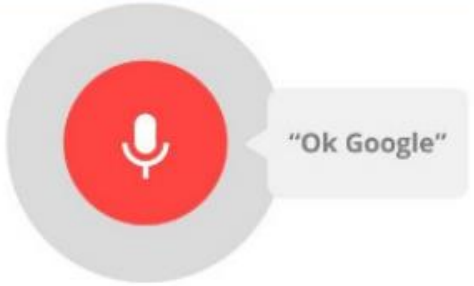
Why Use Machine Learning?

- Develop systems that can automatically adapt and customize themselves to individual users.
- Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.
- Ability to mimic human and replace certain monotonous tasks - which require some intelligence.
- Develop systems that are too difficult/expensive to construct manually
- Problems for which existing solutions require a lot of fine-tuning or long lists of rules
- Fluctuating environments: a Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data.
- Finally, Machine Learning can help humans learn



ML Applications Abound..





Predictive Policing

Online Fraud Detection

Key Enablers for Modern ML

- Availability of large amounts of data to train ML models

- Increased computing power (e.g., GPUs)

Tesla V100

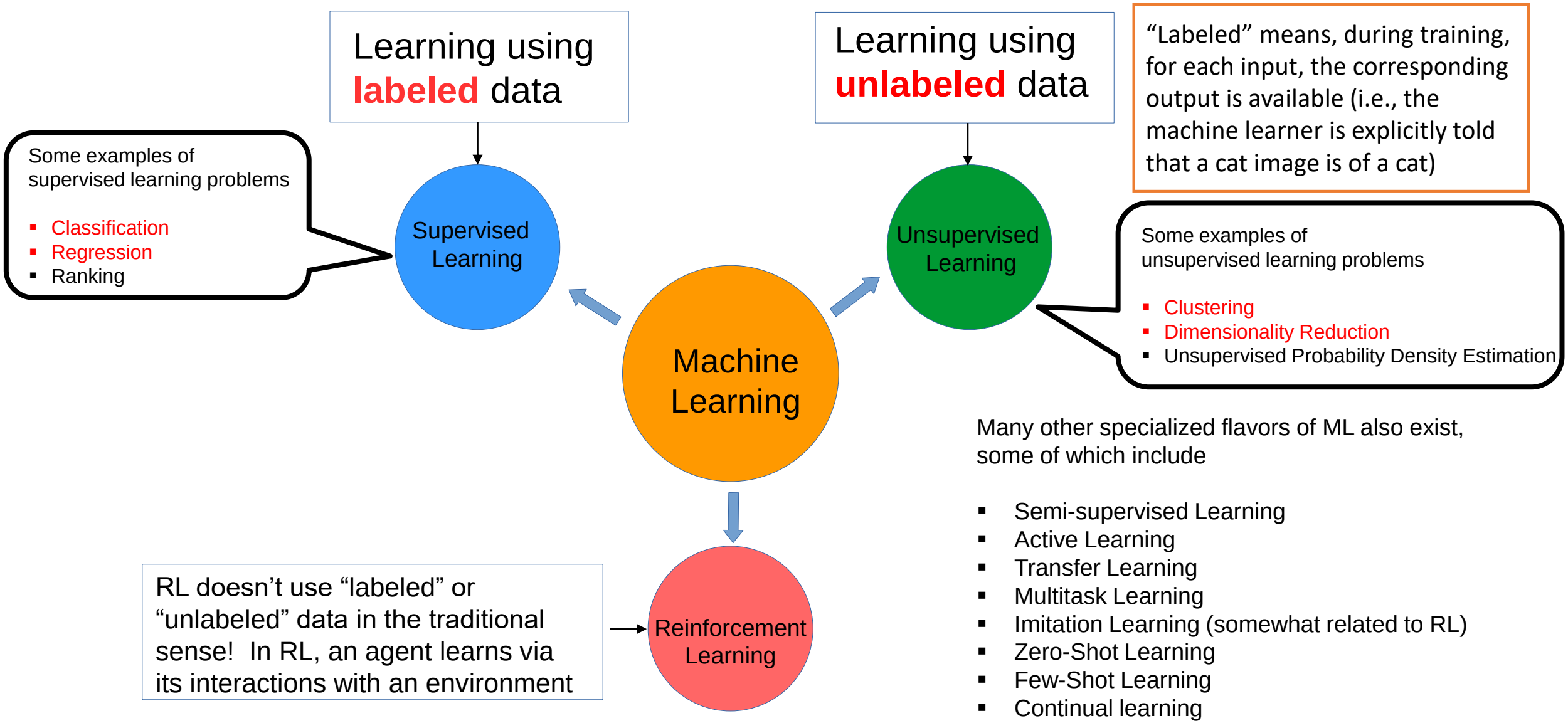
A100

ML is like an exam

- It's the performance on the D-day which matters
- In an exam, our success is measured based on how well we did on the questions in the test (not on the questions we practiced on)
- Likewise, in ML, success of the learned model is measured based on how well it predicts/fits the future test data (not the training data)

In Machine Learning, generalization performance on the test data matters

A Loose Taxonomy of ML

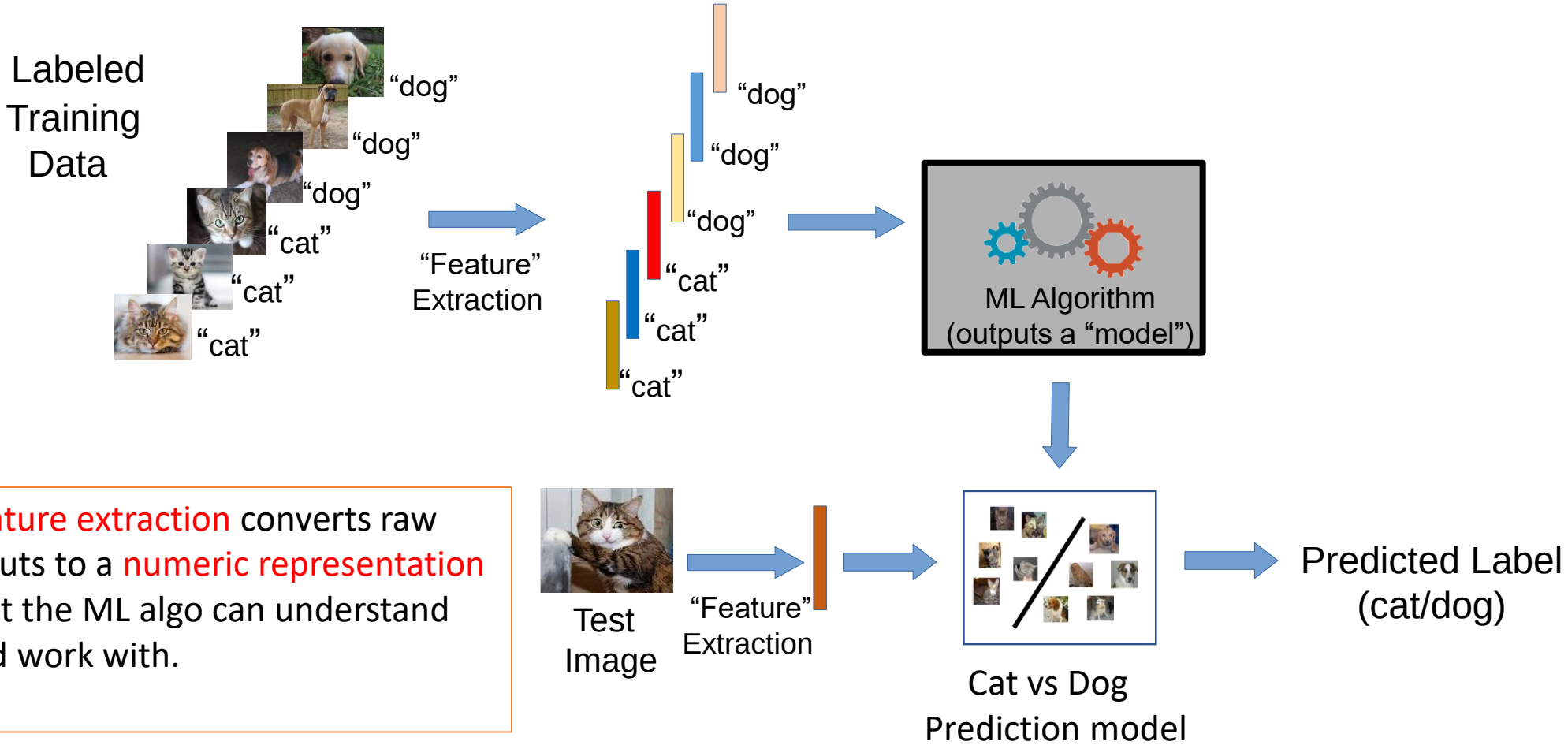


Many other specialized flavors of ML also exist, some of which include

- Semi-supervised Learning
- Active Learning
- Transfer Learning
- Multitask Learning
- Imitation Learning (somewhat related to RL)
- Zero-Shot Learning
- Few-Shot Learning
- Continual learning

A Typical Supervised Learning Workflow

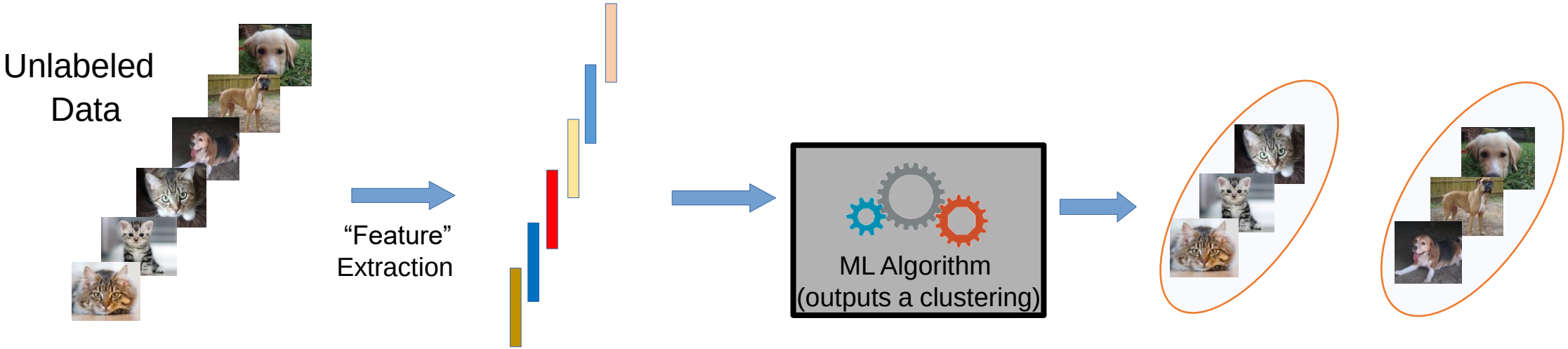
Binary classification



Feature extraction converts raw inputs to a numeric representation that the ML algo can understand and work with.

A Typical Unsupervised Learning Workflow

Data clustering, an unsupervised learning problem



- Unsupervised learning also have a test phase.
- That is, can we also predict the cluster of a new test input?
- In this example, given a new "test" cat/dog image, we can assign it to the cluster with closer centroid

Data and Features

Data and Features

- ML algos require a numeric feature representation of the inputs
- Features are properties that describes each training instances
- Features can be obtained using one of the two approaches
  - Approach 1: Extracting/constructing features manually from raw inputs
  - Approach 2: Learning the features from raw inputs

- Approach 1 is what we will focus on primarily in ML

- Approach 2 is what is followed in Deep Learning algorithms

  Deep Learning = ML with automated feature learning from the raw inputs
- Approach 1 is not as powerful as Approach 2 but still used widely

Example: Feature Extraction for Text Data

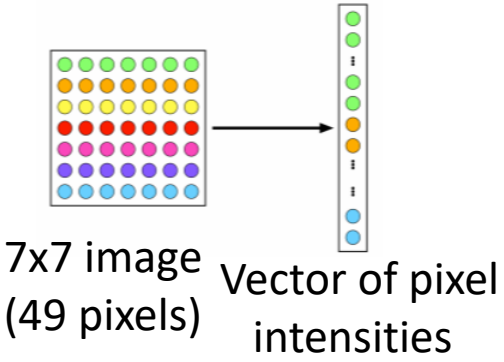
- Consider some text data consisting of the following sentences:
  - John likes to watch movies
  - Mary likes movies too
  - John also likes football
- Want to construct a feature representation for these sentences
- Here is a "bag-of-words" (BoW) feature representation of these sentences

	John	likes	to	watch	movies	Mary	too	also	football
Sentence 1	1	1	1	1	1	0	0	0	0
Sentence 2	0	1	0	0	1	1	1	0	0
Sentence 3	1	1	0	0	0	0	0	1	1

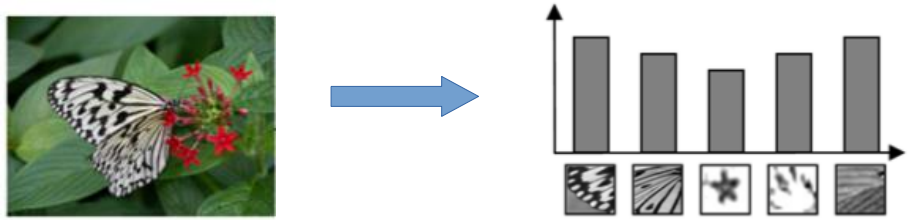
- Each sentence is now represented as a binary vector (each feature is a binary value, denoting presence or absence of a word).

Example: Feature Extraction for Image Data

- A very simple feature extraction approach for image data is flattening



- Histogram of visual patterns is another popular feature extr. method for images



- Many other manual feature extraction techniques developed in computer vision and image processing communities (SIFT, HoG, and others)

Types of Features and Types of Outputs

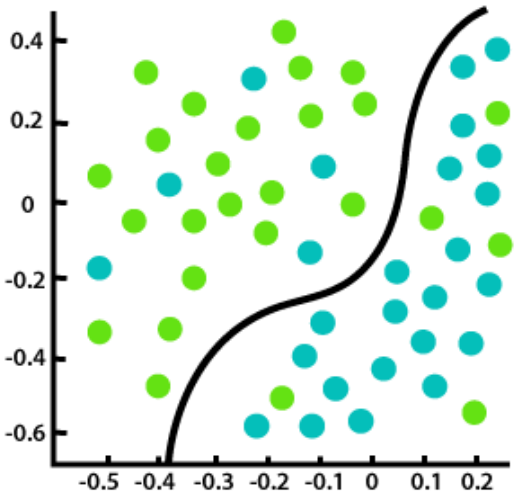
- Features as well as outputs can be real-valued, binary, categorical, ordinal, etc.
- Real-valued: Pixel intensity, house area, house price, rainfall amount, temperature, etc
- Binary: Male/female, adult/non-adult, or any yes/no or present/absent type value
- Categorical/Discrete: Zipcode, blood-group, or any “one from a finite many choices“ value
- Ordinal: Grade (A/B/C etc.) in a course, or any other type where relative values matter
- Often, the features can be of mixed types (some real, some categorical, some ordinal, etc.)

Regression vs Classification

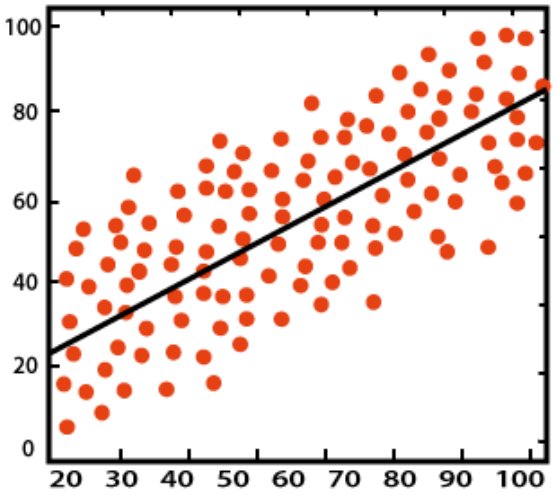
- Supervised Learning requires training data as N input-output pairs $\{(\mathbf{x_n}, y_n)\}_{n=1}^{N}$
- Given examples ($\hat{x}$, y) , Target function f : X → Y, need to be learn ($\hat{x}$, f($\hat{x}$)). Function of features vector.
- Classification: f($\hat{x}$) is discrete
- Regression: f($\hat{x}$) is continuous

- Apart from this two in some cases we may want to find out probability of a particular value of Y.
- Probability estimation : f($\hat{x}$) is probability of $\hat{x}$

		Features				
		X_1	X_2	.....	X_n	Y
Instances	I_1	a_1	a_2	.....	a_n	Y_1
	I_2	b_1	b_2	.....	b_n	Y_2
	I_3	c_1	c_2	.....	c_n	Y_3



Classification



Regression

Some Types of Supervised Learning Problems

- Consider building an ML module for an e-mail client
- Some tasks that we may want this module to perform
  - Predicting whether an email of spam or normal: Binary Classification
  - Predicting which of the many folders the email should be sent to: Multi-class Classification
  - Predicting all the relevant tags for an email: Tagging or Multi-label Classification
  - Predicting what`s the spam-score of an email: Regression
  - Predicting which email(s) should be shown at the top: Ranking
  - Predicting which emails are work/study-related emails: One-class Classification
- These predictive modeling tasks can be formulated as supervised learning problems
- A very simple supervised learning model for binary/multi-class classification
  - This model doesn’ t require any fancy maths – just computing means and distances

Supervised Learning:
Learning by Computing Distances

Computing Distances

- Euclidean (L2 norm) distance between two vectors $\boldsymbol{a} \in \mathbb{R}^D$ and $\boldsymbol{b} \in \mathbb{R}^D$

$$d_2(\boldsymbol{a}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}||_2 = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2} = \sqrt{(\boldsymbol{a} - \boldsymbol{b})^\top (\boldsymbol{a} - \boldsymbol{b})} = \sqrt{\boldsymbol{a}^\top \boldsymbol{a} + \boldsymbol{b}^\top \boldsymbol{b} - 2\boldsymbol{a}^\top \boldsymbol{b}}$$

- Weighted Euclidean distance between two vectors $\boldsymbol{a} \in \mathbb{R}^D$ and $\boldsymbol{b} \in \mathbb{R}^D$
- (W is d by d diagonal matrix) If W is symmetric it called Mahalanobis distance

$$d_w(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum_{i=1}^{D} w_i (a_i - b_i)^2} = \sqrt{(\boldsymbol{a} - \boldsymbol{b})^\top \mathbf{W} (\boldsymbol{a} - \boldsymbol{b})}$$

- Absolute (L1 norm) distance between two vectors $\boldsymbol{a} \in \mathbb{R}^D$ and $\boldsymbol{b} \in \mathbb{R}^D$

$$d_1(\boldsymbol{a}, \boldsymbol{b}) = ||\boldsymbol{a} - \boldsymbol{b}||_1 = \sum_{i=1}^{D} |a_i - b_i|$$

Cat vs Dog: A Very Primitive Classifier

- Consider a binary classification problem – cat vs dog
- Assume training data with just 2 images – one  and one 
- Given a new test image (cat/dog), how do we predict its label?
- A simple idea: Predict using its distance from each of the 2 training images

$d($ Test image $,$  $) < d($ Test image $,$  $)$?

Predict cat <u>else</u> dog

Improving the Primitive Classifier

- Just one input per class may not sufficiently capture variations in a class

- A natural improvement can be by using more inputs per class

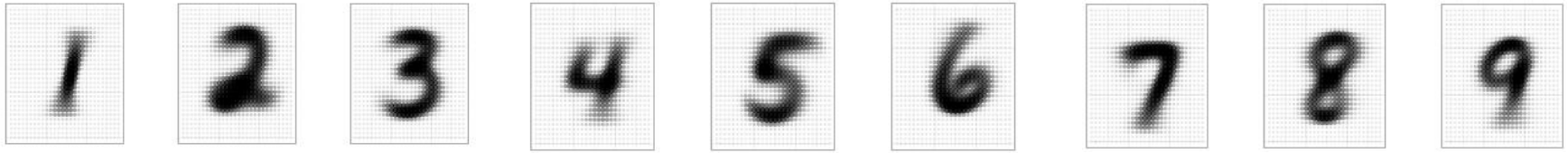
- Let us consider two approaches to do this
  - Learning with Prototypes (LwP)
  - Nearest Neighbors)

- Both LwP and NN will use multiple inputs per class but in different ways

Learning with Prototypes (LwP)

- Basic idea: Represent each class by a "prototype" vector

- Class Prototype: The "mean" or "average" of inputs from that class



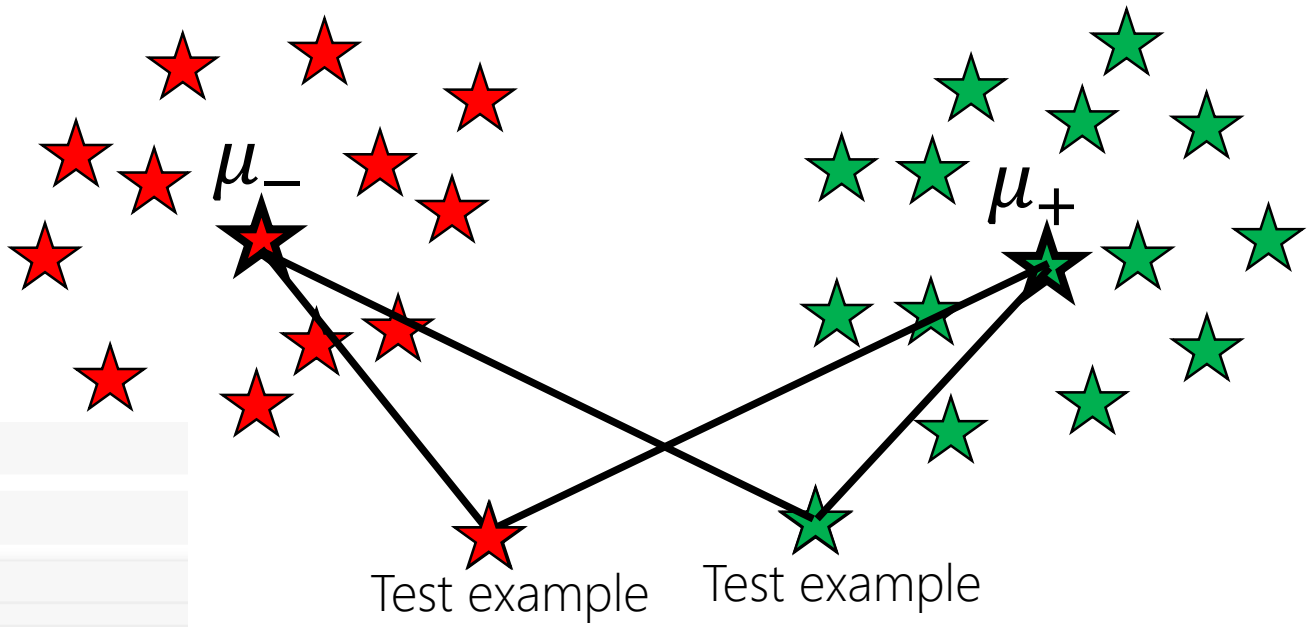
Averages (prototypes) of each of the handwritten digits 1-9

- Predict label of each test input based on its distances from the class prototypes
  - Predicted label will be the class that is the closest to the test input
- How we compute distances can have an effect on the accuracy of this model (may need to try Euclidean, weight Euclidean, Mahalanobis, or something else)

Learning with Prototypes (LwP): An Illustration

- Suppose the task is binary classification (two classes assumed pos and neg)
- Training data: N labelled examples $\{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$, $\mathbf{x}_n \in \mathbb{R}^D$, $y_n \in \{-1, +1\}$
  - Assume N_+ example from positive class, N_- examples from negative class
  - Assume green is positive and red is negative

$$\mu_- = \frac{1}{N_-} \sum_{y_n=-1} \mathbf{x}_n$$



$$\mu_+ = \frac{1}{N_+} \sum_{y_n=+1} \mathbf{x}_n$$

```
[ ]  dog_mean = train_X[train_y==1].mean(axis=0)

[ ]  cat_mean = train_X[train_y==0].mean(axis=0)

lwp_predicts = []
for d in test_X:
    if np.linalg.norm(d-dog_mean)<np.linalg.norm(d-cat_mean):
        lwp_predicts.append(dog)
    else:
        lwp_predicts.append(cat)
```

Nearest Neighbors

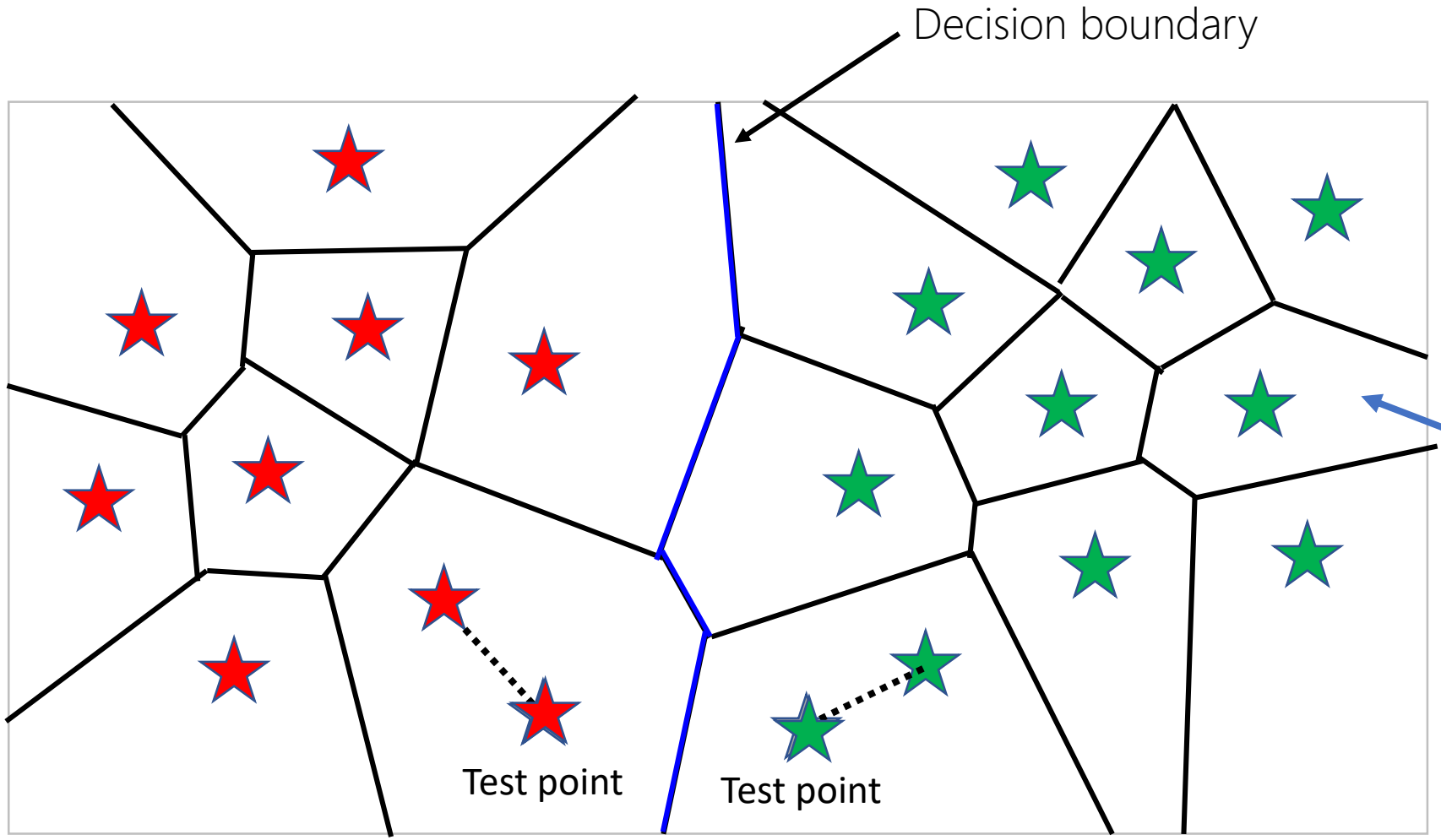
- Another supervised learning technique based on computing distances
- Very simple idea. Simply do the following at test time
  - Compute distance of the test point from all the training points
  - Sort the distances to find the "nearest" input(s) in training data
  - Predict the label using majority or avg label of these inputs
- Can use Euclidean or other dist (e.g., Mahalanobis).

  Choice imp just like LwP

- Unlike LwP which does prototype based comparison, nearest neighbours method looks at the labels of individual training inputs to make prediction
- Applicable to both classifn as well as regression (LwP only works for classifn)

Nearest Neighbors for Classification

Nearest Neighbor (or "One" Nearest Neighbor)

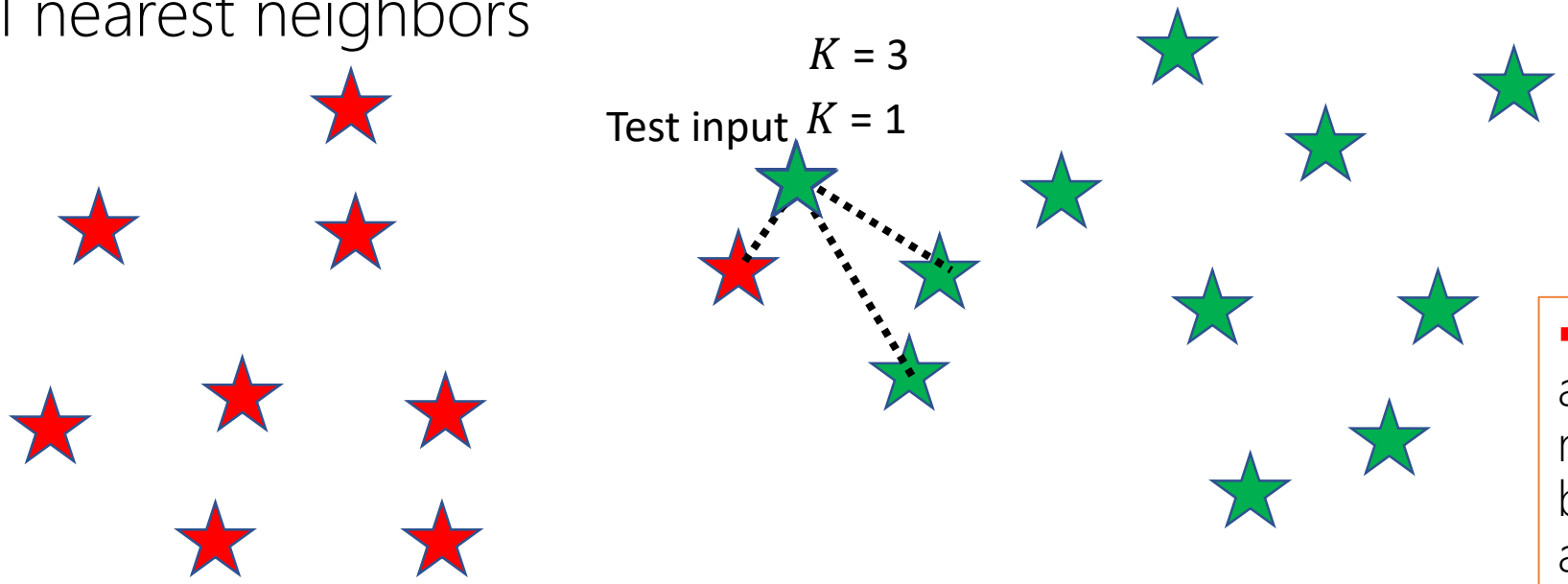


Nearest neighbour approach induces a Voronoi tessellation/partition of the input space (all test points falling in a cell will get the label of the training input in that cell)

K Nearest Neighbors (*K*NN)

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)
model.fit(train_X, train_y)
acc = model.score(test_X, test_y)
print("Model accuracy: {:.2f}%".format(acc * 100))

Model accuracy: 55.44%
```

- In many cases, it helps to look at not one but $K > 1$ nearest neighbors



- ϵ-NN :Rather than looking at a fixed number K of neighbors, can look inside a ball of a given radius ϵ, around the test input

- Essentially, taking more votes helps!
  - Also leads to smoother decision boundaries (less chances of overfitting on training data)
- Can apply KNN/ϵ-NN for other supervised learning problems as well, such as
  - Multi-class classification
  - Regression For regression, simply compute the average of the outputs of nearest neighbors

Hyperparameter Selection

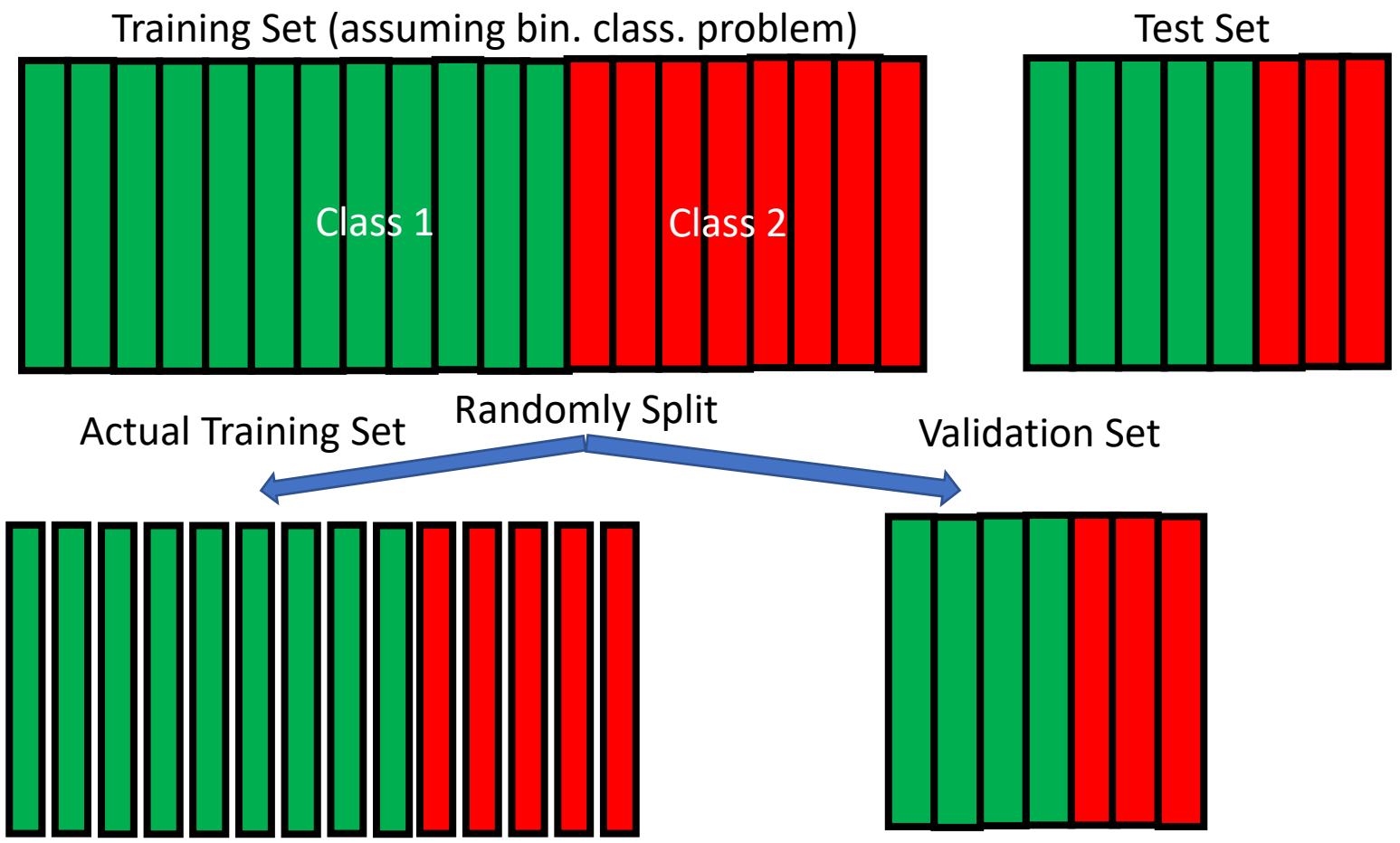
- Every ML model has some hyperparameters that need to be tuned, e.g.,
  - K in KNN or ϵ in ϵ-NN
  - Choice of distance to use in LwP or nearest neighbors

- Would like to choose h.p. values that would give best performance on test data
- Beware. Never Ever touch your test data while building the model

- Use cross-validation - usea part of your training data (we will call it "validation/held-out set") as test data.

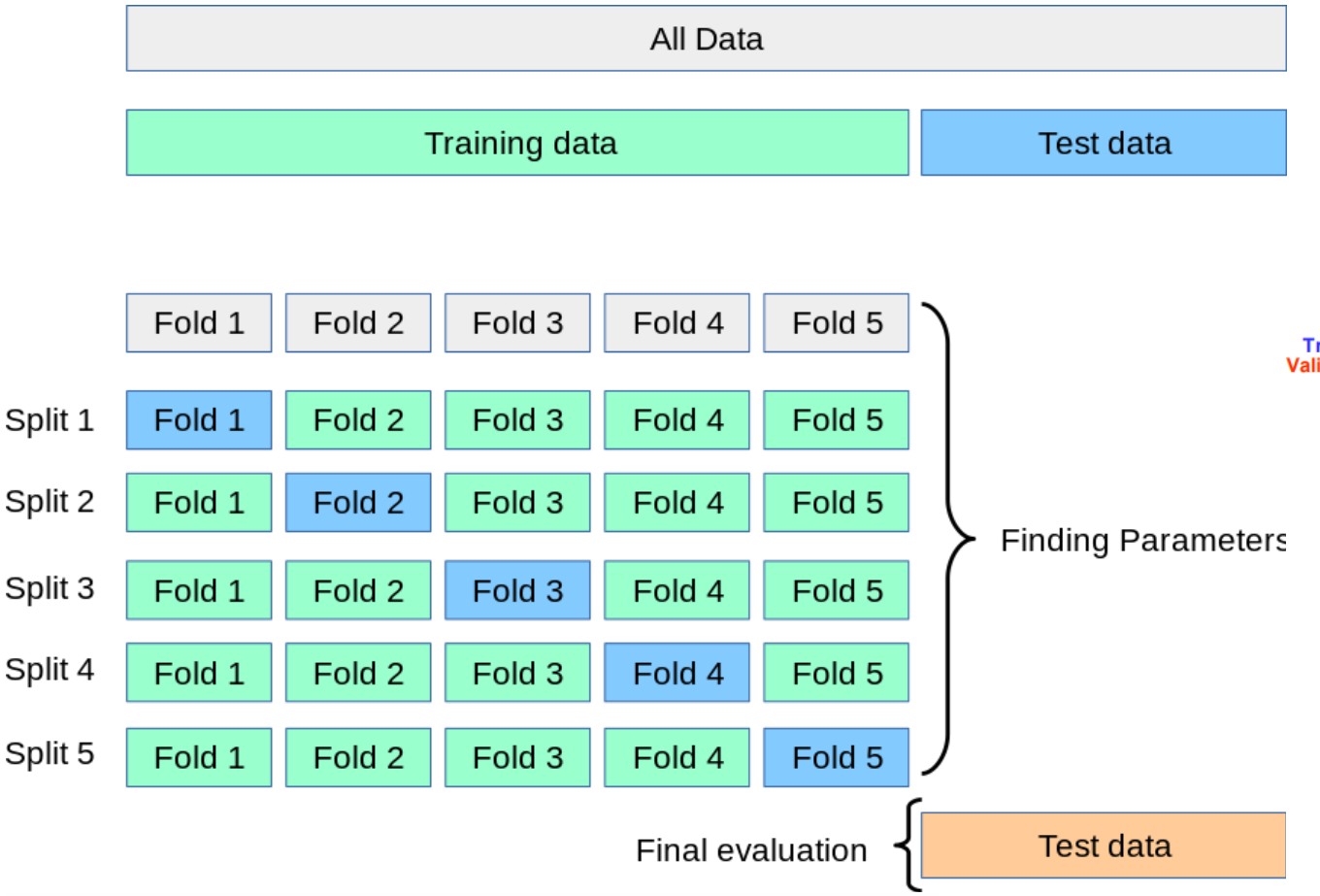
Cross-Validation

No peeking while building the model



- Randomly split the original training data into actual training set and validation set.
- Pick the hyperparam value that gives best accuracy on the validation set
- If you fear an unlucky split, try multiple splits with average CV accuracy
- you are using N splits, this is called N–fold cross validation

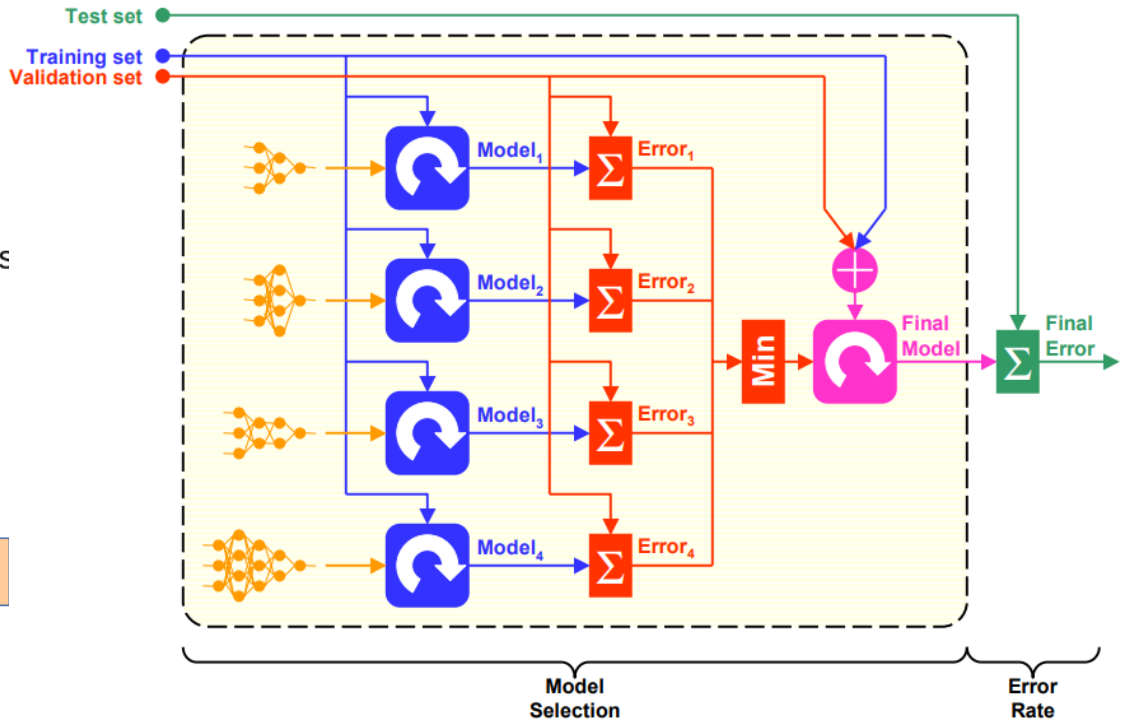
N-Fold Cross-Validation




```
from sklearn.model_selection import cross_val_score
cross_val_score(model, train_X, train_y, cv=3, scoring="accuracy")
```

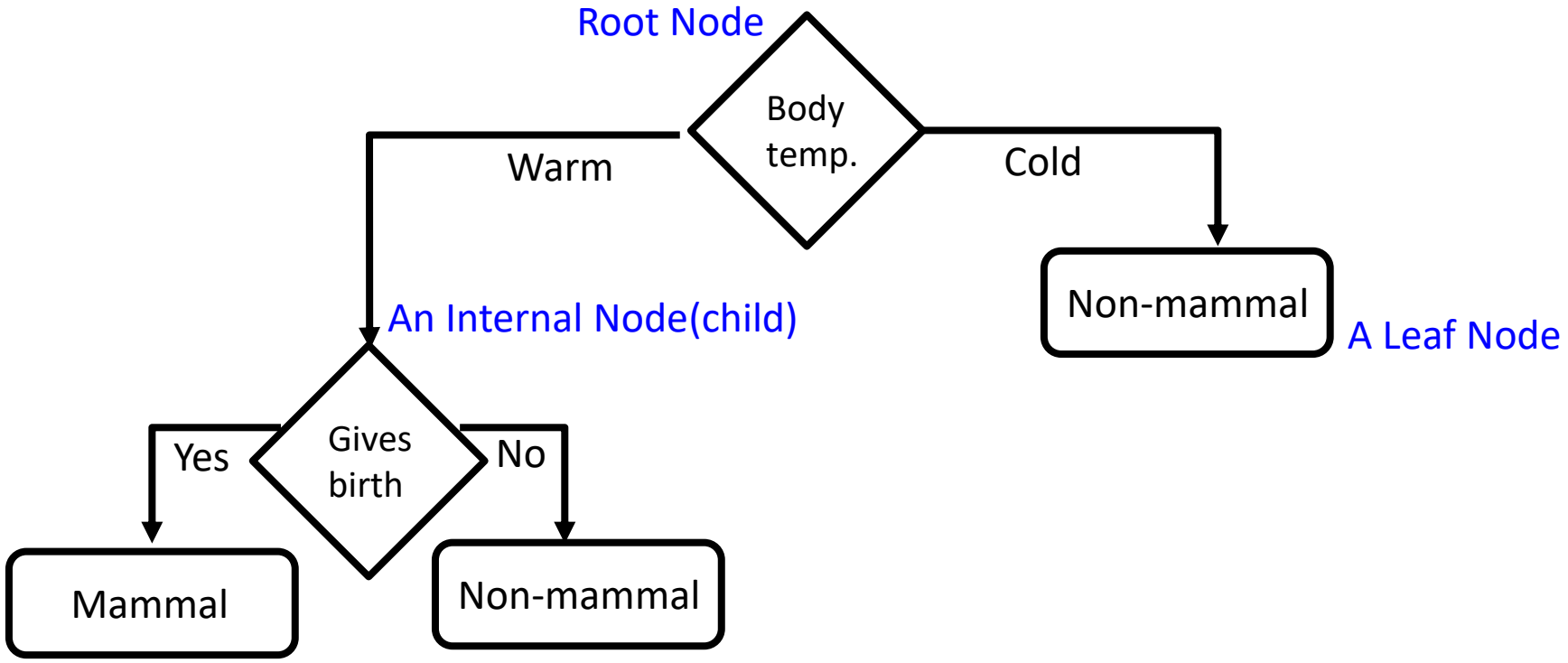
```
array([0.5532    , 0.554     , 0.54341737])
```

- Not just h.p. selection;
- best ML model from a set of different ML models (e.g., say we have to pick between two models we may have trained - LwP and nearest neighbors..



Decision Trees

- A Decision Tree (DT) defines a hierarchy of rules to make a prediction

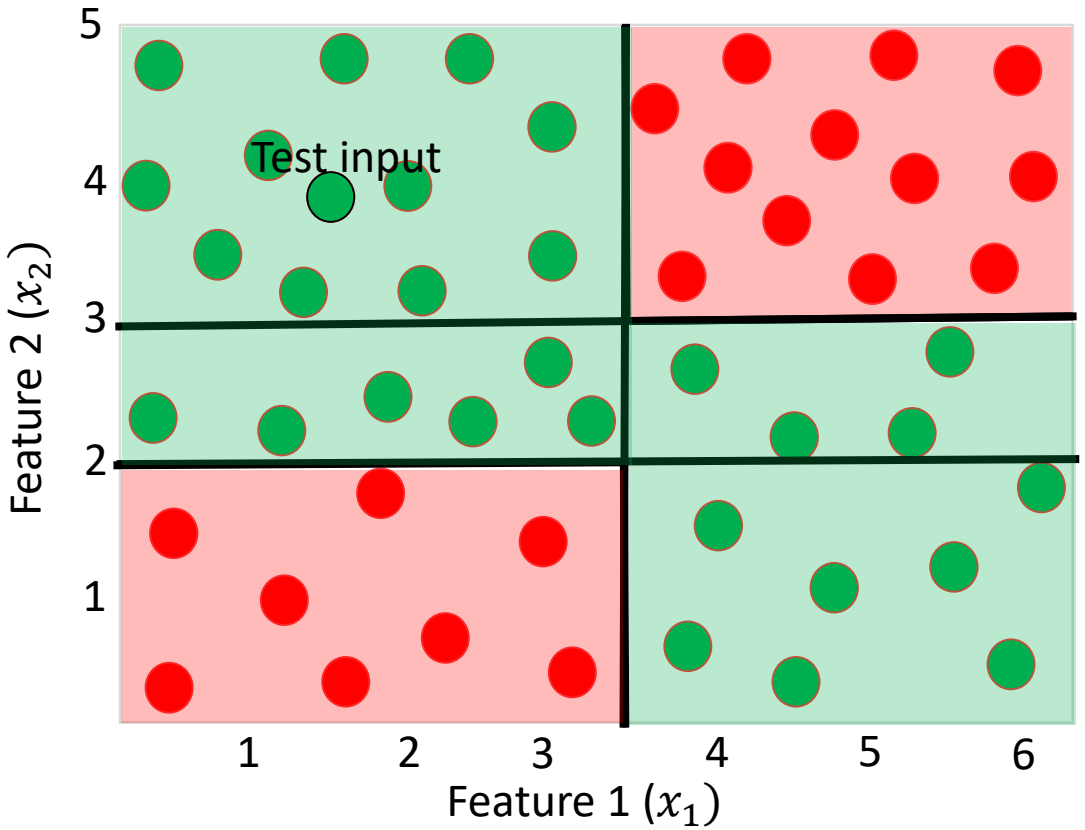


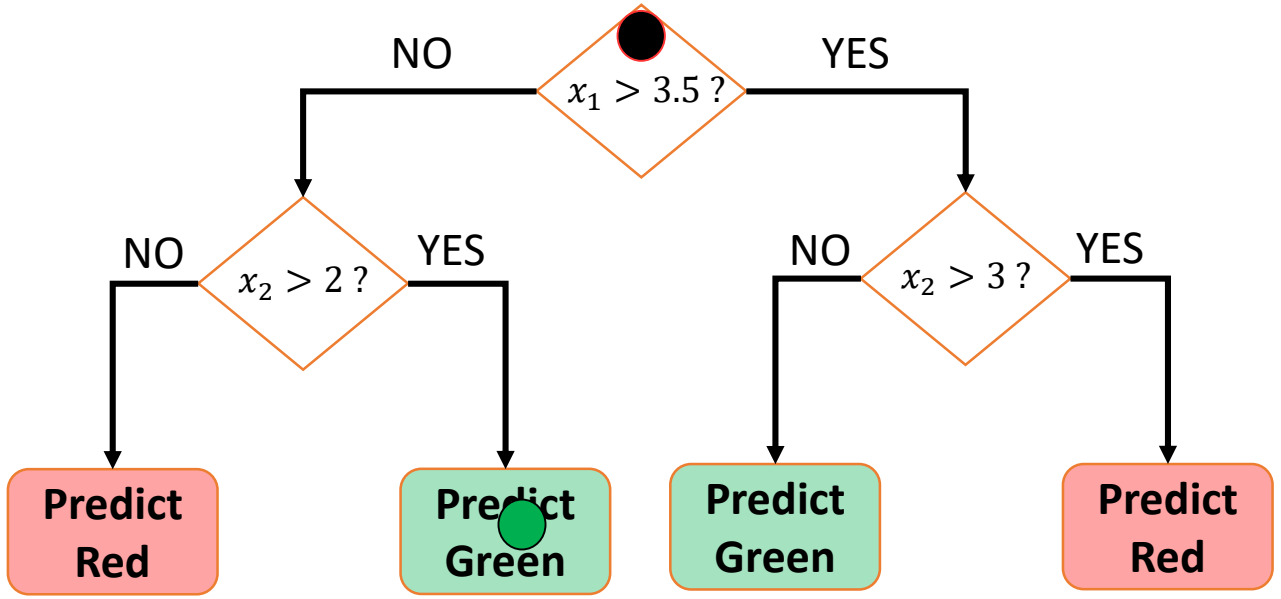
- Root and internal nodes test rules. Leaf nodes make predictions

- Decision Tree (DT) learning is about learning such a tree from labelled data

Decision Trees for Classification

```
[ ] from sklearn.datasets import load_iris
    from sklearn.tree import DecisionTreeClassifier

    iris = load_iris()
    X = iris.data[:, 2:] # petal length and width
    y = iris.target
    tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42, criterion='entropy')
    tree_clf.fit(X, y)
```



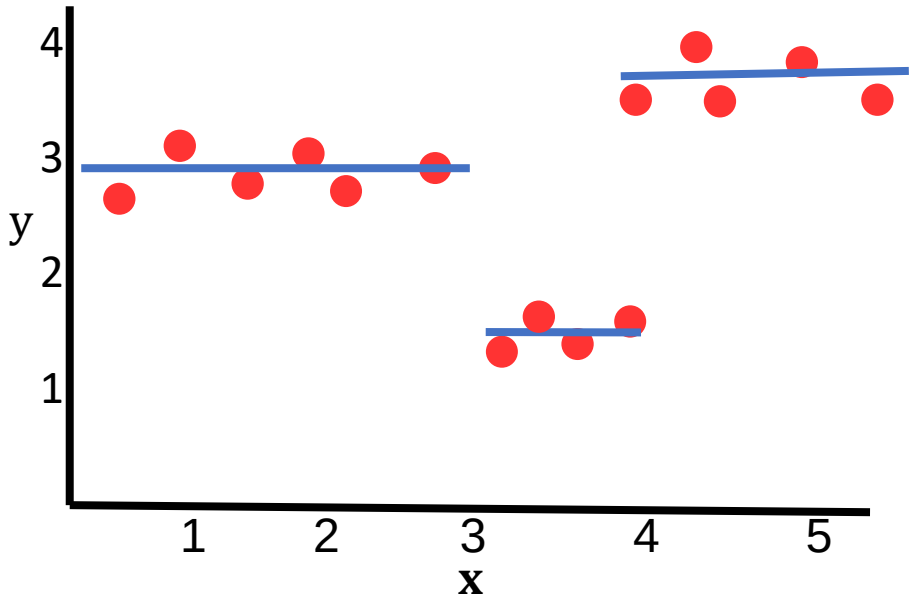


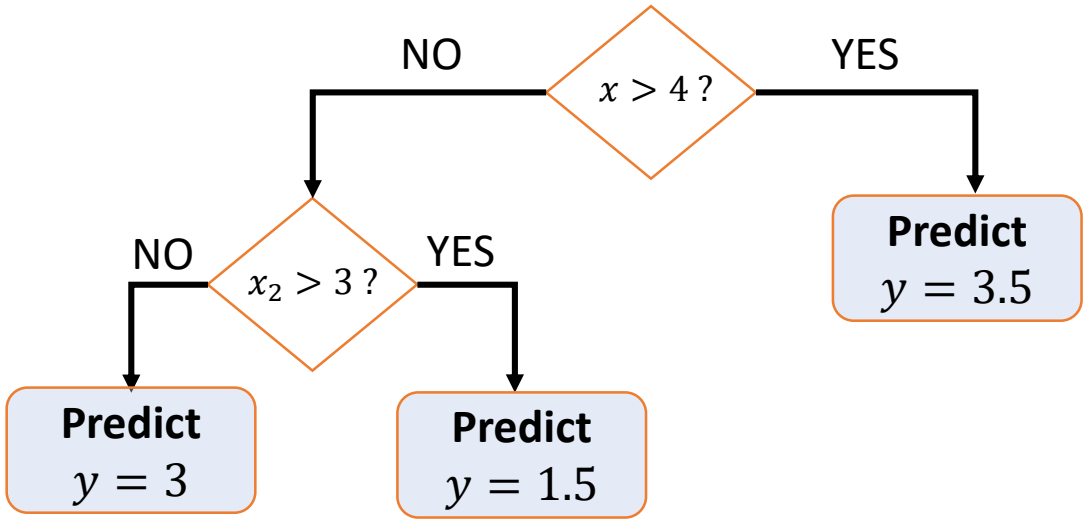
Root node contains all training inputs
Each leaf node receives a subset of training inputs

- The basic idea is very simple
- Recursively partition the training data into homogeneous regions: all (or a majority of) training inputs with the same/similar outputs
- Within each group, fit a simple supervised learner (e.g., predict the majority label)
- DT is very efficient at test time

Decision Trees for Regression

- Can use any regression model but would like a simple one, constant model



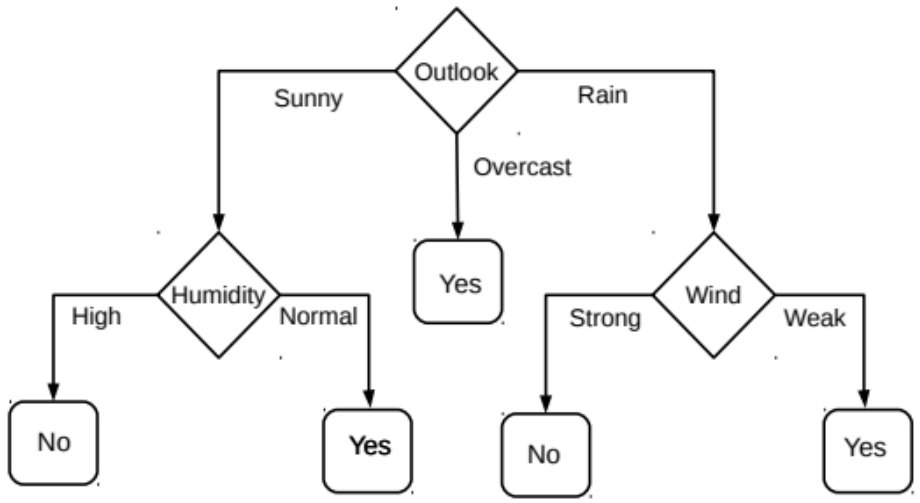


- To predict the output for a test point, nearest neighbors will require computing distances from 15 training inputs.
- DT predicts the label by doing just at most feature-value comparisons.

Decision Tree Construction: An Example

- Let's consider the playing Tennis example
- Assume each internal node will test the value of one of the features

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



- Question: Why does it make more sense to test the feature "outlook" first?
- Answer: Of all the 4 features, it's the most informative
  - It has the highest information gain as the root node

Entropy and Information Gain

- Assume a set of labelled inputs $\boldsymbol{S}$ from C classes, p_c as fraction of class c inputs
- Entropy of the set $\boldsymbol{S}$ is defined as $\mathrm{H}(\boldsymbol{S}) = -\sum_{c \in C} p_c \log p_c$
- Suppose a rule splits $\boldsymbol{S}$ into two smaller disjoint sets $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$
- Reduction in entropy after the split is called information gain

$$IG = H(S) - \frac{|S_1|}{|S|} H(S_1) - \frac{|S_2|}{|S|} H(S_2)$$

```
[ ]  from sklearn.tree import DecisionTreeRegressor

     tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
     tree_reg.fit(X, y)
```

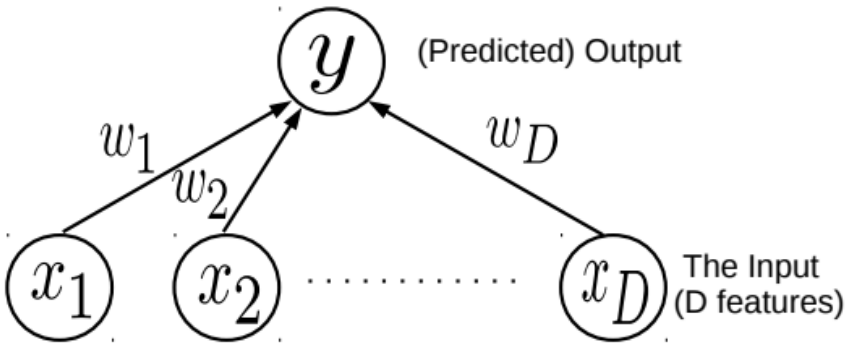
```
]  from sklearn.datasets import load_iris
   from sklearn.tree import DecisionTreeClassifier

   iris = load_iris()
   X = iris.data[:, 2:] # petal length and width
   y = iris.target
   tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42, criterion='entropy')
   tree_clf.fit(X, y)
```

- Uniform sets (all classes roughly equally present) have high entropy; skewed sets low [5,2]
- $IG(S, wind) = 0.048$, IG(S, outlook) = 0.246, IG(S, humidity) = 0.151, IG(S,temp) = 0.029. Thus we choose "outlook" feature to be tested at the root node
- Stop expanding a node further It consist of all training examples having the same label (the node becomes "pure")
- Gini-index defined as $\sum_{c=1}^{C} p_c(1 - p_c)$ can be an alternative to IG
- For regression like entropy/IG/gini etc. are undefined. Thresholding is used
- Learning optimal DT is (NP-hard) intractable. Existing algos mostly greedy heuristics

Linear Models

- Consider learning to map an input $\boldsymbol{x} \in \mathbb{R}^D$ to the corresponding(say real-valued) output $\boldsymbol{y}$
- Assume the output to be a linear weighted combination of the $\boldsymbol{D}$ input features

$$y = \sum_{d=1}^{D} w_d x_d = \boldsymbol{w}^\top \boldsymbol{x}$$



This defines a linear model with D parameters given by a "weight vector" $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$

The "optimal" weights are unknown and have to be learned by solving an optimization problem, using some **training data**

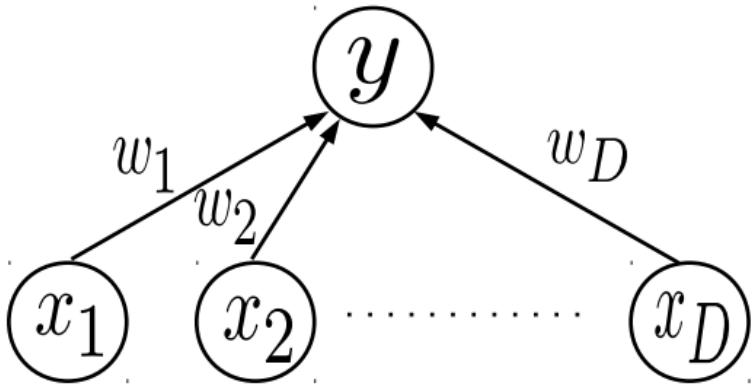
- This simple model can be used for Linear Regression
- This simple model can also be used as a "building block" for more complex models
  - Even classification (binary/multiclass/multi-output/multi-label) and various other ML/deep learning models
  - Even unsupervised learning problems (e.g., dimensionality reduction models)

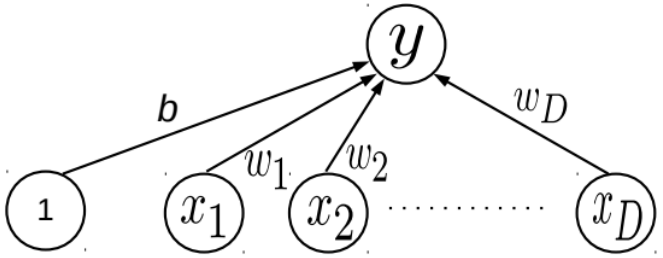
Simple Linear Models as Building Blocks

- A linear model $y = \boldsymbol{w}^\top \boldsymbol{x}$ can be used in classification problems

- For binary classfn, can treat $\boldsymbol{w}^\top \boldsymbol{x}$ as the "score" of input $\boldsymbol{x}$ and threshold to get binary label

$$y = +1 \quad \text{if} \quad \boldsymbol{w}^\top \boldsymbol{x} \geq 0$$
$$y = -1 \quad \text{if} \quad \boldsymbol{w}^\top \boldsymbol{x} < 0$$
$$y = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x})$$





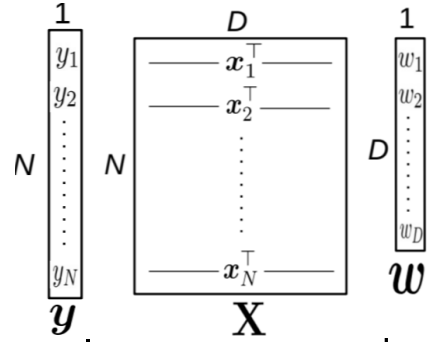
$$y = \sum_{d=1}^{D} w_d x_d + b = \boldsymbol{w}^\top \boldsymbol{x} + b$$

- linear model of the form $\boldsymbol{w}^\top \boldsymbol{x} + b$? . Can easily fold-in the Bias term b here as shown in the figure below

- Can append a constant feature "1" for each input and rewrite as $y = \boldsymbol{w}^\top \boldsymbol{x}$ where now both $\boldsymbol{x}$ and $\boldsymbol{w} \in \mathbb{R}^{D+1}$

Linear Regression

- Given: Training data with N input-output pairs $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^{N}$, $\boldsymbol{x}_n \in \mathbb{R}^D$, $y_n \in \mathbb{R}$



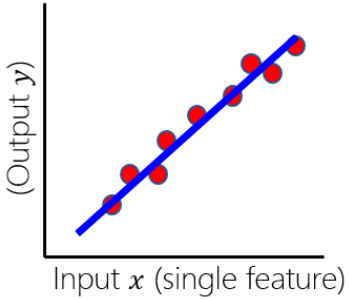
- Goal: Learn a model to predict the output for new test inputs
- Assume the function that approximates the I/O relationship to be a linear model

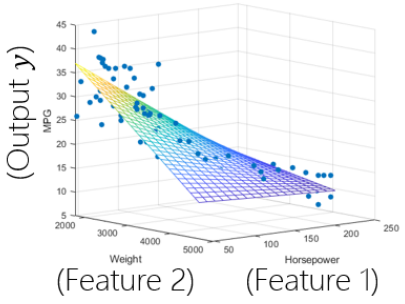
$$y_n \approx f(\boldsymbol{x}_n) = \boldsymbol{w}^\top \boldsymbol{x}_n \quad (n = 1,2,\dots,N)$$

- Let's write the total error or "loss" of this model over the training data as

$$L(\boldsymbol{w}) = \sum_{n=1}^{N} \ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n)$$

- $\ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n)$ measures the prediction error or "loss" or "deviation" of the model on a single training input $(\boldsymbol{x}_n, y_n)$





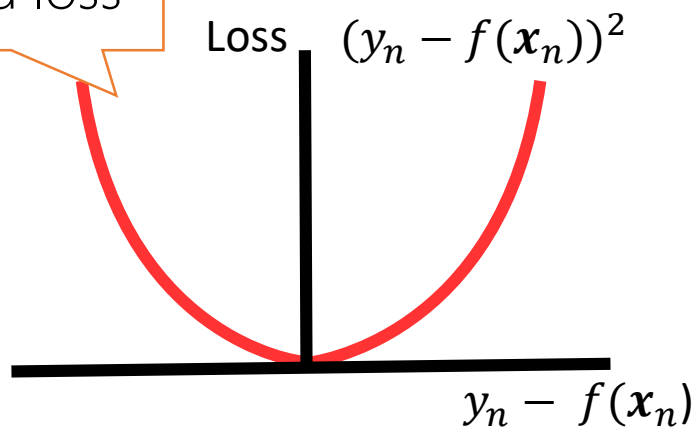
- Goal of learning is to find the $\boldsymbol{w}$ that minimizes this loss + does well on test data

Loss Functions for Regression

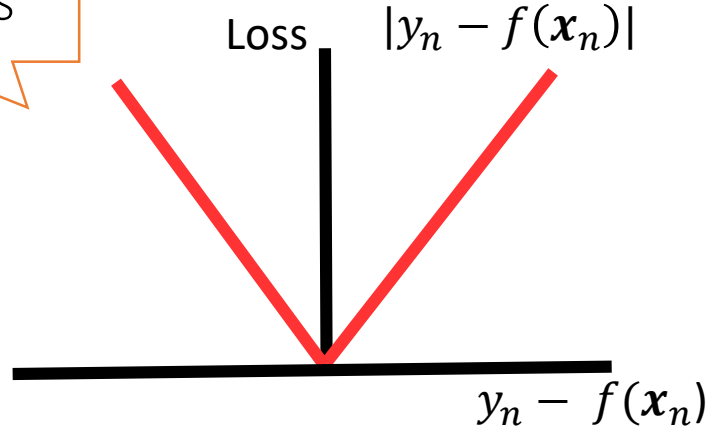
- Many possible loss functions for regression problems

- Choice of loss function usually depends on the nature of the data.
- Also, some loss functions result in easier optimization problem than others

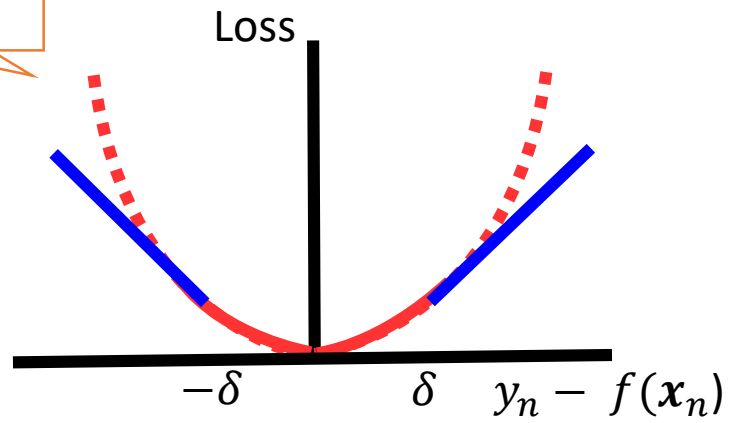
Squared loss



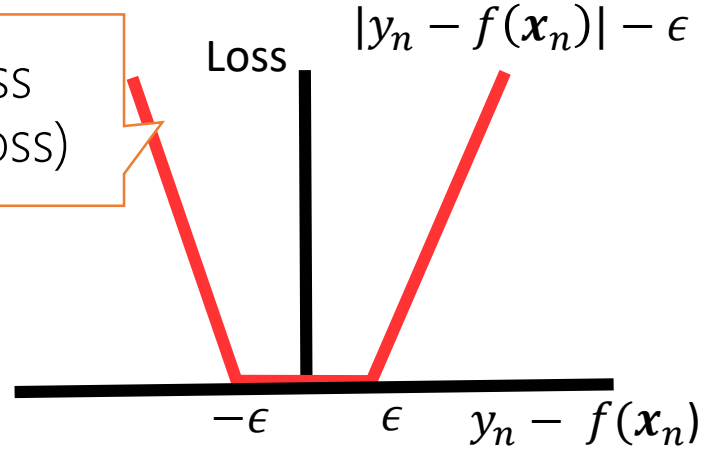
Absolute loss



Huber loss



ϵ-insensitive loss
(a.k.a. Vapnik loss)



Linear Regression with Squared Loss

- In this case, the loss func will be $L(\boldsymbol{w}) = \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$
- In matrix-vector notation, can write it compactly as $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}$
- Let us find the $\boldsymbol{w}$ that optimizes (minimizes) the above squared loss
- The "least squares" (LS) problem
- We need calculus and optimization to do this!
- The LS problem can be solved easily and has a closed form solution

$$\boldsymbol{w}_{LS} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}} \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 = \sum_{n=1}^{N} 2(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n) \frac{\partial}{\partial \boldsymbol{w}}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n) = 0$$

Using the fact $\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{x}_n = \boldsymbol{x}_n$, we get $\sum_{n=1}^{N} 2(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n = 0$

$$\boldsymbol{w}_{LS} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top\right)^{-1} \left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

Normal Equation

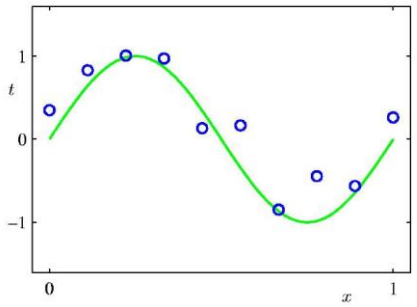
Problem(s) with the Solution!

- We minimized the objective $L(\boldsymbol{w}) = \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$ w.r.t. $\boldsymbol{w}$ and got

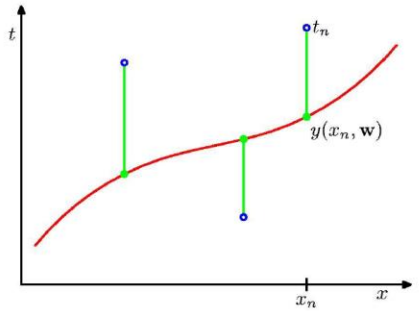
$$\boldsymbol{w}_{LS} = (\sum_{n=1}^{N} \boldsymbol{x}_n \, \boldsymbol{x}_n^\top)^{-1} (\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \, \boldsymbol{X}^\top \boldsymbol{y}$$

- Problem: The matrix $\boldsymbol{X}^\top \boldsymbol{X}$ may not be invertible
  - This may lead to **non-unique solutions** for $\boldsymbol{w}_{opt}$
- Problem: Overfitting since we only minimized loss **defined on training data**
  - Weights $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$ may become arbitrarily large to fit training data perfectly
  - Such weights may perform poorly on the test data however
- One Solution: Minimize a **regularized objective** $L(\boldsymbol{w}) + \lambda \, R(\boldsymbol{w})$
  - $R(\boldsymbol{w})$ is called the Regularizer and measures the "magnitude" of $\boldsymbol{w}$
  - The reg. will prevent the elements of $\boldsymbol{w}$ from becoming too large
  - Reason: Now we are minimizing training error + magnitude of vector $\boldsymbol{w}$

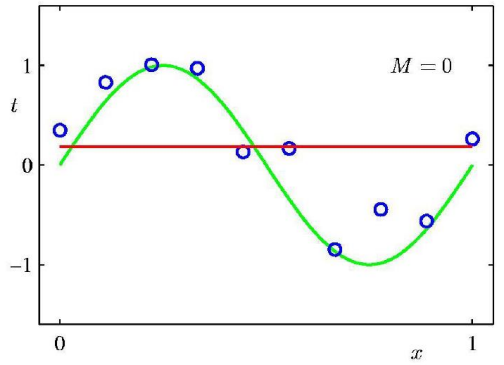
Over-fitting, Under-fitting,

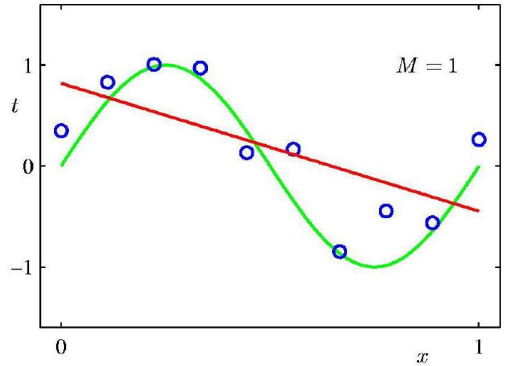


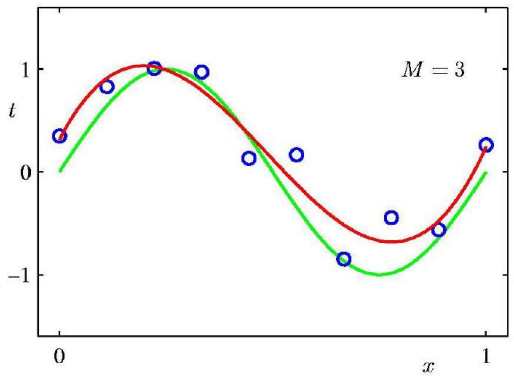
$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

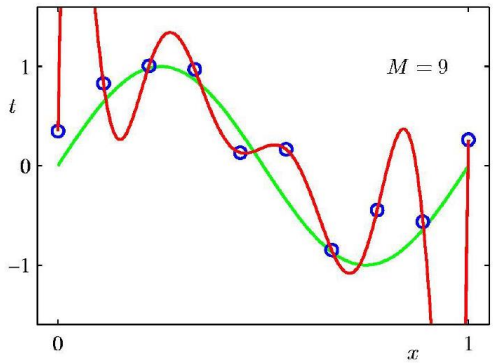


$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2$$









	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

- Training a model there are two major problems one can encounter: Overfitting and Underfitting
- Overfitting : perform well in training but not so well on unseen(test) data
- Underfitting: Neither performs well on train set nor on the test set

Large coeff value for poly degree 9

Regularized with Ridge

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Regularized Least Squares

- Recall that the regularized objective is of the form $L_{reg}(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})$
- One possible/popular regularizer(Ridge): the squared Euclidean (ℓ_2 squared) norm of $\boldsymbol{w}$

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|_2^2 = \boldsymbol{w}^\top \boldsymbol{w}$$

- With this regularizer, we have the regularized least squares problem as

$$\boldsymbol{w}_{ridge} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})$$

$$= \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- Proceeding just like the LS case, we can find the optimal $\boldsymbol{w}$ which is given by

$$\boldsymbol{w}_{ridge} = (\textstyle\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda I_D)^{-1} (\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda I_D)^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

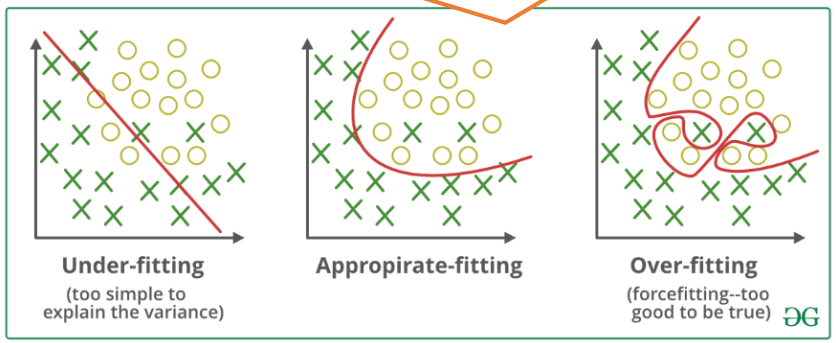
- Use a regularizer $R(\boldsymbol{w})$ defined by other norms, e.g., $\|\boldsymbol{w}\|_1 = \sum_{d=1}^{D} |w_d|$, $\|\boldsymbol{w}\|_0 = \#\text{nnz}(\boldsymbol{w})$

ℓ_1 norm regularizer

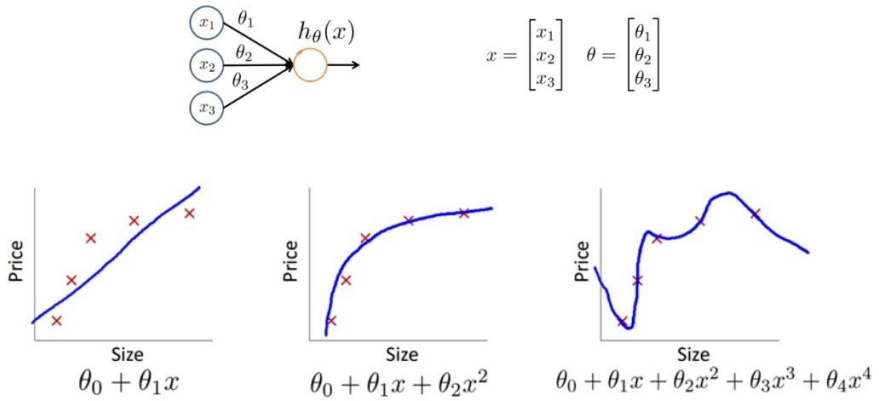
- Use them if you have a very large number of features but many irrelevant features. These regularizers can help in automatic feature selection (explained in next slide)
- Using such regularizers gives a sparse weight vector $\boldsymbol{w}$ as solution
- Use non-regularization based approaches: Early-stopping, Dropout, Injecting noise in the inputs

ℓ_2(Ridge) vs ℓ_1(Lasso) regularization

Classification: Over/Under fitting



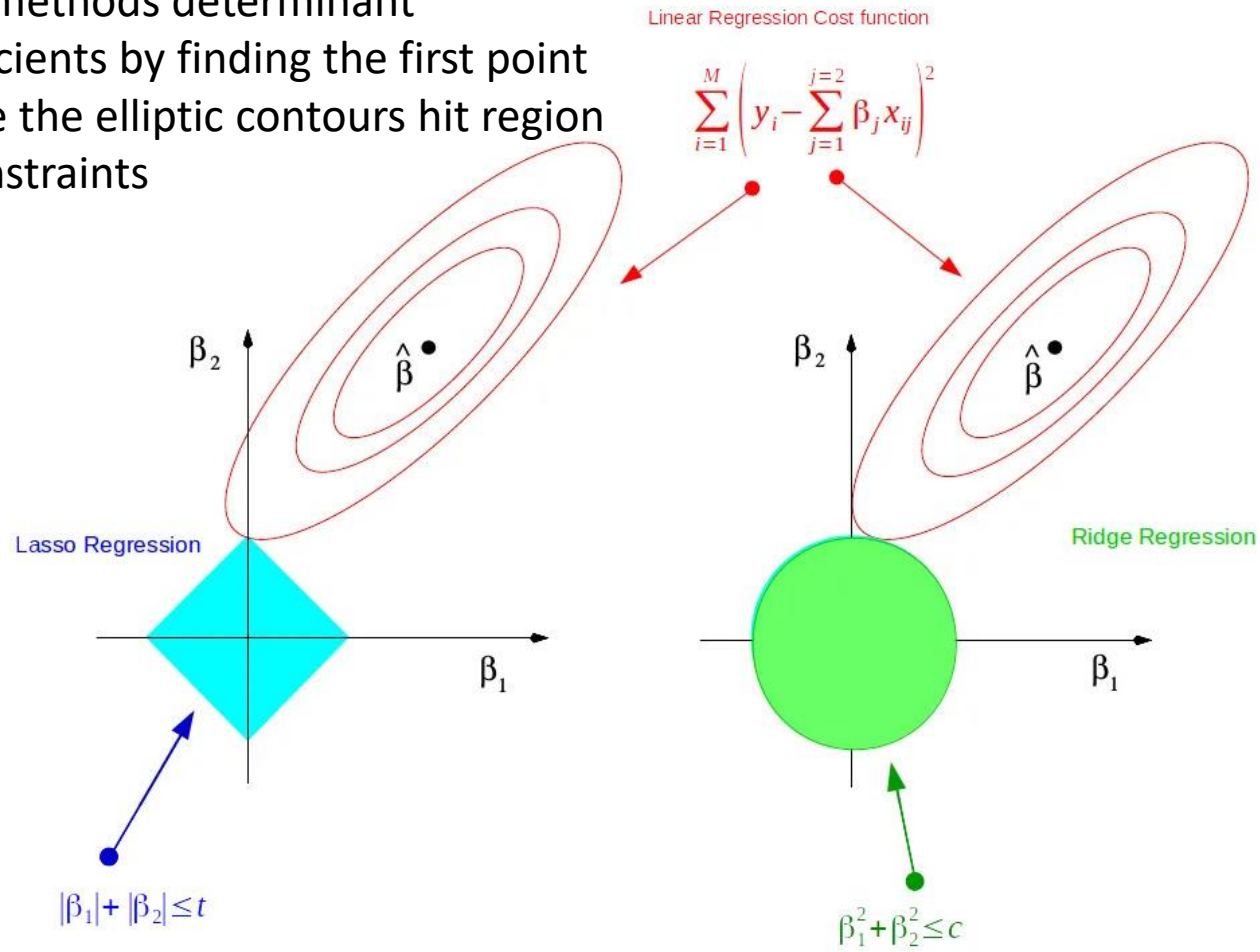
Regression: Over/under fitting



- In overfitting, if we have too many features, the learned hypothesis may fit the training set very well, but fail to generalize to new examples (predict prices on new examples).

- Lasso tends to generate sparser solutions than a quadratic regularizer

Both methods determinant coefficients by finding the first point where the elliptic contours hit region of constraints

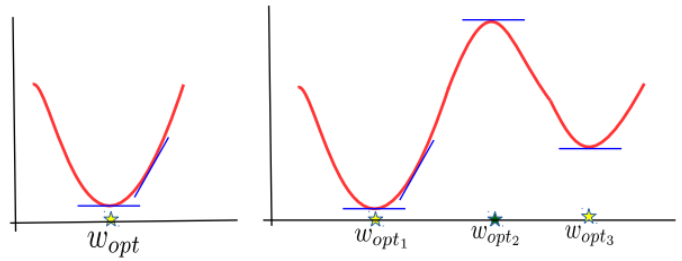


Optimization Problems in ML: Gradient Descent

- The general form of an optimization problem in ML will usually be

$$w_{opt} = \arg\min_{w \in C} L(w)$$

- Here $L(w)$ denotes the loss function to be optimized



- C is the constraint set that the solution must belong to, e.g.,
  - Non-negativity constraint: All entries in w_{opt} must be non-negative
  - Sparsity constraint: w_{opt} is a sparse vector with atmost K non-zeros
- If no C is specified, it is an unconstrained

For max. problems we can use gradient ascent

$$w^{(t+1)} = w^{(t)} + \eta_t g^{(t)}$$

- Method 1: Using First-Order Optimality

First order optimality: The gradient g must be equal to zero at the optima $g = \nabla_w[L(w)] = 0$ and solving for w gives a closed form solution

- **Method 2: Iterative Optimiz. via Gradient Descent**

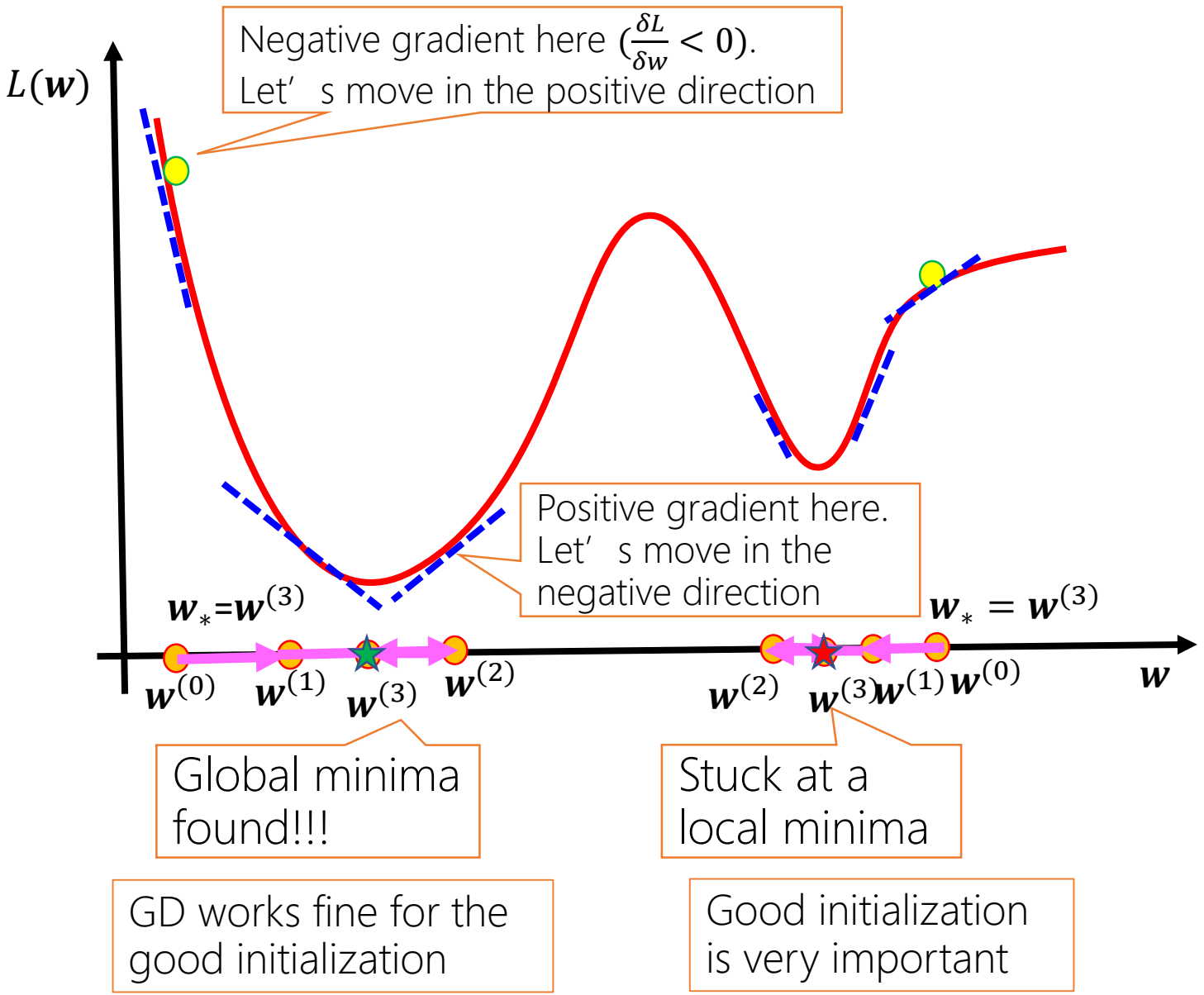
Gradient Descent

- Initialize w as $w^{(0)}$
- For iteration $t = 0,1,2, \ldots$ (or until convergence)
  - Calculate the gradient $g^{(t)}$ using the current iterates $w^{(t)}$
  - Set the learning rate η_t
  - Move in the opposite direction of gradient

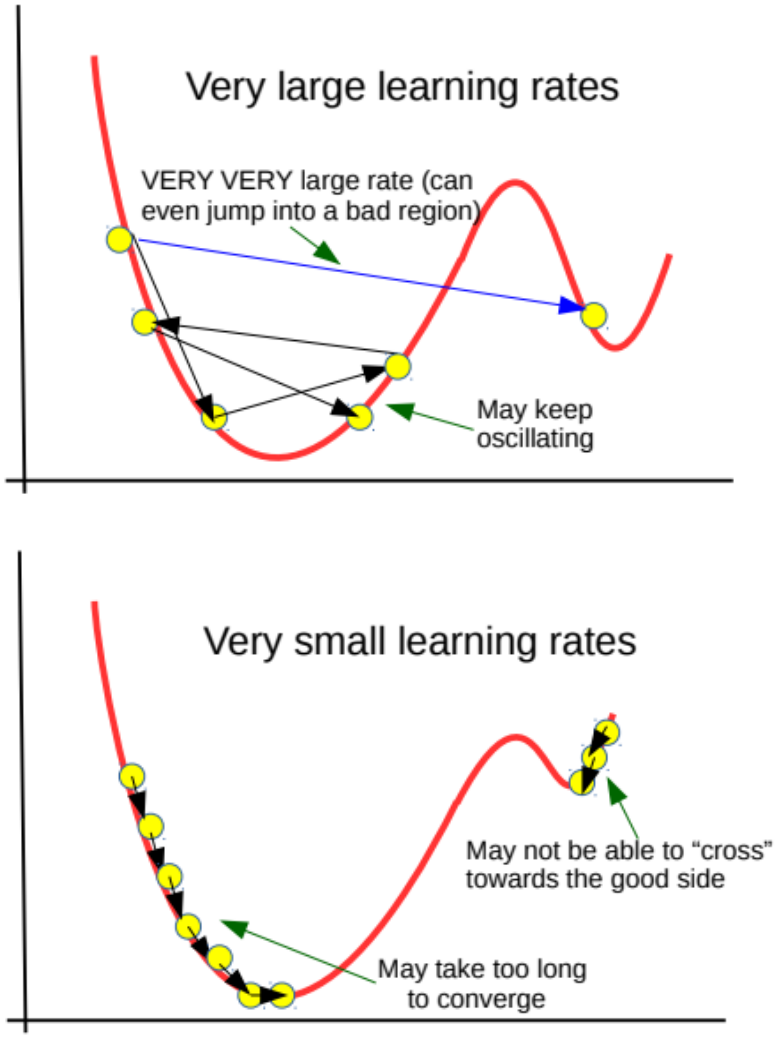
$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}$$

Fact: Gradient gives the direction of steepest change infunction' s value

Gradient Descent: An Illustration



Learning rate is very important



Stochastic Gradient Descent (SGD)

- Consider a loss function of the form $L(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})$
- The gradient in this case can be written as

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} L(w) = \nabla_{\boldsymbol{w}}[\frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w})] = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{g}_n$$

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single training example
- At iter. t, pick an index $i \in \{1,2,\dots,N\}$ uniformly randomly and approximate $\boldsymbol{g}$ as

$$\boldsymbol{g} \approx \boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

- May take more iterations than GD to converge but each iteration is much faster
  - SGD per iter cost is $O(D)$ whereas GD per iter cost is $O(ND)$
- Gradient approximation using a single training example may be noisy
- We can use $B > 1$ unif. rand. chosen train. ex. with indices $\{i_1, i_2, \dots, i_B\} \in \{1,2,\dots,N\}$

$$\boldsymbol{g} \approx \frac{1}{B}\sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

Evaluation of Classification Model

- Classification -- Model evaluations

- Confusion Matrix
  - TP – True Positive ; FP – False Positive
  - FN – False Negative; TN – True Negative

		Predicted Class	
		Class = Yes	Class = No
Actual Class	Class = Yes	a (TP)	b (FN)
	Class = No	c (FP)	d (TN)

$$\text{Precision (p)} = \frac{\text{TP}}{\text{TP + FP}} = \frac{\text{a}}{\text{a + c}}$$

$$\text{Recall (r)} = \frac{\text{TP}}{\text{TP + FN}} = \frac{\text{a}}{\text{a + b}}$$

$$\text{F - measure (F)} = \frac{2\text{rp}}{\text{r + p}} = \frac{2\text{a}}{2\text{a + b + c}}$$

$$\text{Accuracy} = \frac{a + d}{a + b + c + d} = \frac{TP + TN}{TP + TN + FP + FN}$$

Metric can be focused as per problems

- Others evaluation measures

Receive Operating Characteristics(ROC) Curve

It is a plot of true positive rate(TRP) against the false positive rate(FPR)

$$TPR = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{TN + FP}$$

Sensitivity and Specificity:

There are two other evaluation measures:

Sensitivity. Same as TPR

Specificity: called True Negative Rate

$$TNR = \frac{TN}{TN + FP}$$

FPR=1-specificity

https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec

Evaluation of Regression Model

- Regression -- Model evaluations
  - Pearson correlation measures the linear association between continuous variables
    - Quantifies the degree to which a relationship between two variables can be described by a line.

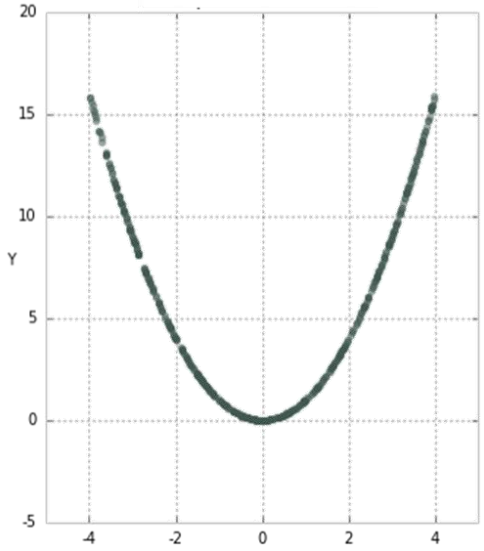
$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$

- Remember the definition of cosine between vectors:

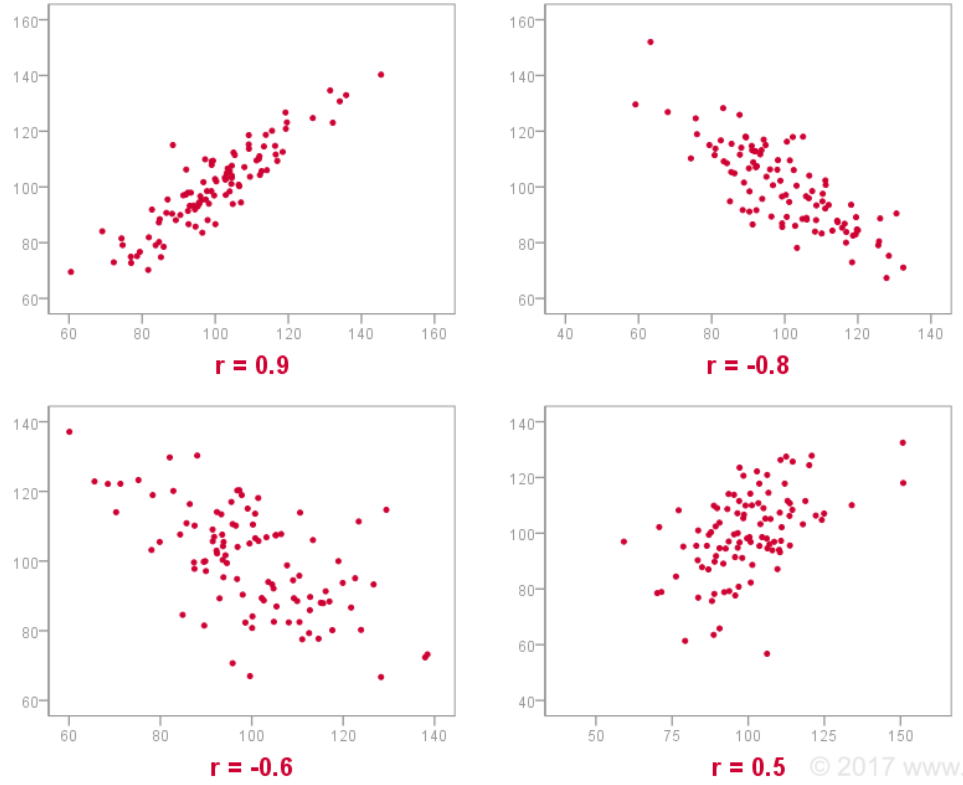
$$\cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Limitation:

- Only linear correlation can be detected.
- Clearly, there are some relationship between X and Y, but the correlation is only 0.02.



- Examples of Pearson correlation



© 2017 www.sp

Evaluation of Regression: Coefficient of determination (R^2)

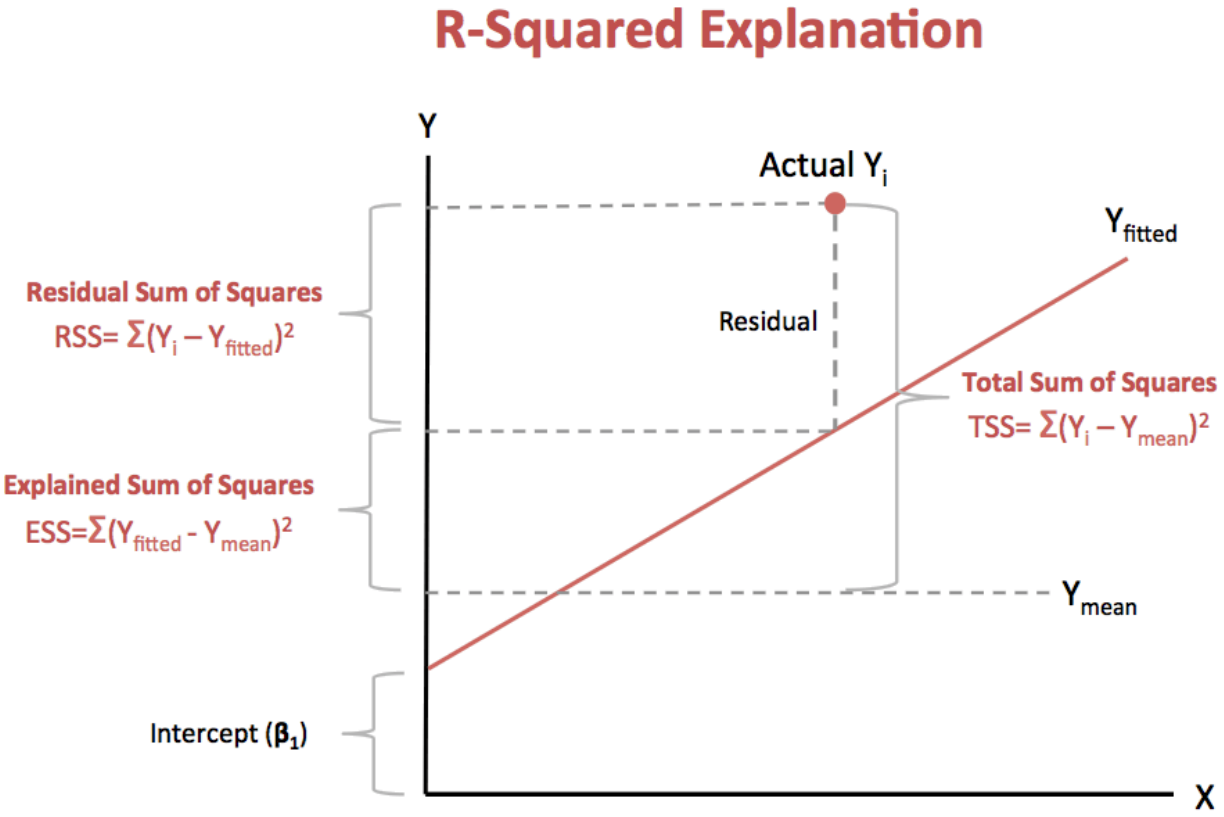
- Coefficient of determination (R^2) is the proportion of the variance in the dependent variable that is predictable from the independent variable.
- It measures how much of the residue can be explained by the regression line

- $R^2 = \dfrac{explained\ variance}{total\ variance}$

- Total variance: $SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$
- Explained variance: $SS_{\text{reg}} = \sum_i (f_i - \bar{y})^2$
- Or, it can be computed as:

$R^2 \equiv 1 - \dfrac{SS_{\text{res}}}{SS_{\text{tot}}}$ where

$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2$

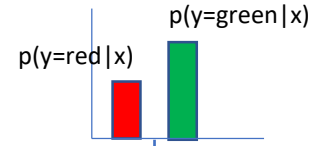


$R_{Sq} = 1 - \dfrac{RSS}{TSS}$

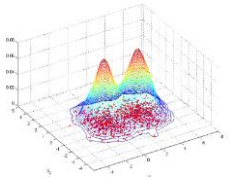
Introduction to Bayesian Learning

Probabilistic ML: Some Motivation

- In many ML problems, we want to model and reason about data probabilistically
- At a high-level, this is the density estimation view of ML, e.g.,



  - Given input-output pairs $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$ estimate the conditional $p(y|\boldsymbol{x})$
  - Given inputs $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$, estimate the distribution $p(\boldsymbol{x})$ of the inputs



  - Note 1: These dist. will depend on some parameters θ (to be estimated), and written as

$$p(y|\boldsymbol{x}, \theta) \qquad \text{or} \qquad p(\boldsymbol{x}|\theta)$$

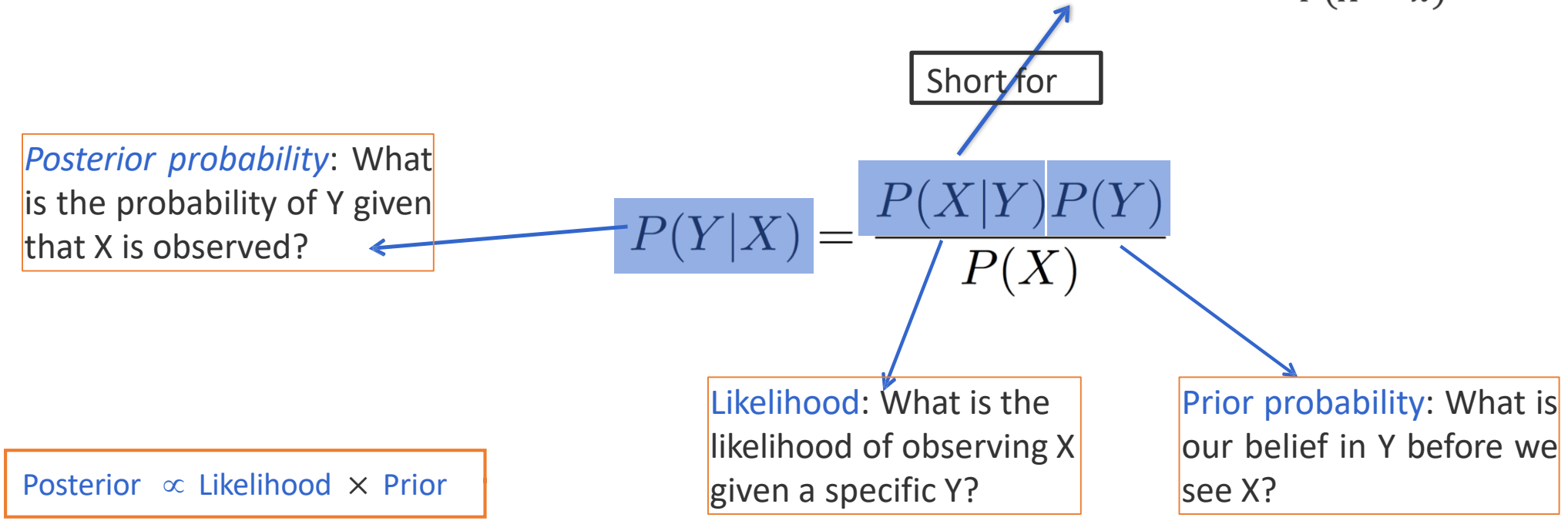
  - Note 2: These dist. sometimes assumed to have a specific form, but sometimes not
- Assuming the form of the distribution to be known, the goal in estimation is to use the observed data to estimate the parameters of these distributions

Probabilistic Learning

- Two different notions of probabilistic learning
- Learning probabilistic concepts
  - The learned concept is a function $c: X \rightarrow [0,1]$
  - c(x) may be interpreted as the probability that the label 1 is assigned to x
- Bayesian Learning: Use of a probabilistic criterion in selecting a hypothesis
  - The hypothesis can be deterministic, a Boolean function
  - The criterion for selecting the hypothesis is probabilistic
- Bayesian Learning: The basics
- Goal: To find the *best* hypothesis from some space H of hypotheses, using the observed data D
- Define best = most probable hypothesis in H
- To do that, we need to assume a probability distribution over the class H
- We also need to know something about the relation between the data observed and the hypotheses

Bayes Theorem

$$\forall x, y \quad P(Y = y|X = x) = \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)}$$



Posterior Probability: Probability of the parameter θ given the evidence data X and is defined by p(θ /X)

Likelihood function: Probability of the evidence of a given parameters: P(X/ θ)

Prior Probability: Prior of a uncertain quantity is a distribution that would express ones beliefs about this quantity before some evidence taken into account

Bayesian Learning

Posterior probability: What is the probability that h is the hypothesis, given that the data D is observed?

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Prior probability of h: Background knowledge. What do we expect the hypothesis to be even before we see any data? For example, in the absence of any information, maybe the uniform distribution.

Likelihood: What is the probability that this data point (an example or an entire dataset) is observed, given that the hypothesis is h ?

What is the probability that the data D is observed (independent of any knowledge about the hypothesis)?

Bayesian learning uses $P(h|D)$, the conditional probability of a hypothesis given the data, to define *best*.

Key insight: Both h and D are events.
D: The event that we observed this particular dataset
h: The event that the hypothesis h is the true hypothesis
So we can apply the Bayes rule here.

Choosing a hypothesis

Given some data, find the most probable hypothesis

– The Maximum a Posteriori hypothesis h_{MAP}

$$
\begin{aligned}
h_{MAP} &= \arg\max_{h \in H} P(h|D) \\
&= \arg\max_{h \in H} \frac{P(D|h)P(h)}{P(D)} \\
&= \arg\max_{h \in H} P(D|h)P(h)
\end{aligned}
$$

If we assume that the prior is uniform

i.e. $P(hi) = P(h_j)$, for all h_i, h_j

– Simplify this to get the Maximum Likelihood hypothesis

$$h_{ML} = \arg\max_{h \in H} P(D|h)$$

Often computationally easier to maximize *log likelihood*

Maximum Likelihood and least squares

Example: Least Squares

- Suppose H consists of real valued functions
- Inputs are vectors $\boldsymbol{x} \in \Re^d$ and the output is a real number $y \in \Re$

Suppose the training data is generated as follows:

- An input $\mathbf{x}_i$ is drawn randomly (say uniformly at random)
- The true function f is applied to get $f(\boldsymbol{x}_i)$
- This value is then perturbed by noise e_i
  - Drawn independently according to an unknown Gaussian with zero mean

$$y_i = f(x_i) + e_i$$

Say we have m training examples (xi, y_i) generated by this process

- Suppose we have a hypothesis h. We want to know what is the probability that a particular label $\boldsymbol{y}_i$ was generated by this hypothesis as $h(\boldsymbol{x}_i)$?
- The error for this example is $y_i - h(\boldsymbol{x}_i)$
- Suppose we assume that this error is from a Gaussian distribution with zero mean and standard deviation= σ
- We can compute the probability of observing one data point (x_i, y_i), if it were generated using the function h

$$p(y_i|h, x_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - h(\mathbf{x}_i))^2}{2\sigma^2}}$$

Maximum Likelihood and least squares

Each example in our dataset D = $\{(x_i, y_i)\}$ is generated *independently* by this process

$$p(D|h) = \prod_{i=1}^{m} p(y_i, x_i|h) \propto \prod_{i=1}^{m} p(y_i|h, x_i)$$

Our goal is to find the most likely hypothesis

$$\begin{aligned} h_{ML} &= \arg\max_{h\in H} p(D|h) = \arg\max_{h\in H} \prod_{i=1}^{m} p(y_i|h, x_i) \\ &= \arg\max_{h\in H} \prod_{i=1}^{m} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - h(\mathbf{x}_i))^2}{2\sigma^2}} \\ &= \arg\max_{h\in H} \sum_{i=1}^{m} \log \frac{1}{\sigma\sqrt{2\pi}} - \frac{(y_i - h(\mathbf{x}_i))^2}{2\sigma^2} \\ &= \arg\max_{h\in H} -\sum_{i=1}^{m} \frac{(y_i - h(\mathbf{x}_i))^2}{2\sigma^2} \\ &= \arg\min_{h\in H} \sum_{i=1}^{m} (y_i - h(\mathbf{x}_i))^2 \end{aligned}$$

Because we assumed that the standard deviation is a constant.

If we consider the set of linear functions as our hypothesis space: $h(xi) = \boldsymbol{w}^T \boldsymbol{x}_i$

$$h_{ML} = \arg\min_{\mathbf{w}} \sum_{i=1}^{m} \left(y_i - \mathbf{w}^T \mathbf{x}_i\right)^2$$

This is the probabilistic explanation for least squares regression

Loss minimization perspective

We want to minimize the difference between the squared loss error of our prediction

Minimize the total squared loss

Bayesian perspective

We believe that the errors are Normally distributed with zero mean and a fixed variance

Find the linear regressor using the maximum likelihood principle

Linear regression: Two perspectives

Parameter Estimation via MLE and MAP

- Since data is assumed to be independently & identically distributed(i.i.d.), we can write down its total probability as $p(\boldsymbol{y}|\theta) = p(y_1, y_2, \ldots, y_N|\theta) = \prod_{n=1}^{N} p(y_n|\theta)$
- $p(\boldsymbol{y}|\theta)$ called "likelihood" - probability of observed data as a function

$$LL(\theta) = \log p(\boldsymbol{y}|\theta) = \log \prod_{n=1}^{N} p(y_n|\theta) = \sum_{n=1}^{N} \log p(y_n|\theta)$$

$$\theta_{MLE} = \operatorname{argmin}_{\theta} NLL(\theta)$$

$$\theta_{MLE} = \operatorname{argmax}_{\theta} LL(\theta) = \operatorname{argmax}_{\theta} \sum_{n=1}^{N} \log p(y_n|\theta)$$

$$\theta_{MLE} = \operatorname{argmax}_{\theta} \sum_{n=1}^{N} \log p(y_n|\theta) = \operatorname{argmin}_{\theta} \sum_{n=1}^{N} -\log p(y_n|\theta)$$

MAP: $$\theta_{MAP} = \arg\max_{\theta} p(\theta|y) = \arg\max_{\theta} \log p(\theta|y) = \arg\max_{\theta} \log \frac{p(\theta)p(\boldsymbol{y}|\theta)}{p(\boldsymbol{y})}$$

- Since $p(y)$ is constant w.r.t. θ, the above simplifies to

$$\theta_{MAP} = \arg\max_{\theta} [\log p(y|\theta) + \log p(\theta)]$$

$$= \arg\min_{\theta} [-\log p(y|\theta) - \log p(\theta)]$$

$$\theta_{MAP} = \arg\min_{\theta} [NLL(\theta) - \log p(\theta)]$$

MLE: An Example

- Consider a sequence of N coin toss outcomes (observations)
- Each observation y_n is a binary random variable. Head: $y_n = 1$, Tail: $y_n = 0$
- Each y_n is assumed generated by a **Bernoulli distribution** with param $\theta \in (0,1)$

$$p(y_n|\theta) = \text{Bernoulli}(y_n|\theta) = \theta^{y_n}(1-\theta)^{1-y_n}$$

- Here θ the unknown param (probability of head). Want to estimate it using MLE
- Log-likelihood: $\sum_{n=1}^{N} \log p(y_n|\theta) = \sum_{n=1}^{N} [y_n \log \theta + (1-y_n)\log(1-\theta)]$
- Maximizing log-lik (or minimizing NLL) w.r.t. θ will give a closed form expression

$$\theta_{MLE} = \frac{\sum_{n=1}^{N} y_n}{N}$$

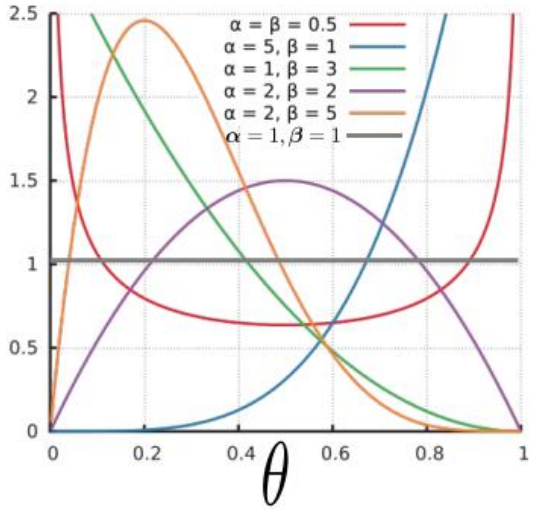
MAP Estimation: An Example

- Let's again consider the coin-toss problem (estimating the bias of the coin)
- Each likelihood term is Bernoulli

$$p(y_n|\theta) = \text{Bernoulli}(y_n|\theta) = \theta^{y_n}(1-\theta)^{1-y_n}$$

- Also need a prior since we want to do MAP estimation
- Since $\theta \in (0,1)$, a reasonable choice of prior for θ would be

$p(\theta)$



$$p(\theta|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \qquad LP(\theta) = \sum_{n=1}^{N} \log p(y_n|\theta) + \log p(\theta|\alpha,\beta)$$

$$LP(\theta) = \sum_{n=1}^{N} [y_n \log \theta + (1-y_n)\log(1-\theta)] + (\alpha-1)\log\theta + (\beta-1)\log(1-\theta)$$

- Maximizing the above log post. (or min. of its negative) w.r.t. θ gives

$$\theta_{MAP} = \frac{\sum_{n=1}^{N} y_n + \alpha - 1}{N + \alpha + \beta - 2}$$

The Naïve Bayes Classifier

MAP prediction

Don't confuse with *MAP learning*: finds hypothesis by

$$h_{MAP} = \arg\max_{h \in H} P(D|h)P(h)$$

- We have seen Bayesian learning
  - Using a probabilistic criterion to select a hypothesis
  - Maximum a posteriori and maximum likelihood learning
- We could also learn functions that predict probabilities of outcomes
- Different from using a probabilistic criterion to learn

  Maximum a posteriori (MAP) prediction as opposed to MAP learning
- Using the Bayes rule for predicting y given an input $\mathbf{x}$

Posterior probability of label being y for this input x

$$P(Y = y \mid X = \mathbf{x}) = \frac{P(X = \mathbf{x} \mid Y = y)P(Y = y)}{P(X = \mathbf{x})}$$

- Predict the label y for the input $\mathbf{x}$ using

$$\operatorname{argmax}_y \frac{P(X = \mathbf{x} \mid Y = y)P(Y = y)}{P(X = \mathbf{x})}$$

$$\operatorname{argmax}_y P(X = \mathbf{x} \mid Y = y)P(Y = y)$$

Likelihood of observing this input $\mathbf{x}$ when the label is y

Prior probability of the label being y

How hard is it to learn probabilistic models?

Prior P(Y)

- If there are k labels, then k – 1 parameters

Likelihood P(**X** | Y)

- If there are d Boolean features:
  - We need a value for each possible $P(x_1, x_2, .., x_d \mid y)$ for each y
  - $k(2^d - 1)$ parameters

Need a lot of data to estimate these many numbers!

High model complexity

If there is very limited data, high variance in the parameters

How can we deal with this?

Answer: Make independence assumptions

Conditional independence

- Suppose X, Y and Z are random variables

  X is *conditionally independent* of Y given Z if the probability distribution of X is independent of the value of Y when Z is observed $P(X|Y,Z) = P(X|Z)$

  Or equivalently $P(X,Y|Z) = P(X|Z)P(Y|Z)$

The Naïve Bayes Classifier

- $P(x_1, x_2, \cdots, x_d|y)$ required k($2^d - 1$) parameters
- What if *all the features were conditionally independent given the label*?

The Naïve Bayes Assumption

That is,

$$P(x_1, x_2, \cdots, x_d|y) = P(x_1|y)P(x_2|y)\cdots P(x_d|y)$$

- Requires only d numbers for each label. kd parameters overall. Not bad!

❑ Assumption: Features are conditionally independent given the label Y

❖ To predict, we need two sets of probabilities

– Prior P(y)

For each x_j, we have the likelihood P(x_j | y)

❑ Decision rule:

$$h_{NB}(\boldsymbol{x}) = \underset{y}{\operatorname{argmax}} P(y)P(x_1, x_2, \cdots, x_d|y)$$

$$= \underset{y}{\operatorname{argmax}} P(y)\prod_j P(x_j|y)$$

Maximum likelihood estimation for Naïve Bayes:

$$h_{ML} = \arg\max_{h \in H} P(D|h)$$

Given a dataset D = $\{(x_i, y_i)\}$ with m examples

Let MLE be a probabilistic criterion to select the hypothesis

$$h_{ML} = \arg\max_h \prod_{i=1}^{m} P((\mathbf{x}_i, y_i)|h)$$

$$= \arg\max_h \prod_{i=1}^{m} P(\mathbf{x}_i|y_i, h) P(y_i|h)$$

$$= \arg\max_h \prod_{i=1}^{m} P(y_i|h) \prod_j P(x_{i,j}|y_i, h)$$

$\mathbf{x}_{ij}$ is the j[th] feature of $\mathbf{x}_i$

What next?

We need to make a modeling assumption about the functional form of these probability distributions

The Naïve Bayes assumption

$$h_{ML} = \arg\max_h \sum_{i=1}^{m} \log P(y_i|h) + \sum_i \sum_j \log P(x_{i,j}|y_i, h)$$

naive_bayes.BernoulliNB(*[, alpha, ...])	Naive Bayes classifier for multivariate Bernoulli models.
naive_bayes.CategoricalNB(*[, alpha, ...])	Naive Bayes classifier for categorical features.
naive_bayes.ComplementNB(*[, alpha, ...])	The Complement Naive Bayes classifier described in Rennie et al. (2003).
naive_bayes.GaussianNB(*[, priors, ...])	Gaussian Naive Bayes (GaussianNB).
naive_bayes.MultinomialNB(*[, alpha, ...])	Naive Bayes classifier for multinomial models.

Logistics Regression

Logistic Regression (LR)

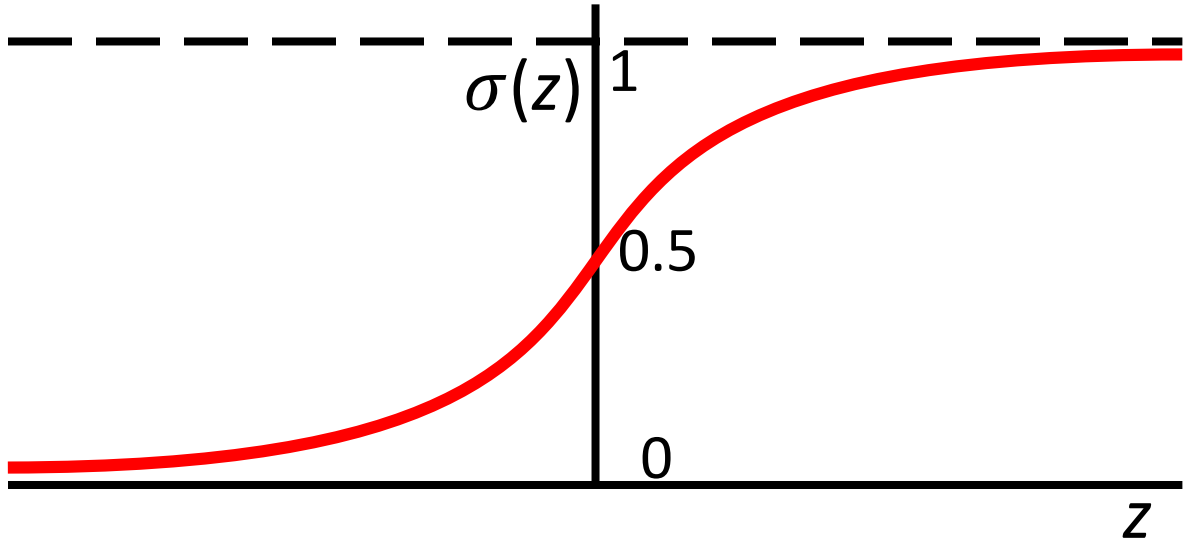
The word "regression" is a misnomer. Both are classification models

- A probabilistic model for binary classification
- Multi-class extension known as "softmax regression". Both very widely used
- Learns the Probability Mass Function(PMF) of the output label given the input, i.e., $p(y|\boldsymbol{x})$
- Uses the sigmoid function to define the conditional probability of y being 1

$$\mu_x = p(y = 1|\boldsymbol{w}, \boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x})$$

$$= \frac{1}{1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x})}$$

$$= \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x})}{1 + \exp(\boldsymbol{w}^\top \boldsymbol{x})}$$



- Here $\boldsymbol{w}^\top \boldsymbol{x}$ is the score for input $\boldsymbol{x}$. The sigmoid turns it into a probability
- We can learn $\mathbf{w}$ by using MLE or MAP estimation

LR: Decision Boundary

- At the decision boundary where both classes are equiprobable

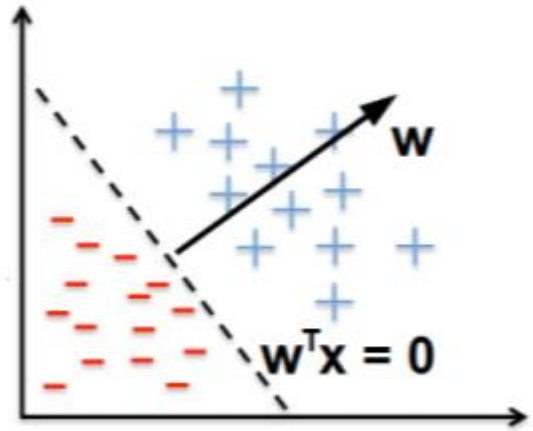
$$p(y=1|\boldsymbol{x},\boldsymbol{w}) = p(y=0|\boldsymbol{x},\boldsymbol{w})$$

$$\frac{\exp(\boldsymbol{w}^\top \boldsymbol{x})}{1+\exp(\boldsymbol{w}^\top \boldsymbol{x})} = \frac{1}{1+\exp(\boldsymbol{w}^\top \boldsymbol{x})}$$

$$\exp(\boldsymbol{w}^\top \boldsymbol{x}) = 1$$

$$\boldsymbol{w}^\top \boldsymbol{x} = 0$$

A linear hyperplane



- Very large positive $\boldsymbol{w}^\top \boldsymbol{x}$ means $p(y=1|\boldsymbol{w},\boldsymbol{x})$ close to 1
- Very large negative $\boldsymbol{w}^\top \boldsymbol{x}$ means $p(y=0|\boldsymbol{w},\boldsymbol{x})$ close to 1
- At decision boundary, $\boldsymbol{w}^\top \boldsymbol{x} = 0$ implies $p(y=1|\boldsymbol{w},\boldsymbol{x}) = p(y=0|\boldsymbol{w},\boldsymbol{x}) = 0.5$

Multiclass Logistic (a.k.a. Softmax) Regression

- Also called multinoulli/multinomial regression: Basically, LR for $K > 2$ classes
- In this case, $y_n \in \{1,2,\ldots,K\}$ and label probabilities are defined as

$$p(y_n = k|\boldsymbol{x}_n, \boldsymbol{W}) = \frac{\exp(\boldsymbol{w}_k^\top \boldsymbol{x}_n)}{\sum_{\ell=1}^{K} \exp(\boldsymbol{w}_\ell^\top \boldsymbol{x}_n)} = \mu_{nk}$$

Softmax function

Also note that $\sum_{\ell=1}^{K} \mu_{n\ell} = 1$ for any input $\boldsymbol{x}_n$

- K weight vecs $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K$ (one per class), each D-dim, and $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$
- Each likelihood $p(y_n|\boldsymbol{x}_n, \boldsymbol{W})$ is a multinoulli distribution. Therefore total likelihood

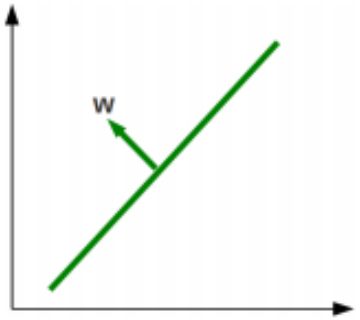
$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{W}) = \prod_{n=1}^{N} \prod_{\ell=1}^{K} \mu_{n\ell}^{y_{n\ell}}$$

Notation: $y_{n\ell} = 1$ if true class of $\boldsymbol{x}_n$ is ℓ and $y_{n\ell'} = 0 \; \forall \ell' \neq \ell$

- Can do MLE/MAP for $\boldsymbol{W}$ similar to LR model

Hyperplane

- Separates a D-dimensional space into two **half-spaces** (positive and negative)
- Defined by a normal vector $\boldsymbol{w} \in \mathbb{R}^D$ (pointing towards positive half-space)



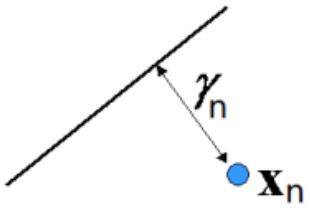
$b > 0$ means moving $\boldsymbol{w}^\top \boldsymbol{x} = 0$ along the direction of $\boldsymbol{w}$; $b < 0$ means in opp. dir.

$$\boldsymbol{w}^\top \boldsymbol{x} + b = 0$$

- Equation of the hyperplane: $\boldsymbol{w}^\top \boldsymbol{x} = 0$
- Assumption: The hyperplane passes through origin. If not, add a bias term b
- Distance of a point $\boldsymbol{x_n}$ from a hyperplane $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$

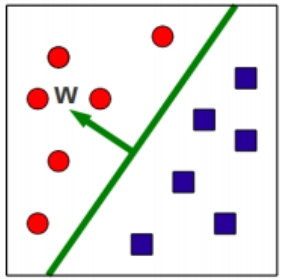
Can be positive or negative

$$\gamma_n = \frac{\boldsymbol{w}^T \boldsymbol{x}_n + b}{||\boldsymbol{w}||}$$



Hyperplane based (binary) classification

- Basic idea: Learn to separate two classes by a hyperplane $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$



Prediction Rule

$$y_* = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_* + b)$$

For multi-class classification with hyperplanes, there will be multiple hyperplanes (e.g., one for each pair of classes)

- The hyperplane may be "implied" by the model, or learned directly
  - Implied: Prototype-based classification, nearest neighbors, etc
  - Directly learned: Logistic regression, Perceptron, Support Vector Machine (SVM), etc
- The "direct" approach defines a model with params $\boldsymbol{w}$ (and optionally a bias param b)
  - The parameters are learned by optimizing a classification loss function (will see examples)
  - These are also discriminative approaches – $\boldsymbol{x}$ is not modeled but treated as fixed (given)
- The hyperplane need not be linear (e.g., can be made nonlinear using kernels; later)

Loss Functions for Classification

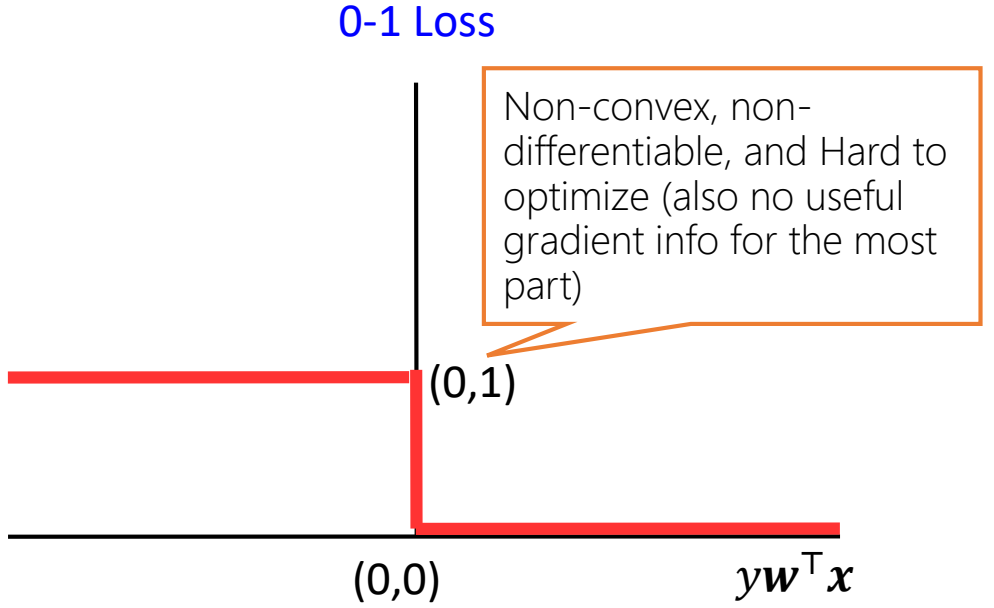
- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss fn

$$\ell(y, \hat{y}) = (y - \hat{y})^2 \qquad \ell(y, \hat{y}) = |y - \hat{y}|$$

- These measure the difference between the true output and model's prediction
- What about loss functions for classification where $\hat{y} = \mathbf{sign}(\boldsymbol{w}^\top \boldsymbol{x})$?
- Perhaps the most natural classification loss function would be a "0-1 Loss"
  - Loss = 1 if $\hat{y} \neq y$ and Loss = 0 if $\hat{y} = y$.
  - Assuming labels as +1/-1, it means

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

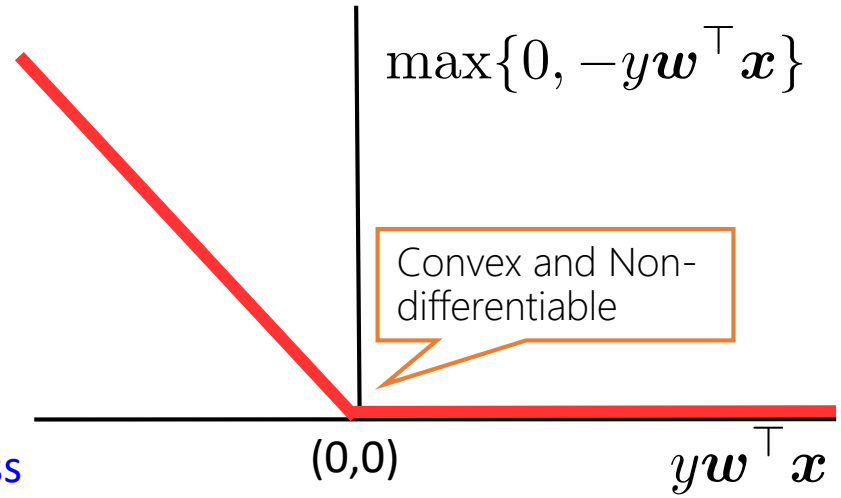
Same as $\mathbb{I}[y\boldsymbol{w}^\top \boldsymbol{x} < 0]$ or $\mathbb{I}[\mathbf{sign}(\boldsymbol{w}^\top \boldsymbol{x}) \neq y]$



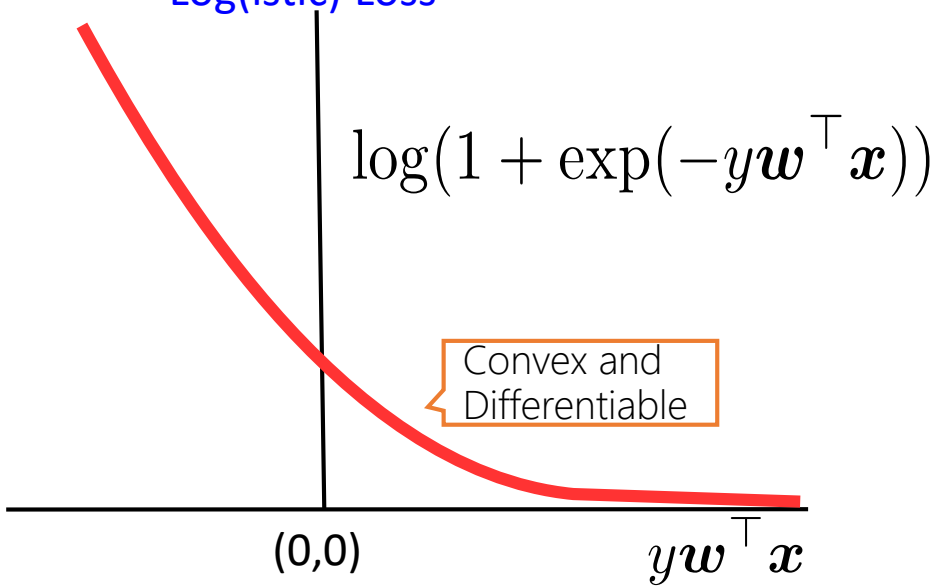
Loss Functions for Classification

- An ideal loss function for classification should be such that
  - Loss is small/zero if y and $\mathbf{sign}(\boldsymbol{w}^\top \boldsymbol{x})$ match
  - Loss is large/non-zero if y and $\mathbf{sign}(\boldsymbol{w}^\top \boldsymbol{x})$ do not match
  - Large positive $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ small/zero loss
  - Large negative $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ large/non-zero loss

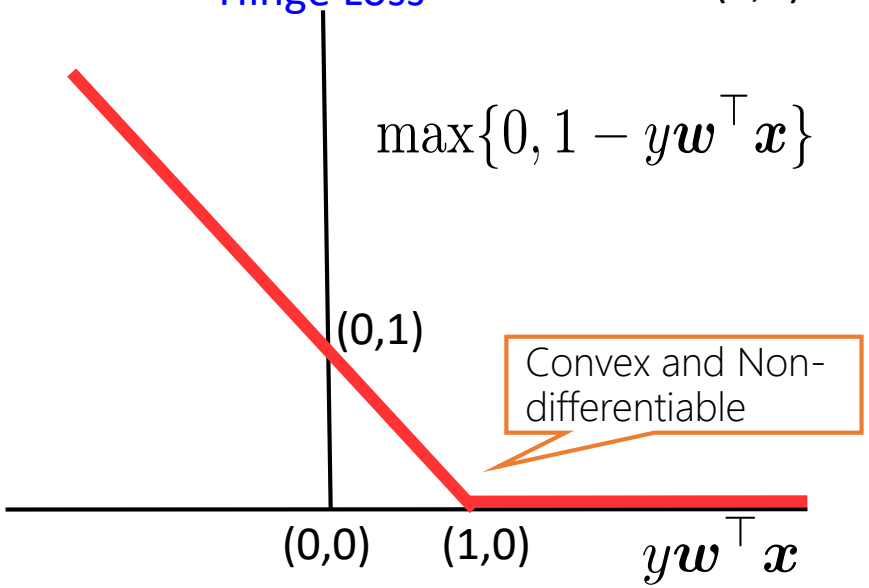
"Perceptron" Loss



Log(istic) Loss

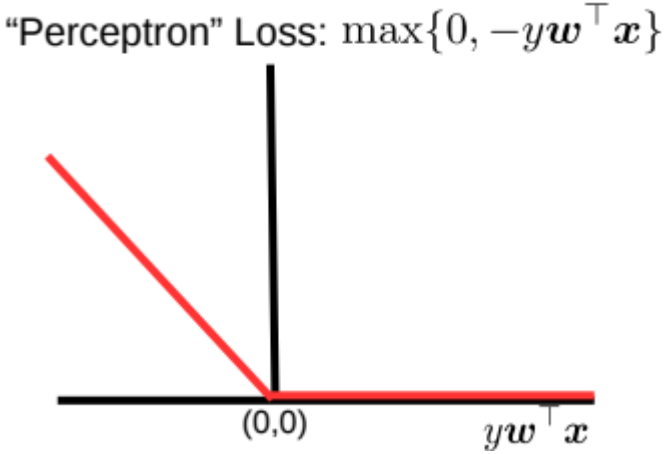


Hinge Loss



Learning by Optimizing Perceptron Loss

- Let's ignore the bias term b for now. So the hyperplane is simply $\boldsymbol{w}^\top \boldsymbol{x} = 0$
- The Perceptron loss function: $L(w) = \sum_{n=1}^{N} \max\{0, -y_n \boldsymbol{w}^\top \boldsymbol{x}_n\}$. Let's do SGD



Subgradients(must be convex) w.r.t. $\boldsymbol{w}$

One randomly chosen example in each iteration

$$\boldsymbol{g}_n = \begin{cases} 0, & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 0 \\ -y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 0 \\ k y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n = 0 \end{cases} \quad (\text{where } k \in [-1, 0])$$

- If we use $k = 0$ then $\boldsymbol{g}_n = 0$ for $y_n \boldsymbol{w}^\top \boldsymbol{x}_n \geq 0$, and $\boldsymbol{g}_n = -y_n \boldsymbol{x}_n$ for $y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 0$
- Non-zero gradients only when the model makes a mistake on current example $(\boldsymbol{x}_n, y_n)$
- Thus SGD will update $\boldsymbol{w}$ only when there is a mistake (mistake-driven learning)

The Perceptron Algorithm

- Stochastic Sub-grad desc on Perceptron loss is also known as the Perceptron algorithm

Stochastic SubGD

1. Initialize $\boldsymbol{w} = \boldsymbol{w}^{(0)}$, $t = 0$, set $\eta_t = 1, \forall t$
2. Pick some $(\boldsymbol{x}_n, y_n)$ randomly.
3. If current $\boldsymbol{w}$ makes a mistake on $(\boldsymbol{x}_n, y_n)$, i.e., $y_n \boldsymbol{w}^{(t)\top} \boldsymbol{x}_n < 0$

$$\begin{aligned} \boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + y_n \boldsymbol{x}_n \\ t &= t + 1 \end{aligned}$$

4. If not converged, go to step 2.

Mistake condition

Updates are "corrective": If $y\_n = +1$ and $\boldsymbol{w}^\top \boldsymbol{x}_n < 0$, after the update $\boldsymbol{w}^\top \boldsymbol{x}_n$ will be less negative. Likewise, if $y_n = -1$ and $\boldsymbol{w}^\top \boldsymbol{x}_n > 0$, after the update $\boldsymbol{w}^\top \boldsymbol{x}_n$ will be less positive

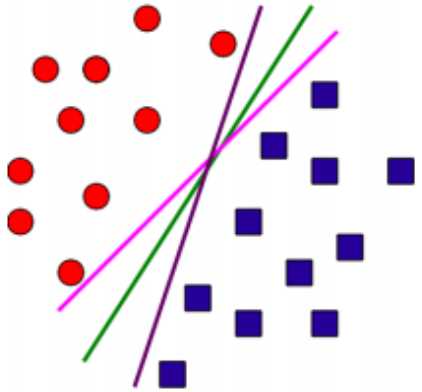
If training data is linearly separable, the Perceptron algo will converge in a finite number of iterations (Block & Novikoff theorem)

- Assuming $\boldsymbol{w}^{(0)} = \boldsymbol{0}$, easy to see that the final $\boldsymbol{w}$ has the form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$
  - α_n is total number of mistakes made by the algorithm on example $(\boldsymbol{x}_n, y_n)$
  - As we'll see, many other models also have weights $\boldsymbol{w}$ in the form $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$

Meaning of α_n may be different

Perceptron and (lack of) Margins

- Perceptron would learn a hyperplane (of many possible) that separates the classes



Basically, it will learn the hyperplane which corresponds to the $\boldsymbol{w}$ that minimizes the Perceptron loss

- Doesn't guarantee any "margin" around the hyperplane
  - The hyperplane can get arbitrarily close to some training example(s) on either side
  - This may not be good for generalization performance

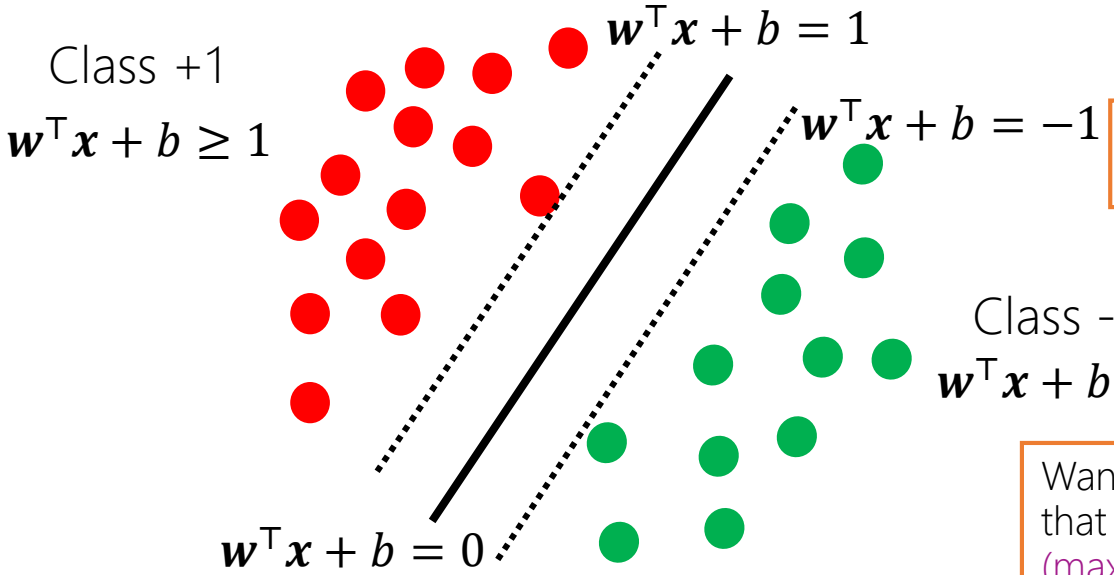
$\gamma > 0$ is some pre-specified margin

- Can artificially introduce margin by changing the mistake condition to $y_n \boldsymbol{w}^\top \boldsymbol{x}_n < \gamma$
- Support Vector Machine (SVM) does it directly by learning the max. margin hyperplane

Support Vector Machine (SVM)

SVM originally proposed by Vapnik and colleagues in early 90s

- Hyperplane based classifier. Ensures a large margin around the hyperplane
- Will assume a linear hyperplane to be of the form $\boldsymbol{w}^\top \boldsymbol{x} + b = 0$ (nonlinear ext. later)



$$\boldsymbol{w}^\top \boldsymbol{x}_n + b \geq 1 \quad \text{if} \quad y_n = +1$$
$$\boldsymbol{w}^\top \boldsymbol{x}_n + b \leq -1 \quad \text{if} \quad y_n = -1$$
$$y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1 \quad \forall n$$

Distance from the closest point (on either side)

"Margin" of the hyperplane

$$\gamma = \min_{1 \leq n \leq N} \frac{|\boldsymbol{w}^\top \boldsymbol{x}_n + b|}{\|\boldsymbol{w}\|} = \frac{1}{\|\boldsymbol{w}\|}$$

$$\text{Total margin} = \frac{2}{\|\boldsymbol{w}\|}$$

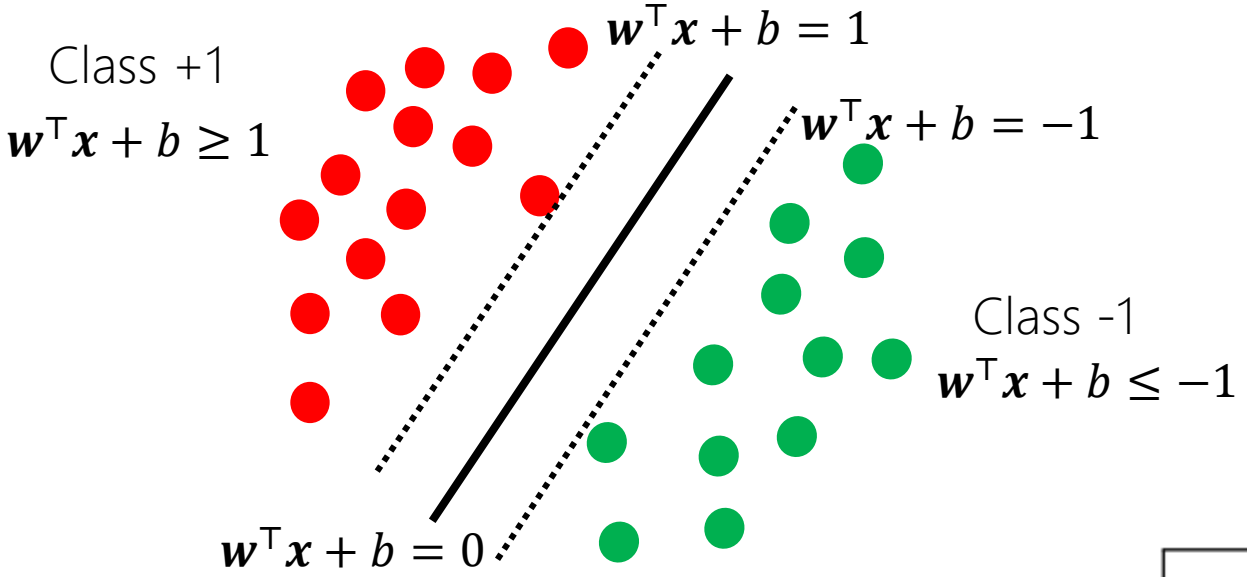
Want the hyperplane $(\boldsymbol{w}, b)$ such that this margin is maximized (max-margin hyperplane) and $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1 \quad \forall n$

- Two other "supporting" hyperplanes defining a "no man's land"
  - Ensure that zero training examples fall in this region (will relax later)
  - The SVM idea: Position the hyperplane s.t. this region is as "wide" as possible

The 1/-1 in supp. h.p. equations is arbitrary; can replace by any scalar m/-m and solution won't change, except a simple scaling of $\boldsymbol{w}$

Hard-Margin SVM

- Hard-Margin: Every training example must fulfil margin condition $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1$
- Meaning: Must not have any example in the no-man's land



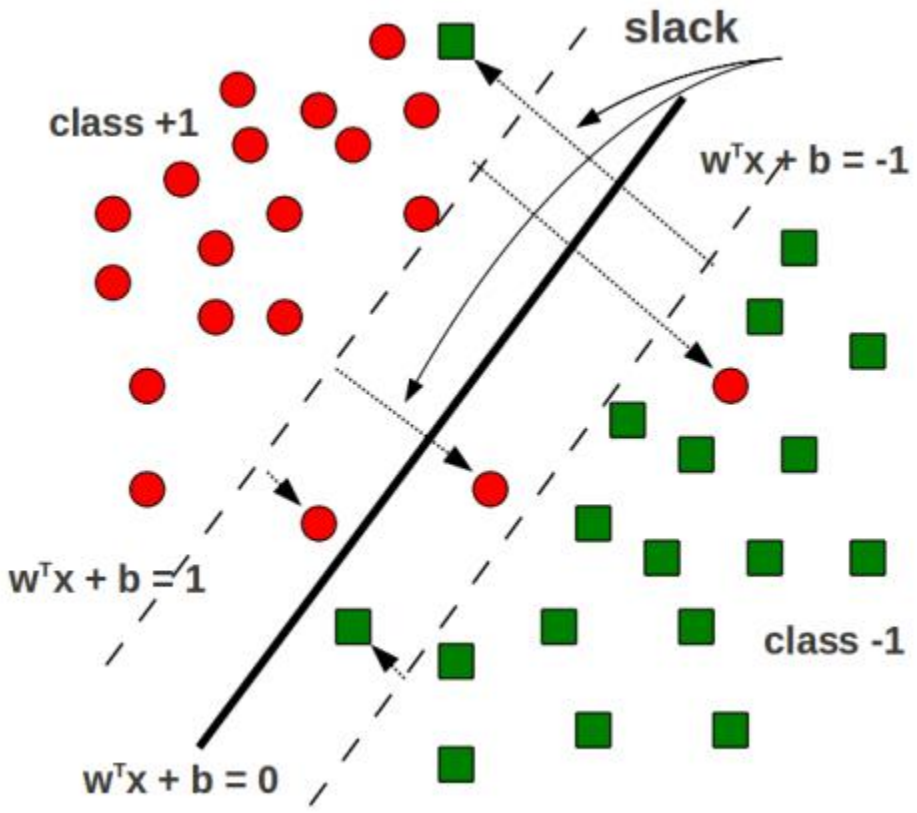
- Also want to maximize margin $2\gamma = \frac{2}{\|\boldsymbol{w}\|}$
- Equivalent to <u>minimizing</u> $\|\boldsymbol{w}\|^2$ or $\frac{\|\boldsymbol{w}\|^2}{2}$
- The objective func. for hard-margin SVM

Constrained optimization problem with N inequality constraints. Objective and constraints both are convex

$$\min_{\boldsymbol{w},b} \quad f(\boldsymbol{w}, b) = \frac{||\boldsymbol{w}||^2}{2}$$

$$\text{subject to} \quad y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1, \qquad n = 1, \ldots, N$$

Soft-Margin SVM (More Commonly Used)



- Allow some training examples to fall within the no-man's land (margin region)
- Even okay for some training examples to fall totally on the wrong side of h.p.
- Extent of "violation" by a training input $(\boldsymbol{x}_n, y_n)$ is known as slack $\xi_n \geq 0$
- $\xi_n > 1$ means totally on the wrong side

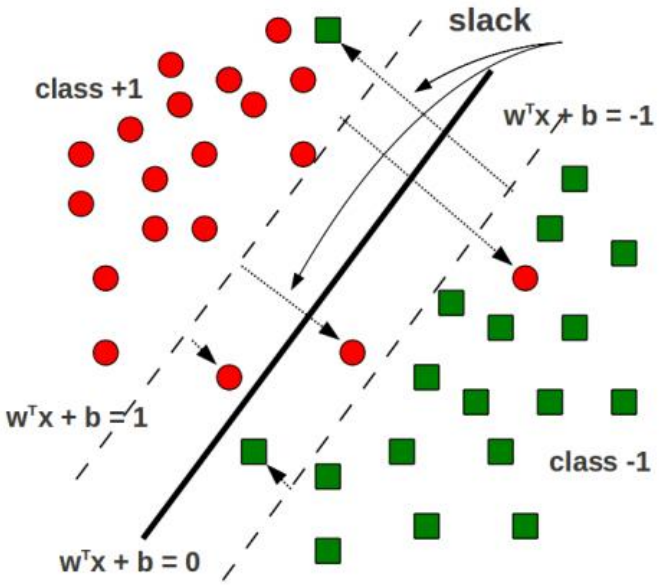
$$\boldsymbol{w}^\top \boldsymbol{x}_n + b \geq 1 - \xi_n \quad \text{if} \quad y_n = +1$$

$$\boldsymbol{w}^\top \boldsymbol{x}_n + b \leq -1 + \xi_n \quad \text{if} \quad y_n = -1$$

Soft-margin constraint: $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1 - \xi_n \quad \forall n$

Soft-Margin SVM (Contd)

- Goal: Still want to maximize the margin such that
  - Soft-margin constraints $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) \geq 1 - \xi_n$ are satisfied for all training ex.
  - Do not have too many margin violations (sum of slacks $\sum_{n=1}^{N} \xi_n$ should be small)



- The objective func. for soft-margin SVM

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \; f(\boldsymbol{w}, b, \boldsymbol{\xi}) = \frac{||\boldsymbol{w}||^2}{2} + C\sum_{n=1}^{N} \xi_n$$

$$\text{subject to} \quad y_n(\boldsymbol{w}^T \boldsymbol{x}_n + b) \geq 1 - \xi_n, \quad \xi_n \geq 0 \quad n = 1, \ldots, N$$

Trade-off hyperparam

training error

Constrained optimization problem with $2N$ inequality constraints. Objective and constraints both are convex

- Hyperparameter C controls the trade off between large margin and small training error (need to tune)
  - Large C: small training error but also small margin (bad)
  - Small C: large margin but large training error (bad)

Solving the SVM Problem

Solving Hard-Margin SVM

- The hard-margin SVM optimization problem is

$$\min_{\boldsymbol{w},b} \quad f(\boldsymbol{w},b) = \frac{||\boldsymbol{w}||^2}{2}$$
$$\text{subject to} \quad 1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) \leq 0, \qquad n = 1,\ldots,N$$

- A constrained optimization problem. One option is to solve using Lagrange's method
- Introduce Lagrange multipliers α_n $(n = 1,\ldots,N)$, one for each constraint, and solve

$$\min_{\boldsymbol{w},b} \max_{\boldsymbol{\alpha}\geq 0} \quad \mathcal{L}(\boldsymbol{w},b,\boldsymbol{\alpha}) = \frac{||\boldsymbol{w}||^2}{2} + \sum_{n=1}^{N} \alpha_n\{1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b)\}$$

- $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_N]$ denotes the vector of Lagrange multipliers
- It is easier to solve the dual: min and then max

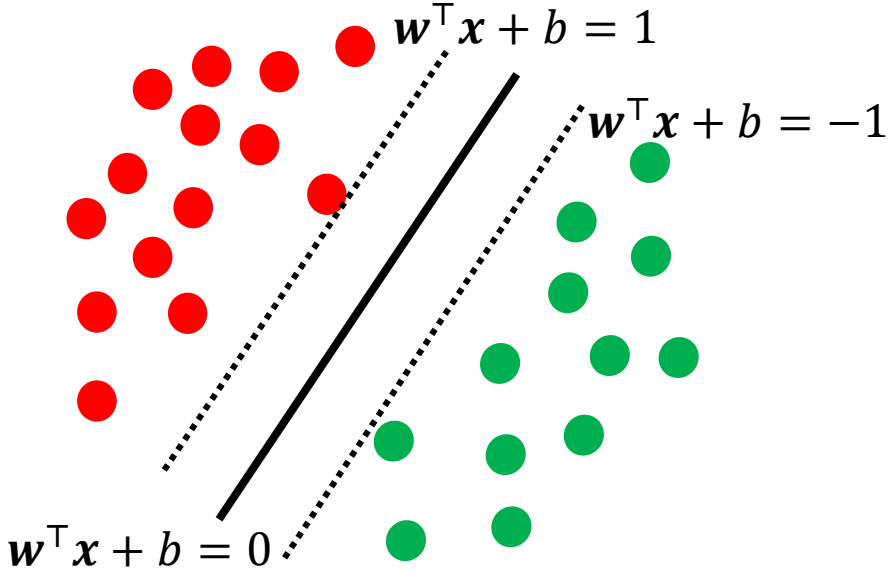
Solving Hard-Margin SVM

- One we have the α_n' s by solving the dual, we can get $\boldsymbol{w}$ and b as

$$\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n \quad \text{(we already saw this)}$$

$$b = -\frac{1}{2}\left(\min_{n:y_n=+1} \boldsymbol{w}^T \boldsymbol{x}_n + \max_{n:y_n=-1} \boldsymbol{w}^T \boldsymbol{x}_n\right)$$

- A nice property: Most α_n' s in the solution will be zero (sparse solution)



- Reason: Karush Kuhn Tucker(KKT) conditions
- For the optimal α_n' s, we must have
- Thus α_n nonzero only if $y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b) = 1$, i.e., the training example lies on the boundary

$$\alpha_n\{1 - y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b)\} = 0$$

- These examples are called support vectors

Solving Soft-Margin SVM

- Recall the soft-margin SVM optimization problem

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad f(\boldsymbol{w},b,\boldsymbol{\xi}) = \frac{||\boldsymbol{w}||^2}{2} + C\sum_{n=1}^{N}\xi_n$$

$$\text{subject to} \quad 1 \le y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) + \xi_n, \quad -\xi_n \le 0 \qquad n = 1,\ldots,N$$

Slack variables are introduced to allow certain constraints to be violated. That is, certain train- ing points will be allowed to be within the margin. We want the number of points within the margin to be as small as possible

- Here $\boldsymbol{\xi} = [\xi_1, \xi_2, \ldots, \xi_N]$ is the vector of slack variables
- Introduce Lagrange multipliers α_n, β_n for each constraint and solve Lagrangian

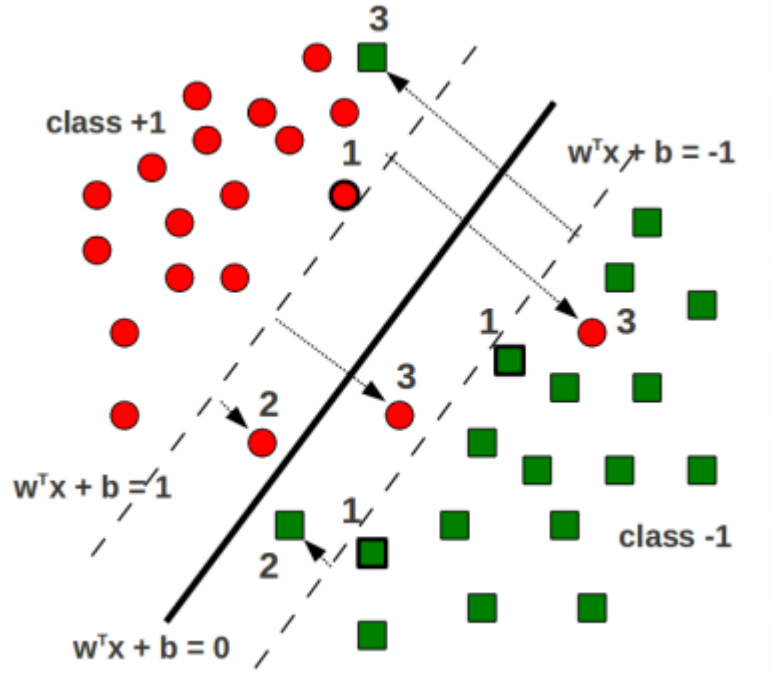
$$\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \max_{\boldsymbol{\alpha}\ge 0,\boldsymbol{\beta}\ge 0} \mathcal{L}(\boldsymbol{w},b,\xi,\alpha,\beta) = \frac{||\boldsymbol{w}||^2}{2} + +C\sum_{n=1}^{N}\xi_n + \sum_{n=1}^{N}\alpha_n\{1 - y_n(\boldsymbol{w}^T\boldsymbol{x}_n + b) - \xi_n\} - \sum_{n=1}^{N}\beta_n\xi_n$$

- The terms in red color above were not present in the hard-margin SVM
- Two set of dual variables $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \ldots, \alpha_N]$ and $\boldsymbol{\beta} = [\beta_1, \beta_2, \ldots, \beta_N]$
- Will eliminate the primal var $\boldsymbol{w}$, b, $\boldsymbol{\xi}$ to get dual problem containing the dual variables

Support Vectors in Soft-Margin SVM

- The hard-margin SVM solution had only one type of support vectors
  - All lied on the supporting hyperplanes $\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n + b = 1$ and $\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n + b = -1$

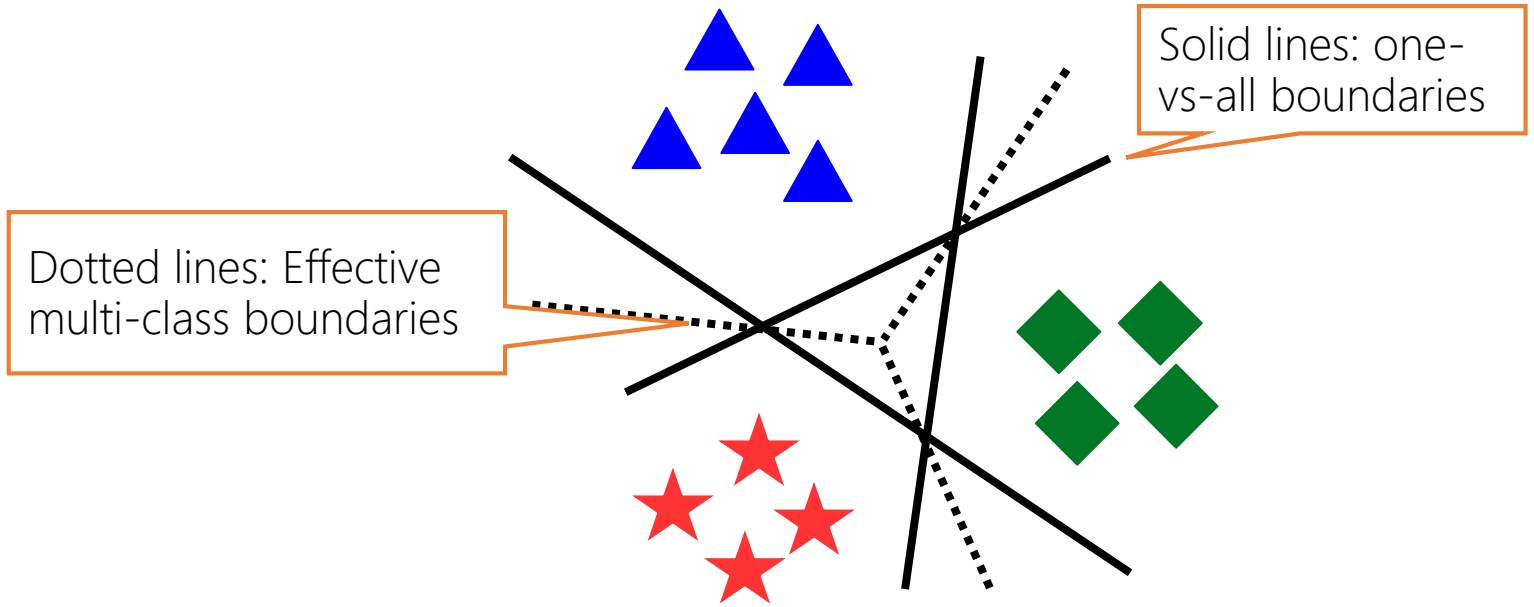
- The soft-margin SVM solution has <u>three</u> types of support vectors (with nonzero α_n)



1. Lying on the supporting hyperplanes
2. Lying within the margin region but still on the correct side of the hyperplane
3. Lying on the wrong side of the hyperplane (misclassified training examples)

Multi-class SVM using Binary SVM

- Can use binary classifiers to solve multiclass problems
- One-vs-All (also called One-vs-Rest): Construct K binary classification problems



- All-Pairs(One-vs-One): Learn K-choose-2 binary classifiers, one for each pair of classes (j, k)

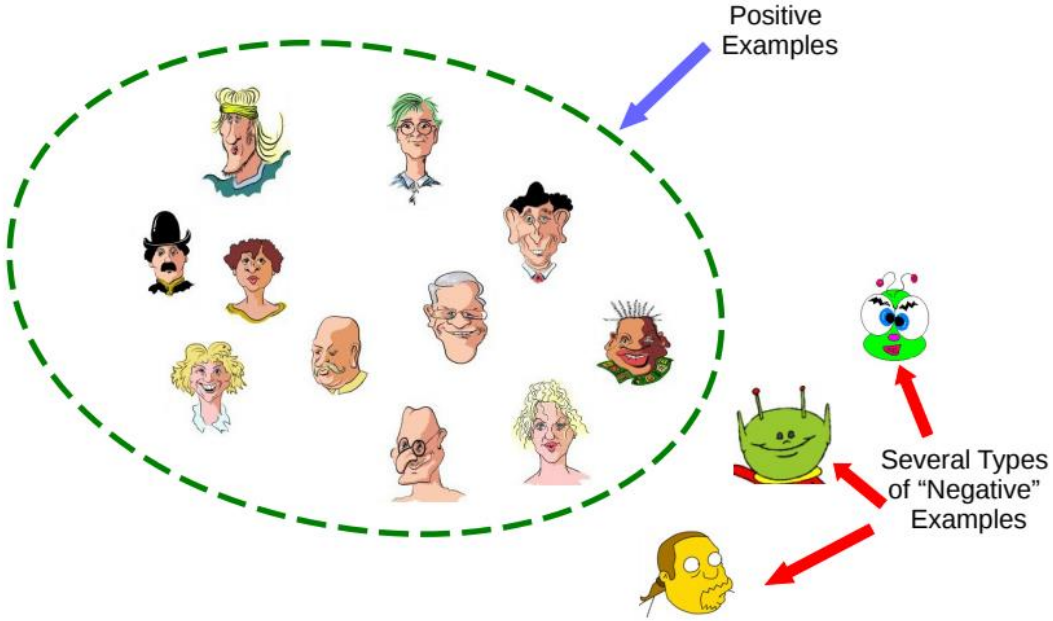
$$y_* = \arg\max_k \sum_{j \neq k} \boldsymbol{w}_{j,k}^\top \boldsymbol{x}_*$$

Weight vector of the pairwise classifier for class j and k

Positive score if class k wins over class j in pairwise comparison

One-class Classification

- Can we learn from examples of just one class, say positive examples?
- May be desirable if there are many types of negative examples



"Outlier/Novelty Detection" problems can also be formulated like this(Credit Card fraud)

- One-class classification is an approach to learn using only one class of examples
- One-class Classification via SVM-type Methods: 1) "Support Vector Data Description" (SVDD) [Tax and Duin, 2004]. 2) "One-Class SVM" (OC-SVM) [Schölkopf et al., 2001]

SVM: Summary

- A hugely (perhaps the most!) popular classification algorithm
- Reasonably mature, highly optimized SVM softwares freely available (perhaps the reason why it is more popular than various other competing algorithms)
- Some popular ones: libSVM, LIBLINEAR, sklearn also provides SVM
- Lots of work on scaling up SVMs* (both large N and large D)
- Extensions beyond binary classification (e.g., multiclass,)
- Can even be used for regression problems (Support Vector Regression(SVR))
- Nonlinear extensions possible via kernels

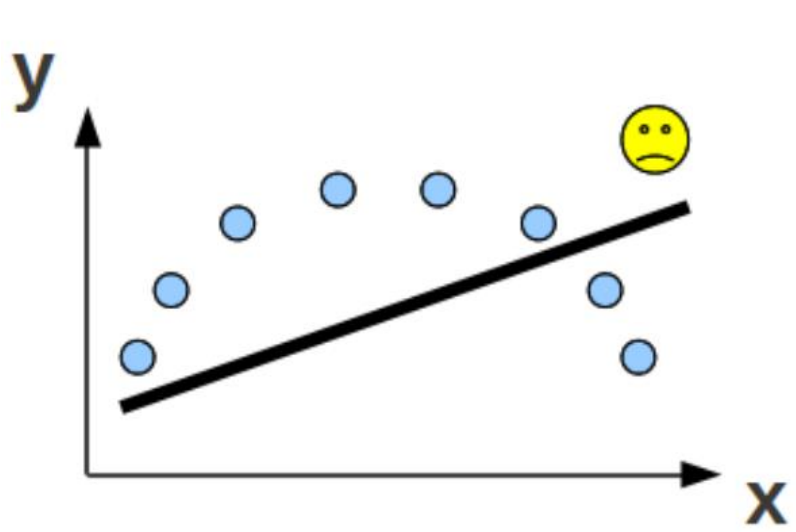
svm.LinearSVC([penalty, loss, dual, tol, C, ...])	Linear Support Vector Classification.
svm.LinearSVR(*[, epsilon, tol, C, loss, ...])	Linear Support Vector Regression.
svm.NuSVC(*[, nu, kernel, degree, gamma, ...])	Nu-Support Vector Classification.
svm.NuSVR(*[, nu, C, kernel, degree, gamma, ...])	Nu Support Vector Regression.
svm.OneClassSVM(*[, kernel, degree, gamma, ...])	Unsupervised Outlier Detection.
svm.SVC(*[, C, kernel, degree, gamma, ...])	C-Support Vector Classification.
svm.SVR(*[, kernel, degree, gamma, coef0, ...])	Epsilon-Support Vector Regression.

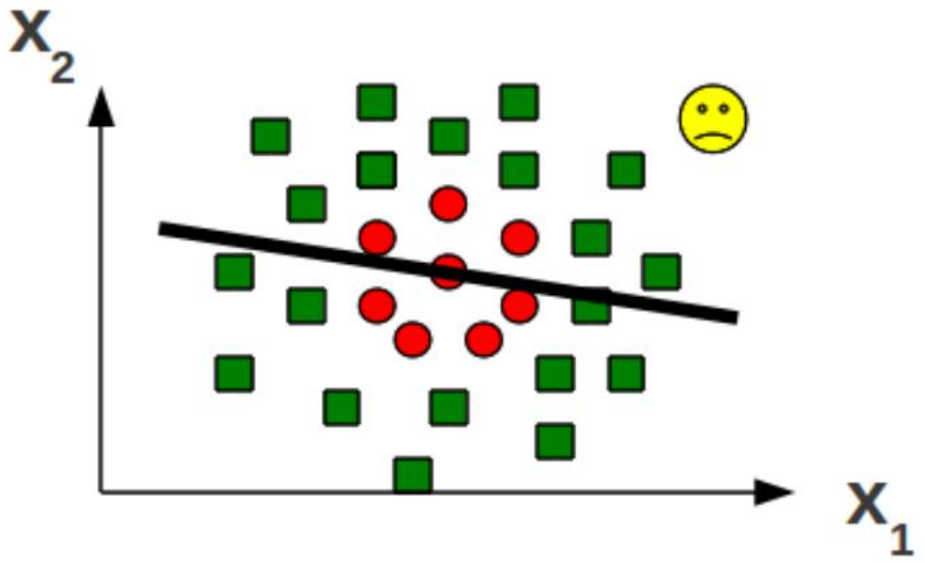
https://towardsdatascience.com/unlocking-the-true-power-of-support-vector-regression-847fd123a4a0

Kernel methods and nonlinear SVM via kernels

Linear Models

- Nice and interpretable but can’ t learn “difficult” nonlinear patterns





- So, are linear models useless for such problems?

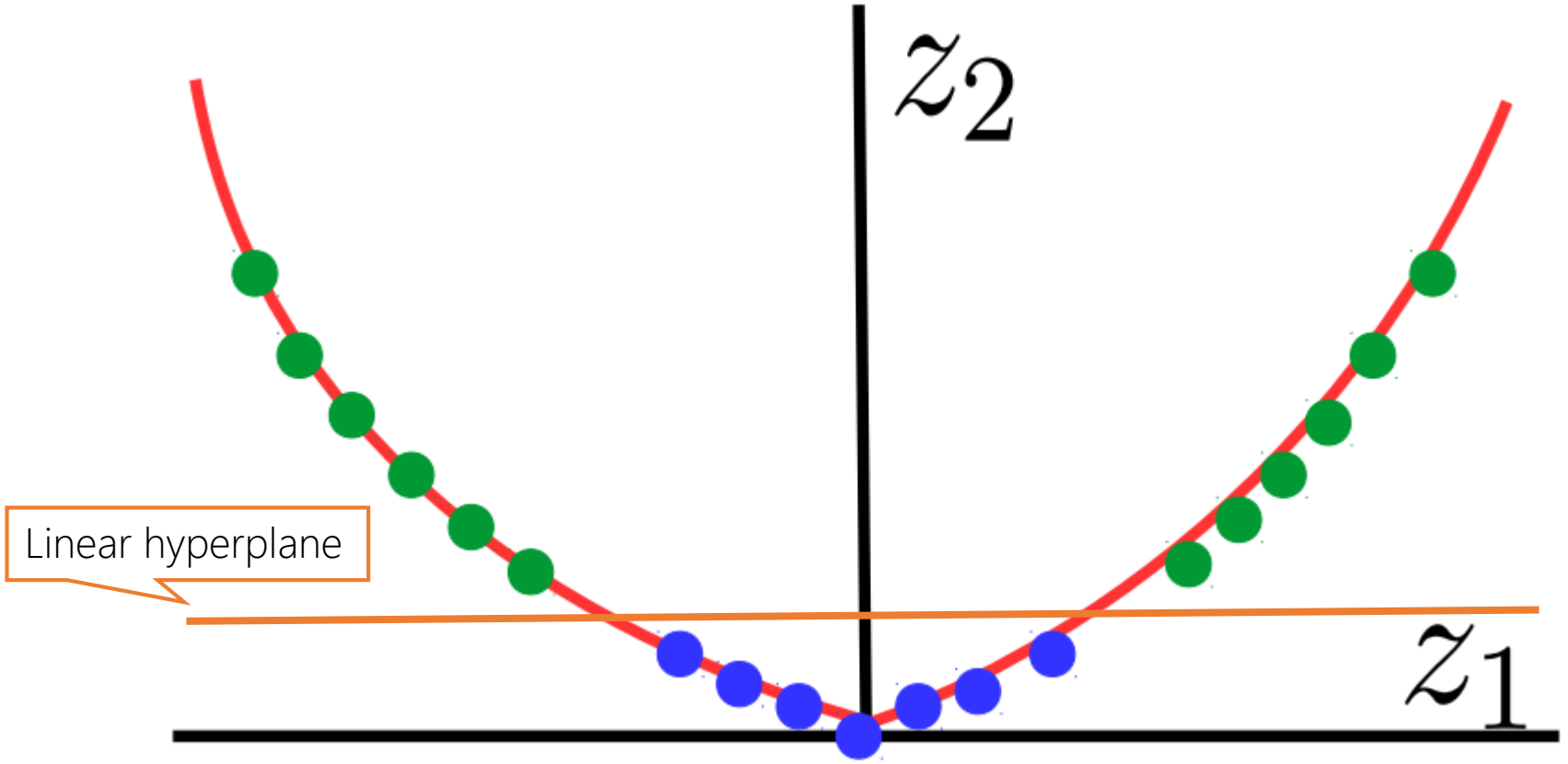
Linear Models for Nonlinear Problems

- Consider the following one-dimensional inputs from two classes

- Can’ t separate using a linear hyperplane

Linear Models for Nonlinear Problems

- Consider mapping each x to two-dimensions as $x \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, x^2]$



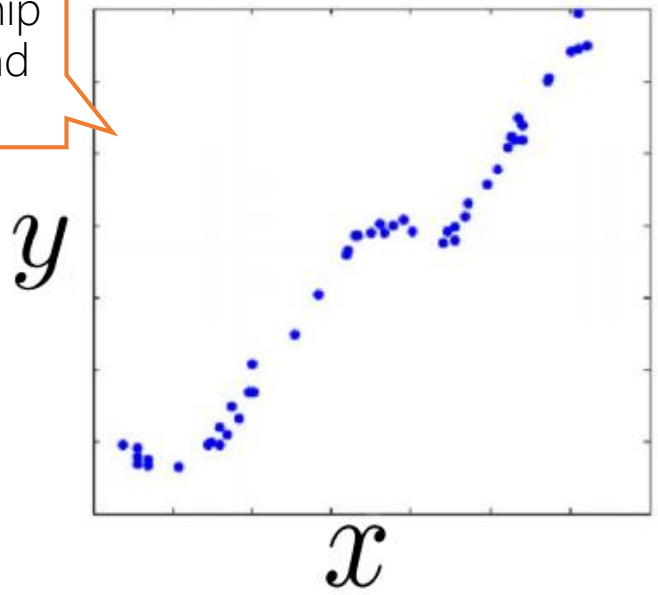
- Classes are now linearly separable in the two-dimensional space

Linear Models for Nonlinear Problems

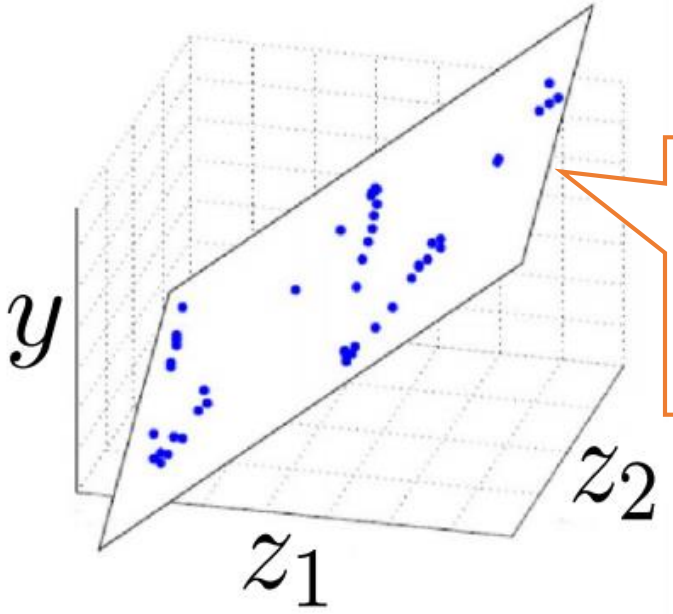
- The same idea can be applied for nonlinear regression as well

Not a linear relationship between inputs (x) and outputs (y)

A linear regression model will not work well



$$x \rightarrow \boldsymbol{z} = [z_1, z_2] = [x, \cos(x)]$$



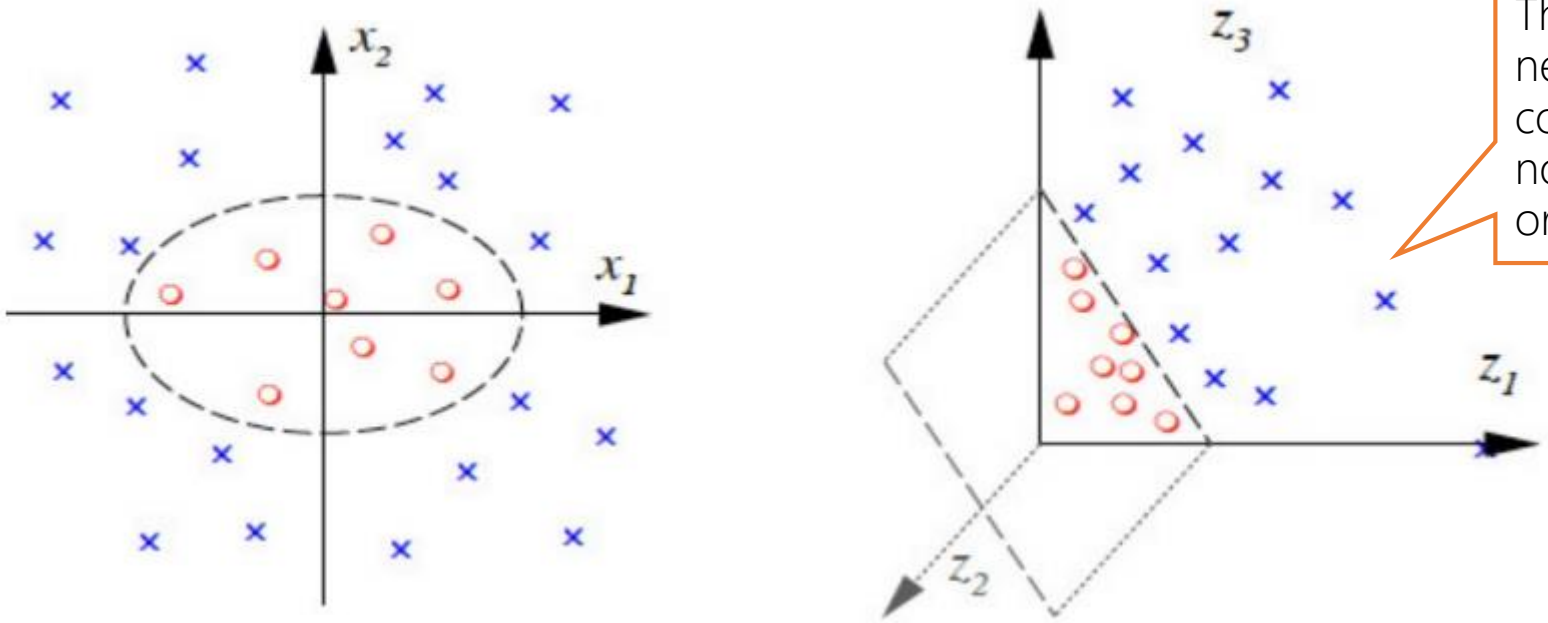
A linear regression model will work well with this new two-dim representation of the original one-dim inputs

Linear Models for Nonlinear Problems

- Can assume a feature mapping ϕ that maps/transforms the inputs to a "nice" space

$$\phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$$

$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)} x_1 x_2, x_2^2)$$

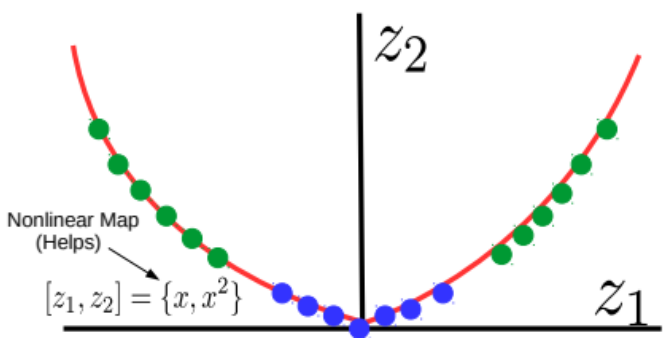


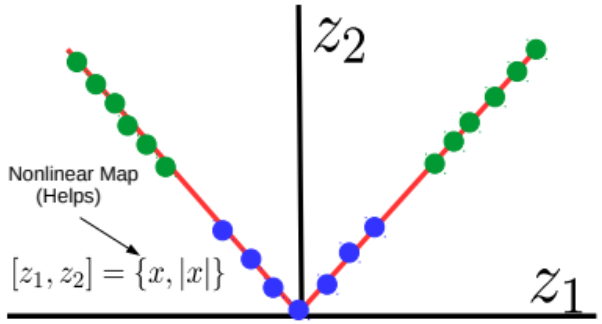
The linear model in the new feature space corresponds to a nonlinear model in the original feature space

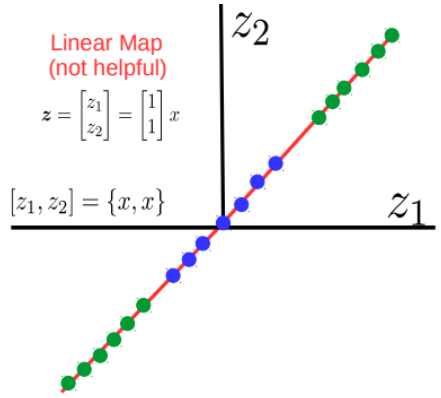
- .. and then happily apply a linear model in the new space!

Not Every Mapping is Helpful

- Not every higher-dim mapping helps in learning nonlinear patterns
- Must be a nonlinear mapping
- For the nonlin classfn problem we saw earlier, consider some possible mappings







How to get these "good" (nonlinear) mappings?

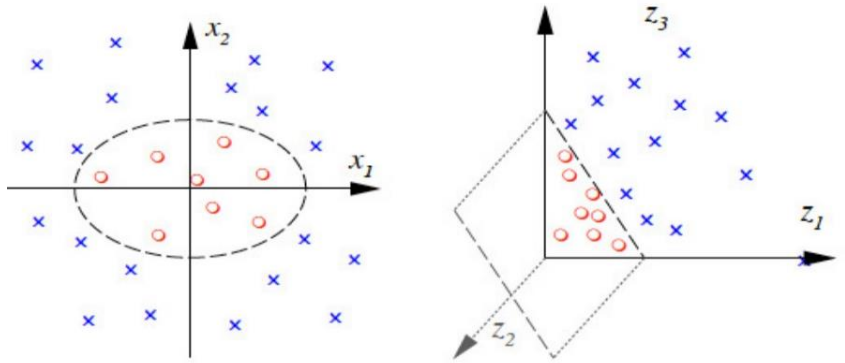
- Can try to learn the mapping from the data itself (e.g., using deep learning)
- Can use pre-defined "good" mappings (e.g., defined by kernel functions)

$$\phi : \mathcal{X} \to \mathcal{F}$$
$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad k(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$$

Thankfully, using kernels, you don't need to compute these mappings explicitly

The kernel will define an "implicit" feature mapping

$$\phi : \mathbb{R}^2 \to \mathbb{R}^3$$
$$(x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{(2)} x_1 x_2, x_2^2)$$



Important: The idea can be applied to any ML algo in which training and test stage only require computing pairwise similarities b/w inputs

- Kernel: A function $k(.,.)$ that gives dot product similarity b/w two inputs, say $\boldsymbol{x}_n$ and $\boldsymbol{x}_m$

$$k(\boldsymbol{x}_n, \boldsymbol{x}_m) = \phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{x}_m)$$

Important: As we will see, computing $k(.,.)$ does not require computing the mapping ϕ

- The kernel matrix $\boldsymbol{K}$ is of size $N \times N$ with each entry defined as

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j)$$

Kernels as (Implicit) Feature Maps

- Consider two inputs (in the same two-dim feature space): $\boldsymbol{x} = [x_1, x_2], \boldsymbol{z} = [z_1, z_2]$
- Suppose we have a function $k(.,.)$ which takes two inputs $\boldsymbol{x}$ and $\boldsymbol{z}$ and computes

$$k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$$

$$= (x_1 z_1 + x_2 z_2)^2$$

$$= x_1^2 z_1^2 + x_2^2 z_2^2 + 2 x_1 x_2 z_1 z_2$$

$$= \left(x_1^2, \sqrt{2} x_1 x_2, x_2^2\right)^\top \left(z_1^2, \sqrt{2} z_1 z_2, z_2^2\right)$$

$$= \phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$$

Didn't need to compute $\phi(x)$ explicitly. Just using the definition of the kernel $k(x, z) = (x^\top z)^2$ implicitly gave us this mapping for each input

Thus kernel function $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ implicitly defined a feature mapping ϕ such that for $\boldsymbol{x} = [x_1, x_2]$, $\phi(\boldsymbol{x}) = \left(x_1^2, \sqrt{2} x_1 x_2, x_2^2\right)$

Dot product similarity in the new feature space defined by the mapping ϕ

Remember that a kernel does two things: Maps the data implicitly into a new feature space (feature transformation) and computes pairwise similarity between any two inputs under the new feature representation

- Also didn't have to compute $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{z})$. Defn $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ gives that

Some Pre-defined Kernel Functions

Kernels can have a pre-defined form or can be learned from data (a bit advanced)

- Linear kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^\top \boldsymbol{z}$
- Quadratic Kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^2$ or $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x}^\top \boldsymbol{z})^2$
- Polynomial Kernel (of degree d): $k(\boldsymbol{x}, \boldsymbol{z}) = (\boldsymbol{x}^\top \boldsymbol{z})^d$ or $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x}^\top \boldsymbol{z})^d$
- Radial Basis Function (RBF) or "Gaussian" Kernel: $k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma \|\boldsymbol{x} - \boldsymbol{z}\|^2]$
  - Gaussian kernel gives a similarity score between 0 and 1
  - $\gamma > 0$ is a hyperparameter (called the kernel bandwidth parameter)
  - The RBF kernel corresponds to an infinite dim. feature space $\mathcal{F}$ (i.e., you can't actually write down or store the map $\boldsymbol{\phi}(\boldsymbol{x})$ explicitly – but we don't need to do that anyway)
  - Also called "stationary kernel": only depends on the distance between $\boldsymbol{x}$ and $\boldsymbol{z}$ (translating both by the same amount won't change the value of $k(\boldsymbol{x}, \boldsymbol{z})$)
- Kernel hyperparameters (e.g., d, γ) can be set via cross-validation

RBF Kernel = Infinite Dimensional Mapping

- We saw that the RBF/Gaussian kernel is defined as $k(\boldsymbol{x}, \boldsymbol{z}) = \exp[-\gamma \|\boldsymbol{x} - \boldsymbol{z}\|^2]$
- Using this kernel corresponds to mapping data to infinite dimensional space

$$k(x, z) = \exp[-(x - z)^2] \quad \text{(assuming } \gamma = 1 \text{ and } x \text{ and } z \text{ to be scalars)}$$

$$= \exp(-x^2)\exp(-z^2)\exp(2xz)$$

$$= \exp(-x^2)\exp(-z^2)\sum_{k=1}^{\infty}\frac{2^k x^k z^k}{k!}$$

$$= \phi(x)^\top \phi(z)$$

Thus an infinite-dim vector (ignoring the constants coming from the 2^k and $k!$ terms

- Here $\phi(\boldsymbol{x}) = [\exp(-x^2)x^1, \exp(-x^2)x^2, \exp(-x^2)x^3, \ldots, \exp(-x^2)x^\infty]$
- But again, note that we never need to compute $\boldsymbol{\phi(x)}$ to compute $k(x, z)$
  - $k(x, z)$ is easily computable from its definition itself ($\exp[-(x - z)^2]$ in this case)

Using Kernels

- Kernels can turn many linear models into nonlinear models
- $k(\boldsymbol{x}, \boldsymbol{z})$ represents a dot product in some high-dim feature space $\mathcal{F}$
- Important: Any ML model/algo in which, during training and test, inputs only appear as dot product can be "kernelized"
- Just replace each term of the form $\boldsymbol{x}_i^\top \boldsymbol{x}_j$ by $\boldsymbol{\phi}(\boldsymbol{x}_i)^\top \boldsymbol{\phi}(\boldsymbol{x}_j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = K_{ij}$
- Most ML models/algos can be easily kernelized, e.g.,
  - Distance based methods, Perceptron, SVM, linear regression, etc.
  - Many of the unsupervised learning algorithms too can be kernelized (e.g., K-means clustering, Principal Component Analysis, etc.)
  - Let's look at two examples: Kernelized SVM and Kernelized Ridge Regression

Nonlinear SVM using Kernels

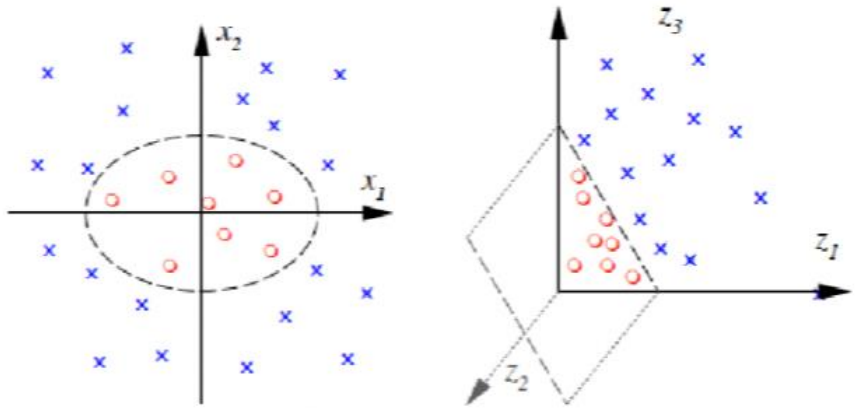
Kernelized SVM Training

- The soft-margin linear SVM objective (with no bias term)

$$\underset{\mathbf{0}\leq\boldsymbol{\alpha}\leq\boldsymbol{C}}{\operatorname{argmax}} \quad \boldsymbol{\alpha}^{\mathsf{T}}\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^{\mathsf{T}}\mathbf{G}\boldsymbol{\alpha}$$

$G_{ij} = y_i y_j \, \boldsymbol{x}_i^{\mathsf{T}} \boldsymbol{x}_j$

- To kernelize, we can simply replace $G_{ij} = y_i y_j \, \boldsymbol{x}_i^{\mathsf{T}} \boldsymbol{x}_j$ by $y_i y_j K_{ij}$
  - .. where $K_{ij} = k(x_i, x_j) = \phi(\boldsymbol{x}_i)^{\mathsf{T}} \phi(\boldsymbol{x}_j)$ for a suitable kernel function k
- The problem can now be solved just like the linear SVM case
- The new SVM learns a linear separator in kernel-induced feature space $\mathcal{F}$
  - This corresponds to a non-linear separator in the original feature space $\mathcal{X}$



Nonlinear Ridge Regression using Kernels

Kernelized Ridge Regression

- Recall the ridge regression problem: $\boldsymbol{w} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$
- The solution to this problem was

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)(\sum_{n=1}^{N} y_n \boldsymbol{x}_n) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- Can use matrix inversion lemma $(\mathbf{F}\mathbf{H}^{-1}\mathbf{G} - \mathbf{E})^{-1}\mathbf{F}\mathbf{H}^{-1} = \mathbf{E}^{-1}\mathbf{F}(\mathbf{G}\mathbf{E}^{-1}\mathbf{F} - \mathbf{H})^{-1}$
- Using the lemma, can rewrite $\boldsymbol{w}$ as

$$\boldsymbol{w} = \mathbf{X}^\top (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = \mathbf{X}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n \boldsymbol{x}_n \quad \text{where} \quad \boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y} = (\mathbf{K} + \lambda \mathbf{I}_N)^{-1} \boldsymbol{y}$$

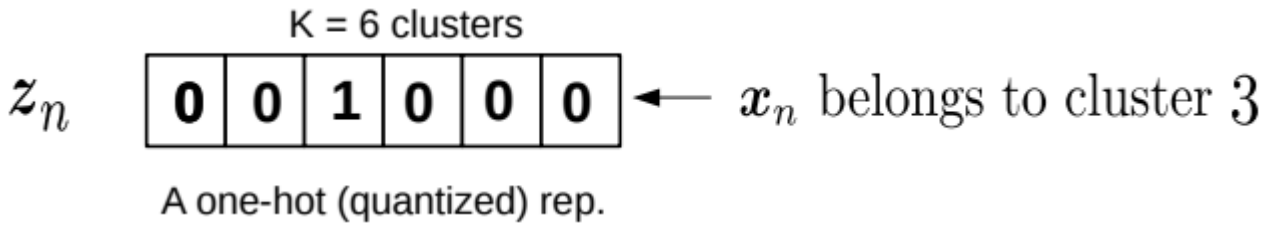
- Kernelized weight vector will be $\boldsymbol{w} = \sum_{n=1}^{N} \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)$
- Prediction for a test input $\boldsymbol{x}_*$ will be $\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_*) = \sum_{n=1}^{N} \alpha_n \boldsymbol{\phi}(\boldsymbol{x}_n)^\top \boldsymbol{\phi}(\boldsymbol{x}_*) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x}_*)$

Unsupervised Learning: Data Clustering via K-means, Dimensionality Reduction: PCA

Unsupervised Learning

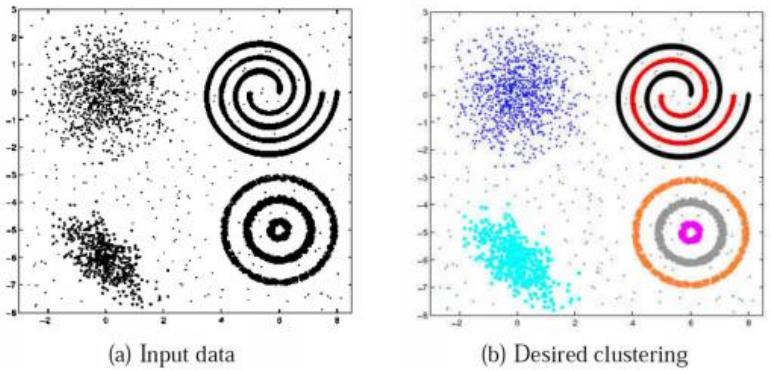
- It's about learning interesting/useful structures in the data (unsupervisedly!)
- There is no supervision (no labels/responses), only inputs $\boldsymbol{x_1, x_2, \ldots, x_N}$
- Some examples of unsupervised learning
  - Clustering: Grouping similar inputs together (and dissimilar ones far apart)
  - Dimensionality Reduction: Reducing the data dimensionality
  - Estimating the probability density of data (which distribution "generated" the data)
- Most unsup. learning algos can also be seen as learning a new representation of data
  - For example, hard clustering can be used to get a one-hot representation

In contrast, there is also soft/probabilistic clustering in which $\boldsymbol{z_n}$ will be be probability vector that sums to)



Clustering

- Given: N unlabeled examples $\boldsymbol{x_1, x_2, \ldots, x_N}$; desired no. of partitions K
- Goal: Group the examples into K "homogeneous" partitions



(a) Input data (b) Desired clustering

Picture courtesy: "Data Clustering: 50 Years Beyond K-Means", A.K. Jain (2008)

In some cases, we may not know the right number of clusters in the data and may want to learn that (technique exists for doing this but beyond the scope)

- Loosely speaking, it is classification without ground truth labels
- A good clustering is one that achieves
  - High within-cluster similarity
  - Low inter-cluster similarity

Similarity can be Subjective

- Clustering only looks at similarities b/w inputs, since no labels are given
- Without labels, similarity can be hard to define

- Thus using the right distance/similarity is very important in clustering
- In some sense, related to asking: "Clustering based on what" ?

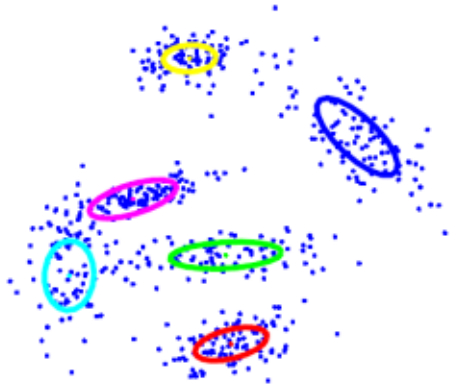
Clustering: Some Examples

- Document/Image/Webpage Clustering
- Image Segmentation (clustering pixels)

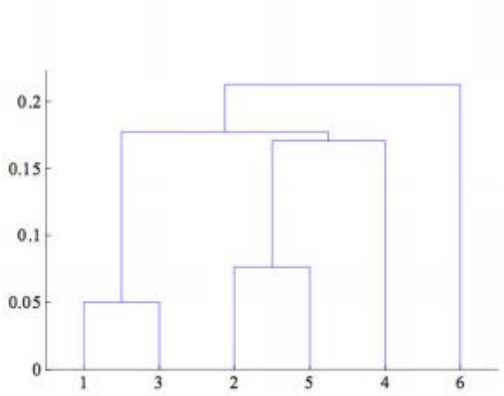
- Clustering web-search results
- Clustering (people) nodes in (social) networks/graphs
- .. and many more..

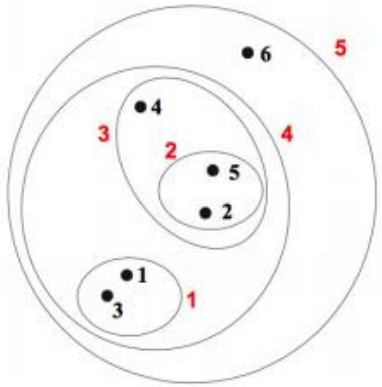
Types of Clustering

- Flat or Partitional clustering
  - Partitions are independent of each other



- Hierarchical clustering
  - Partitions can be visualized using a tree structure (Application : Social Network Data)





Flat Clustering: *K*-means algorithm (Lloyd, 1957)

- **Input:** N unlabeled examples $\boldsymbol{x_1, x_2, \ldots, x_N}$; $\boldsymbol{x_n} \in \mathbb{R}^D$; desired no. of partitions K
- **Desired Output:** Cluster assignments of these N examples and K cluster means
- **Initialize:** The K cluster means denoted by $\boldsymbol{\mu_1, \mu_2, \ldots, \mu_K}$; with each $\boldsymbol{\mu_k} \in \mathbb{R}^D$
  - Usually initialized randomly, but good initialization is crucial; many smarter initialization heuristics exist (e.g., *K*-means++, Arthur & Vassilvitskii, 2007)
- **Iterate:**
  - (Re)-Assign each input $\boldsymbol{x_n}$ to its closest cluster center (based on the smallest Eucl. distance)

$\mathcal{C}_k$: Set of examples assigned to cluster k with center $\boldsymbol{\mu_k}$

$$\mathcal{C}_k = \{n : \quad k = \arg\min_k ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2\}$$

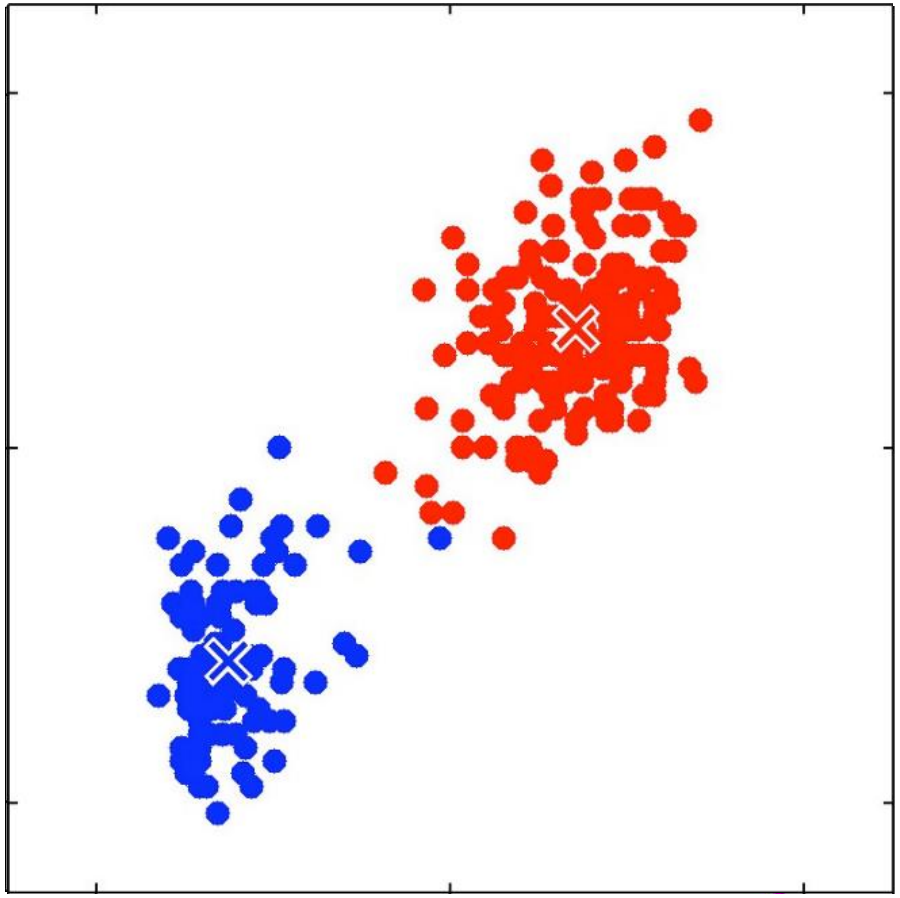
  - Update the cluster means

$$\mu_k = \text{mean}(\mathcal{C}_k) = \frac{1}{|\mathcal{C}_k|} \sum_{n \in \mathcal{C}_k} \boldsymbol{x}_n$$

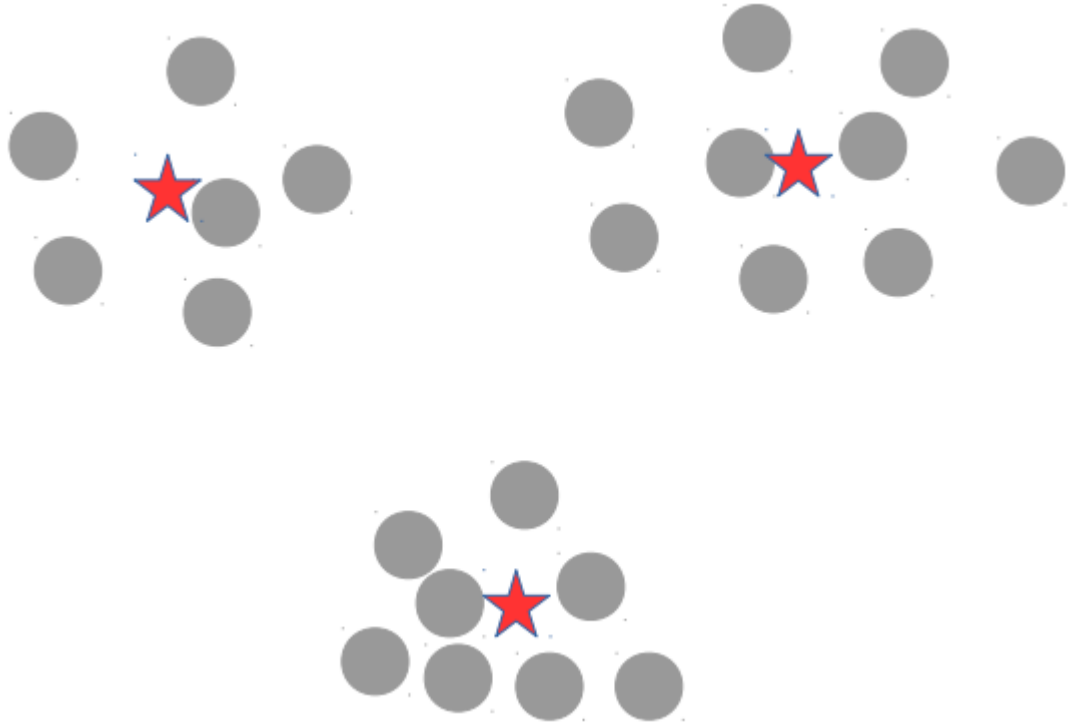
  - Repeat until not converged

Some ways to declare convergence if between two successive iterations:
- Cluster means don't change
- Cluster assignments don't change
- Clustering "loss" doesn't change by much

K-means: An Illustration



K-means = LwP with Unlabeled Data



- Guess the means
- Predict the labels (cluster ids)
- Recompute the means using these predicted labels
- Repeat until not converged

What Loss Function is *K*-means Optimizing?

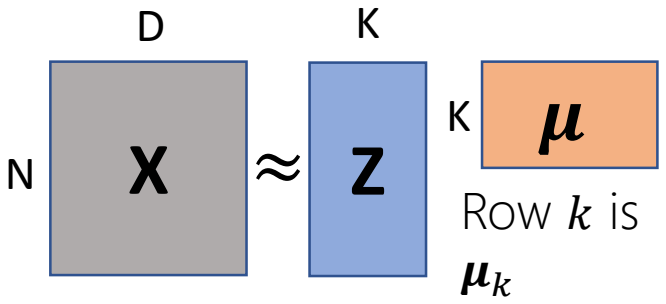
$\boldsymbol{z}_n = [z_{n1}, z_{n2}, \ldots, z_{nK}]$ denotes a length K one-hot encoding of $\boldsymbol{x}_n$

- Let $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K$ be the K cluster centroids/means
- Let $z_{nk} \in \{0, 1\}$ be s.t. $z_{nk} = 1$ if $\boldsymbol{x}_n$ belongs to cluster k, and 0 otherwise
- Define the distortion or "loss" for the cluster assignment of $\boldsymbol{x}_n$

$$\ell(\boldsymbol{\mu}, \boldsymbol{x}_n, \boldsymbol{z}_n) = \sum_{k=1}^{K} z_{nk} ||\boldsymbol{x}_n - \boldsymbol{\mu}_k||^2$$

- Total distortion over all points defines the K-means "loss function"

$$L(\boldsymbol{\mu}, \mathbf{X}, \mathbf{Z}) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} ||\boldsymbol{x}_n - \mu_k||^2 = \underbrace{||\mathbf{X} - \mathbf{Z}\boldsymbol{\mu}||_F^2}_{\text{matrix factorization view}}$$

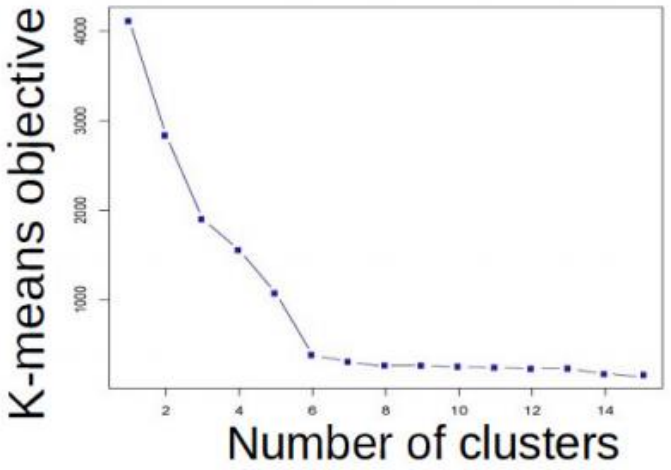


- The K-means problem is to minimize this objective w.r.t. $\boldsymbol{\mu}$ and $\boldsymbol{Z}$

Row n is $\boldsymbol{z}_n$ (one-hot vector)

K-means: Choosing *K*

- One way to select K for the K-means algorithm is to try different values of K, plot the K-means objective versus K, and look at the "elbow-point"



K=6 is the elbow point

- Can also information criterion such as AIC (Akaike Information Criterion)

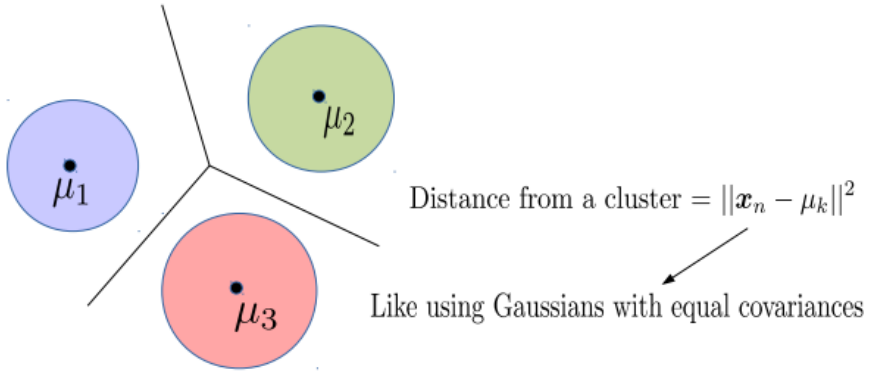
$$AIC = 2\mathcal{L}(\hat{\boldsymbol{\mu}}, \mathbf{X}, \hat{\mathbf{Z}}) + KD$$

and choose K which gives the smallest AIC

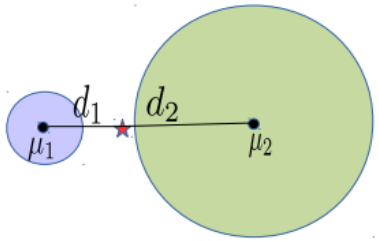
K-means: Decision Boundaries and Cluster Sizes/Shapes

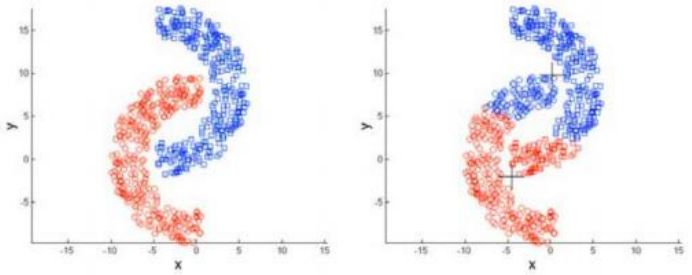
- K-mean assumes that the decision boundary between any two clusters is linear
- Reason: The K-means loss function implies assumes equal-sized, spherical clusters

Reason: Use of Euclidean distances



- May do badly if clusters are not roughly equi-sized and convex-shaped





Kernel *K*-means

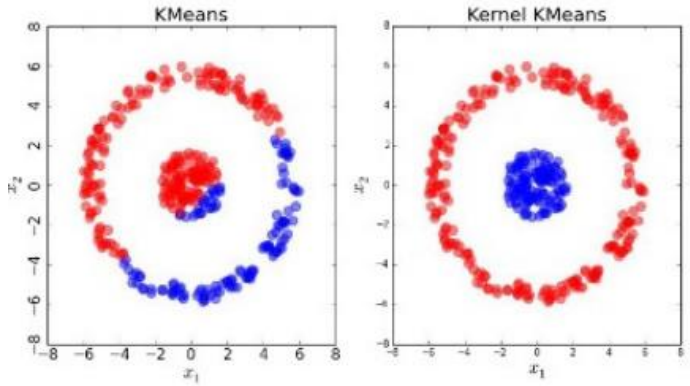
- Basic idea: Replace the Eucl. distances in K-means by the kernelized versions

$$||\phi(\boldsymbol{x}_n) - \phi(\boldsymbol{\mu}_k)||^2 = ||\phi(\boldsymbol{x}_n)||^2 + ||\phi(\boldsymbol{\mu}_k)||^2 - 2\phi(\boldsymbol{x}_n)^\top \phi(\boldsymbol{\mu}_k)$$
$$= k(\boldsymbol{x}_n, \boldsymbol{x}_n) + k(\boldsymbol{\mu}_k, \boldsymbol{\mu}_k) - 2k(\boldsymbol{x}_n, \boldsymbol{\mu}_k)$$

Kernelized distance between input $\boldsymbol{x}_n$ and mean of cluster k

- Here $k(.,.)$ denotes the kernel function and ϕ is its (implicit) feature map
- Note: $\phi(\mu_k)$ is the mean of ϕ mappings of the data points assigned to cluster k

$$\phi(\mu_k) = \frac{1}{|\mathcal{C}_k|} \sum_{n:z_n=k} \phi(\boldsymbol{x}_n)$$



Not the same as the ϕ mapping of the mean of the data points assigned to cluster k

Dimensionality Reduction: PCA

Dimensionality Reduction

- A broad class of techniques
- Goal is to compress the original representation of the inputs
- Example: Approximate each input $\boldsymbol{x}_n \in \mathbb{R}^D$, $n = 1,2,\dots,N$ as a linear combination of $K < \min\{D, N\}$ "basis" vectors $\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K$, each also $\in \mathbb{R}^D$

Note: These "basis" vectors need not necessarily be linearly independent but classic principal component analysis (PCA), they are

$$\boldsymbol{x}_n \approx \sum_{k=1}^{K} z_{nk}\boldsymbol{w}_k = \mathbf{W}\boldsymbol{z}_n$$

$\mathbf{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \dots, \boldsymbol{w}_K]$ is $D \times K$

$\boldsymbol{z}_n = [z_{n1}, z_{n2}, \dots, z_{nK}]$ is $K \times 1$

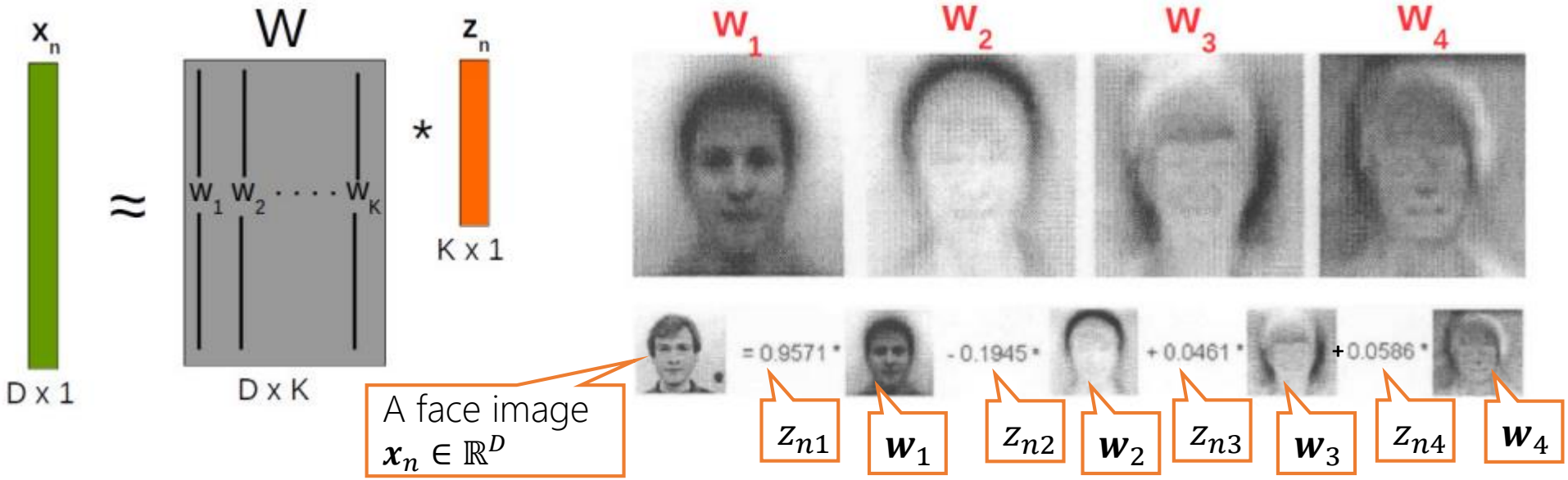
- We have represented each $\boldsymbol{x}_n \in \mathbb{R}^D$ by a K-dim vector $\boldsymbol{z}_n$ (a new feat. rep)
- Can think of $\mathbf{W}$ as a linear mapping that transforms low-dim $\boldsymbol{z}_n$ to high-dim $\boldsymbol{x}_n$
- To store N such inputs $\{\boldsymbol{x}_n\}_{n=1}^N$, we need to keep $\mathbf{W}$ and $\{\boldsymbol{z}_n\}_{n=1}^N$
  - Originally we required $N \times D$ storage, now $N \times K + D \times K = (N + D) \times K$ storage
  - If $K \ll \min\{D, N\}$, this yields substantial storage saving, hence good compression

Dimensionality Reduction

- Dim-red for face images

Each "basis" image is like a "template" that captures the common properties of face images in the dataset

K=4 "basis" face images



- In this example, $z_n \in \mathbb{R}^K$ $(K = 4)$ is a low-dim feature rep. for $x_n \in \mathbb{R}^D$
- Essentially, each face image in the dataset now represented by just 4 real numbers
- Different dim-red algos differ in terms of how the basis vectors are defined/learned
  - .. And in general, how the function f in the mapping $x_n = f(z_n)$ is defined

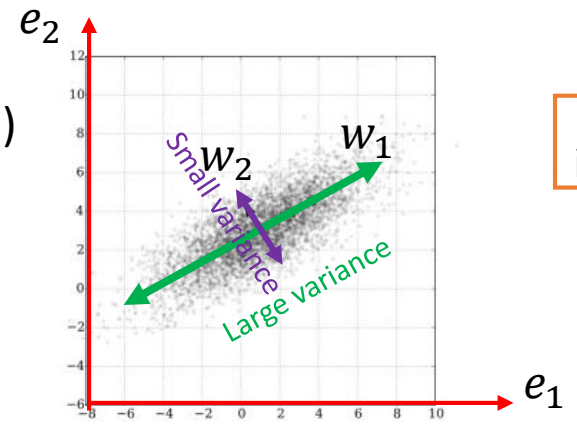
Like 4 new features

Principal Component Analysis (PCA)

- A classic linear dim. reduction method (Pearson, 1901; Hotelling, 1930)
- Can be seen as
  - Learning directions (co-ordinate axes) that capture maximum variance in data

e_1, e_2: Standard co-ordinate axis ($\boldsymbol{x} = [x_1, x_2]$)

w_1, w_2: New co-ordinate axis ($\boldsymbol{z} = [z_1, z_2]$)



PCA is essentially doing a change of axes in which we are representing the data

  - Learning projection directions that result in **smallest reconstruction error**

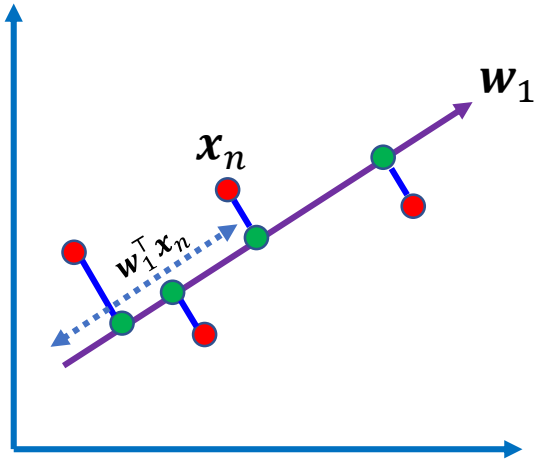
$$\underset{\boldsymbol{W},\boldsymbol{Z}}{\operatorname{argmin}} \sum_{n=1}^{N} \|\boldsymbol{x}_n - \boldsymbol{W}\boldsymbol{z}_n\|^2 = \underset{\boldsymbol{W},\boldsymbol{Z}}{\operatorname{argmin}} \|\boldsymbol{X} - \boldsymbol{Z}\boldsymbol{W}\|^2$$

Subject to orthonormality constraints: $\boldsymbol{w}_i^\top \boldsymbol{w}_j = 0$ for $i \neq j$ and $\|\boldsymbol{w}_i\|^2 = 1$

- PCA also assumes that the projection directions are orthonormal

Solving PCA by Finding Max. Variance Directions

- Consider projecting an input $\boldsymbol{x}_n \in \mathbb{R}^D$ along a direction $\boldsymbol{w}_1 \in \mathbb{R}^D$
- Projection of $\boldsymbol{x}_n$ (red points below) will be $\boldsymbol{w}_1^\top \boldsymbol{x}_n$ (green pts below)



Mean of projections of all inputs:

$$\frac{1}{N}\sum_{n=1}^{N} \boldsymbol{w}_1^\top \boldsymbol{x}_n = \boldsymbol{w}_1^\top \left(\frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n\right) = \boldsymbol{w}_1^\top \boldsymbol{\mu}$$

$\boldsymbol{S}$ is the $D \times D$ cov matrix of the data:

$$\boldsymbol{S} = \frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{x}_n - \boldsymbol{\mu})(\boldsymbol{x}_n - \boldsymbol{\mu})^\top$$

Variance of the projections:

$$\frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{w}_1^\top \boldsymbol{x}_n - \boldsymbol{w}_1^\top \boldsymbol{\mu})^2 = \frac{1}{N}\sum_{n=1}^{N} \{\boldsymbol{w}_1^\top (\boldsymbol{x}_n - \boldsymbol{\mu})\}^2 = \boldsymbol{w}_1^\top \boldsymbol{S} \boldsymbol{w}_1$$

- Want $\boldsymbol{w}_1$ such that variance $\boldsymbol{w}_1^\top \boldsymbol{S} \boldsymbol{w}_1$ is maximized

Need this constraint otherwise the objective's max will be infinity

$$\underset{\boldsymbol{w}_1}{\text{argmax}}\ \boldsymbol{w}_1^\top \boldsymbol{S} \boldsymbol{w}_1 \quad \text{s.t.} \quad \boldsymbol{w}_1^\top \boldsymbol{w}_1 = 1$$

For already centered data, $\boldsymbol{\mu} = \boldsymbol{0}$ and $\boldsymbol{S} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top = \frac{1}{N}\boldsymbol{X}\boldsymbol{X}^\top$

Max. Variance Direction

- Our objective function was $\underset{\boldsymbol{w}_1}{\text{argmax}}\, \boldsymbol{w}_1^\top \boldsymbol{S}\boldsymbol{w}_1$ s.t. $\boldsymbol{w}_1^\top \boldsymbol{w}_1 = 1$
- Can construct a Lagrangian for this problem

$$\underset{\boldsymbol{w}_1}{\text{argmax}}\, \boldsymbol{w}_1^\top \boldsymbol{S}\boldsymbol{w}_1 + \lambda_1(1\text{-}\boldsymbol{w}_1^\top \boldsymbol{w}_1)$$

- Taking derivative w.r.t. $\boldsymbol{w}_1$ and setting to zero gives $\boldsymbol{S}\boldsymbol{w}_1 = \lambda_1 \boldsymbol{w}_1$
- Therefore $\boldsymbol{w}_1$ is an eigenvector of the cov matrix $\boldsymbol{S}$ with eigenvalue λ_1
- Claim: $\boldsymbol{w}_1$ is the eigenvector of $\boldsymbol{S}$ with largest eigenvalue λ_1. Note that

$$\boldsymbol{w}_1^\top \boldsymbol{S}\boldsymbol{w}_1 = \lambda_1 \boldsymbol{w}_1^\top \boldsymbol{w}_1 = \lambda_1$$

- Thus variance $\boldsymbol{w}_1^\top \boldsymbol{S}\boldsymbol{w}_1$ will be max. if λ_1 is the largest eigenvalue (and $\boldsymbol{w}_1$ is the corresponding top eigenvector; also known as the first Principal Component)
- Other large variance directions can also be found likewise (with each being orthogonal to all others) using the eigendecomposition of cov matrix $\boldsymbol{S}$ (this is PCA)

Note: Total variance of the data is equal to the sum of eigenvalues of $\boldsymbol{S}$, i.e., $\sum_{d=1}^{D} \lambda_d$

PCA would keep the top $K < D$ such directions of largest variances

Note: In general, $\boldsymbol{S}$ will have D eigvecs

Principal Component Analysis

- Center the data (subtract the mean $\boldsymbol{\mu} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n$ from each data point)
- Compute the $D \times D$ covariance matrix $\mathbf{S}$ using the centered data matrix $\mathbf{X}$ as

$$\mathbf{S} = \frac{1}{N}\mathbf{X}^\top\mathbf{X}$$ (Assuming $\mathbf{X}$ is arranged as $N \times D$)

- Do an eigendecomposition of the covariance matrix $\mathbf{S}$ (many methods exist)
- Take top $K < D$ leading eigvectors $\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K\}$ with eigvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_K\}$
- The K-dimensional projection/embedding of each input is

$$\boldsymbol{z}_n \approx \mathbf{W}_K^\top \boldsymbol{x}_n$$

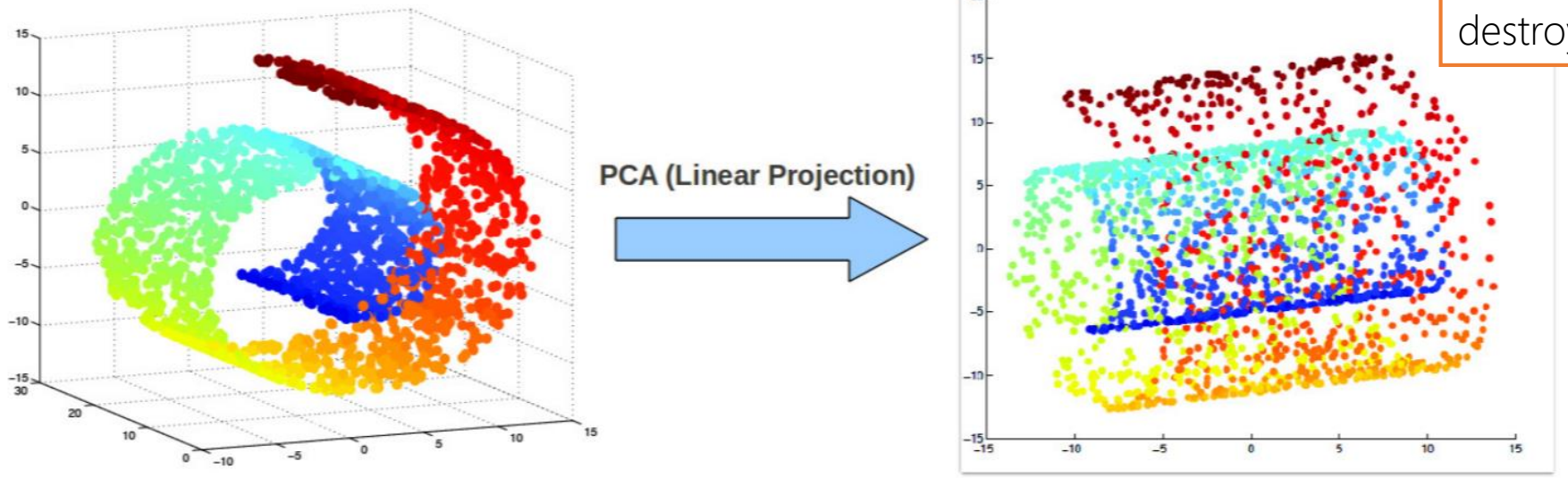
$\mathbf{W}_K = [\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K]$ is the "projection matrix" of size $D \times K$

Can decide how many eigvecs to use based on how much variance we want to campure)

Nonlinear Dimensionality Reduction

Beyond Linear Projections

- Consider the swiss-roll dataset (points lying close to a manifold)

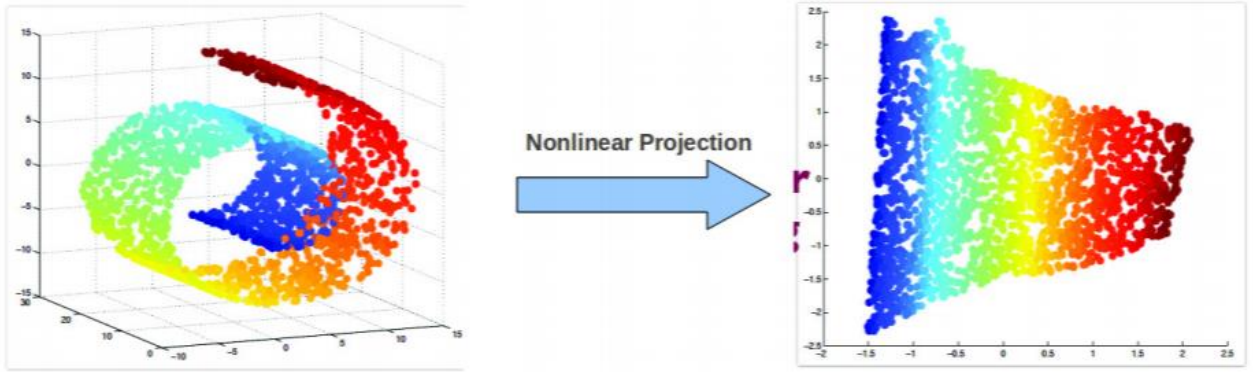


Relative positions of points destroyed after the projection

- Linear projection methods (e.g., PCA) can’ t capture intrinsic nonlinearities
  - Maximum variance directions may not be the most interesting ones

Nonlinear Dimensionality Reduction

- We want to a learn nonlinear low-dim projection



Relative positions of points preserved after the projection

- Some ways of doing this
  - Nonlinearize a linear dimensionality reduction method. E.g.:
    - Cluster data and apply linear PCA within each cluster (mixture of PCA)
    - Kernel PCA (nonlinear PCA)
  - Using manifold based methods that intrinsically preserve nonlinear geometry,
    - See Literatures for more
- .. or use unsupervised deep learning techniques

Kernel PCA

- Recall PCA: Given N observations $\boldsymbol{x}_n \in \mathbb{R}^D$, $n = 1,2,\ldots,N$,

$D \times D$ cov matrix assuming centered data

$$\mathbf{S} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top$$

D eigenvectors of $\mathbf{S}$

$$\mathbf{S}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i \;\; \forall i = 1,\ldots,D$$

- Assume a kernel k with associated M dimensional nonlinear map $\boldsymbol{\phi}$

$M \times M$ cov matrix assuming centered data in the kernel-induced feature space

$$\mathbf{C} = \frac{1}{N}\sum_{n=1}^{N} \phi(\boldsymbol{x}_n)\phi(\boldsymbol{x}_n)^\top$$

M eigenvectors of $\mathbf{C}$

$$\mathbf{C}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i \;\; \forall i = 1,\ldots,M$$

- Would like to do it without computing $\mathbf{C}$ and the mappings $\boldsymbol{\phi}(\boldsymbol{x}_n)'s$ since M can be very large (even infinite, e.g., when using an RBF kernel)

- Can be reduced to eigendecomposition of the $N \times N$ kernel matrix $\mathbf{K}$
  - Can verify that each $\boldsymbol{v}_i$ above can be written as a lin-comb of the inputs: $\boldsymbol{v}_i = \sum_{n=1}^{N} a_{in}\boldsymbol{\phi}(\boldsymbol{x}_n)$
  - Can show that finding $\boldsymbol{a}_i = [a_{i1}, a_{i2}, \ldots, a_{iN}]$ reduces to solving an eigendecomposition of $\mathbf{K}$
  - Note: Due to req. of centering, we work with a centered kernel matrix $\widetilde{\mathbf{K}} = \mathbf{K} - 1_N\mathbf{K} - \mathbf{K}1_N + 1_N\mathbf{K}1_N$

$N \times N$ matrix of all 1s

References

- NPTEL ML lecture notes
- YouTube Video Lectures
- **Book by Bishop**
- **Book by Duda**
- **Book by Tom Mitchell**
- **Hands on ML book by A. Geron**
- Stackexchange.com (datascience)
- Towards Data Science Website
- Datacamp.com
- Medium.com
- Different source of ML course web pages
- Google Search
- Scikit-learn.org

Thank You