microsoft · principles of machine learning

september · november 2016 ¨ **sharepoint interface**

MODULE1 · CLASSIFICATION

- INTRODUCTION TO CLASSIFICATION
- LOSS FUNCTIONS FOR CLASSIFICATION
- STATISTICAL LEARNING THEORY FOR SUPERVISED LEARNING
- LOGISTIC REGRESSION
- MAXIMUM LIKELIHOOD PERSPECTIVE
- EVALUATION METHODS FOR CLASSIFIERS
- ROC CURVE ALGORITHM

BUILDING CLASSIFICATION MODELS

- IMBALANCED DATA

CLASSIFICATION IN AZURE ML AND R

MODULE2 · REGRESSION

- INTRODUCTION TO LINEAR REGRESSION
- MULTIPLE LINEAR REGRESSION
- EVALUATING REGRESSION MODELS

CREATING REGRESSION MODELS

- INFLUENTIAL POINTS
- OUTLIERS

REGRESSION IN AZURE ML AND R

MODULE3 · IMPROVING MACHINE LEARNING MODELS

- FEATURE SELECTION
- REGULARIZATION
- INTERPRETING FEATURES
- FEATURE SCALING

TECHNIQUES FOR IMPROVING MODELS

- SWEEPING PARAMETERS
- CROSS VALIDATION
- NESTED CROSS VALIDATION

IMPROVING MACHINE LEARNING MODELS IN AZURE ML AND R

MODULE4 · TREE AND ENSEMBLE METHODS

- DECISION TREES
- CONSTRUCTION DECISION TREES
- WHAT IS INFORMATION?
- ENTROPY
- SPLITTING CRITERIA FOR DECISION TREES: INFORMATION GAIN

ENSEMBLE METHODS

- BOOSTING
- ADABOOST
- COORDINATE DESCENT
- DECISION FORESTS

DECISION TREES IN AZURE ML AND R

MODULE5 · OPTIMIZATION-BASED METHODS

NEURAL NETWORKS

- INTRODUCTION TO NEURAL NETWORKS
- BACKPROPAGATION
- BACKPROPAGATION THOUGH A HIDDEN NEURAL NETWORK LAYER

SUPPORT VECTOR MACHINES (SVM)

- INTRODUCTION TO SVMS
- KERNELS FOR SVMS

SUPPORT VECTOR MACHINES IN AZURE ML AND R

MODULE6 · CLUSTERING AND RECOMMENDERS

CLUSTERING

- INTRODUCTION TO CLUSTERING
- K-MEANS CLUSTERING
- CHOOSING K FOR K-MEANS CLUSTERING
- HIERARCHICAL AGGLOMERATIVE CLUSTERING

RECOMMENDERS

- RECOMMENDER SYSTEMS
- MATRIX FACTORIZATION

DECISION TREES IN AZURE ML AND R

module1 · classification

introduction to classification

Machine Learning

Machine Learning is a field that grew out of artificial intelligence within computer science. The practice teaches a computer through examples.

Machine Learning is most closely related to statistics.

Classification

Classification is the process of labeling examples of a dataset into defined classes or categories.

A **Training Set** of data exists to learn what the model is trying to predict.

A **Test Set** of data is assigned as images not in the training set purposed to evaluate performance.

In the case of images, each observation is represented by a set of numbers (features) as a vector.

The classification assigned to an observation is referred to as the **label** (often '1' or '0').

Machine Learning requires that the data is initially represented in the correct manner.

Defining **Classification** formally:

A given training set (x_i, y_i) for $1 = n$, a classification model f is created to predict label y for a new x.

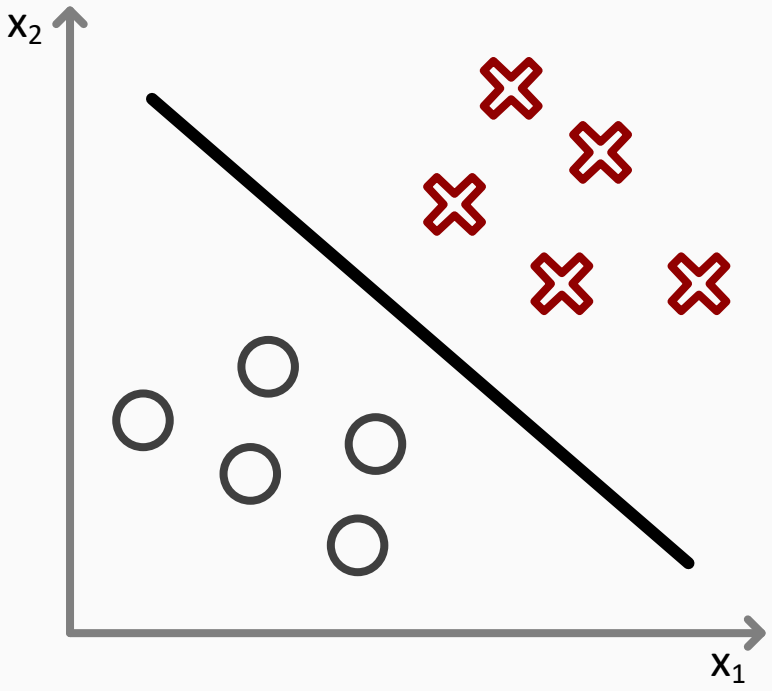
The machine learning algorithm will create the function f.

The predicted y for a new x is simple the sign of $f(x)$.

Classification is designed for Yes/No question → **Binary Classification**

Common Classification Algorithms:

- Logistic Regression (with L1 or L2 regularization)
- Decision Trees/Classification Trees/CART/C4.5/C5.0
- AdaBoost (Boosted Decision Trees)
- Support Vector Machines
- Random Forests
- Neural Networks



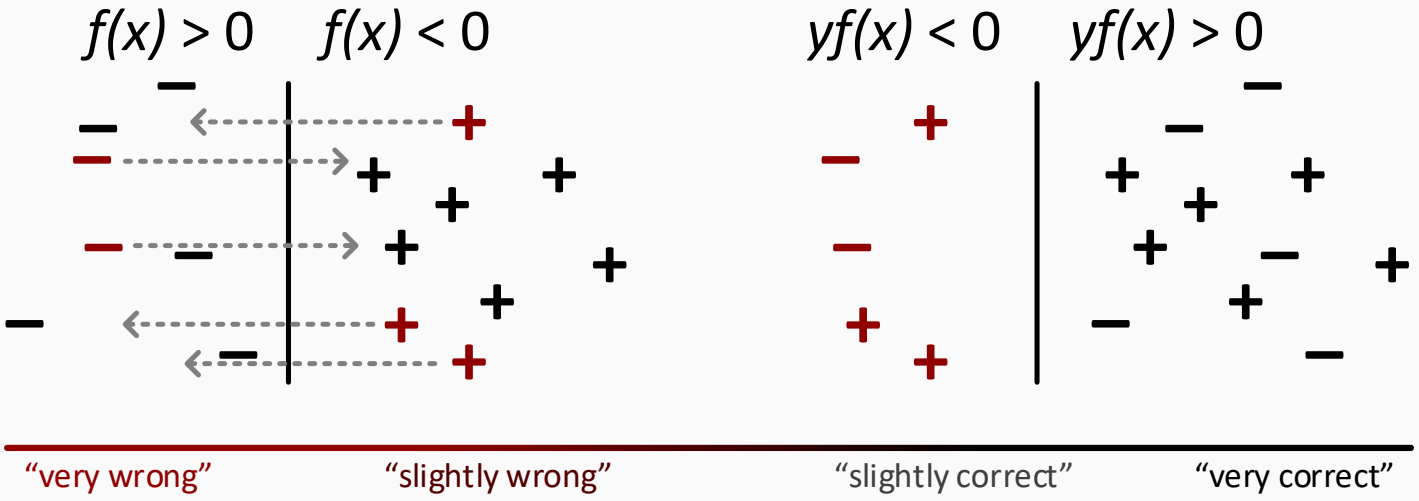
loss functions ⤢ for classification

Classification error is measured as:

Fraction of times $sign(f(x_i))$ is not y_i:

$$y_i = \frac{1}{n}\sum_{i=1}^{n}[y_i \neq sign(f(x_i))]$$

The expression is geometrically illustrated as follows:



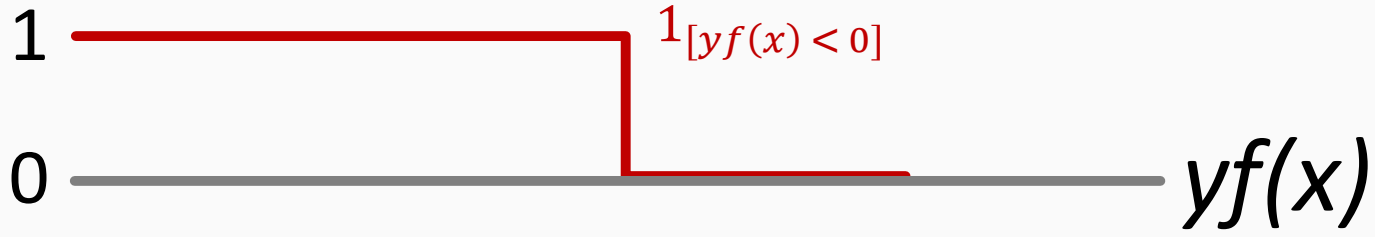
The **Decision Boundary** is the center line dividing examples in each illustration above. In the **left** image, the left cluster represents negative **(-)** function values and the right represents positive function values **(+)**. The examples in **red** represent **misclassified examples**. The **right** image is the effect once all examples are shifted across the decision boundary to cluster them based on correct/incorrect classification predictions opposed to the original values. The new representation in the **right** illustration changes the function from **(+/-)** predictions on each side of the decision boundary to functions where:

- Both $\boldsymbol{y(x)}$ **and** $\boldsymbol{f(x)}$ are positive **(+)** on the **left** side of the decision boundary;
- Both $\boldsymbol{y(x)}$ **and** $\boldsymbol{f(x)}$ are negative **(-)** on the **right** side of the decision boundary

Therefore, the right side of the decision boundary represents cases where the sign of $\boldsymbol{f \neq y}$ and will thus be penalized heavily for their values.

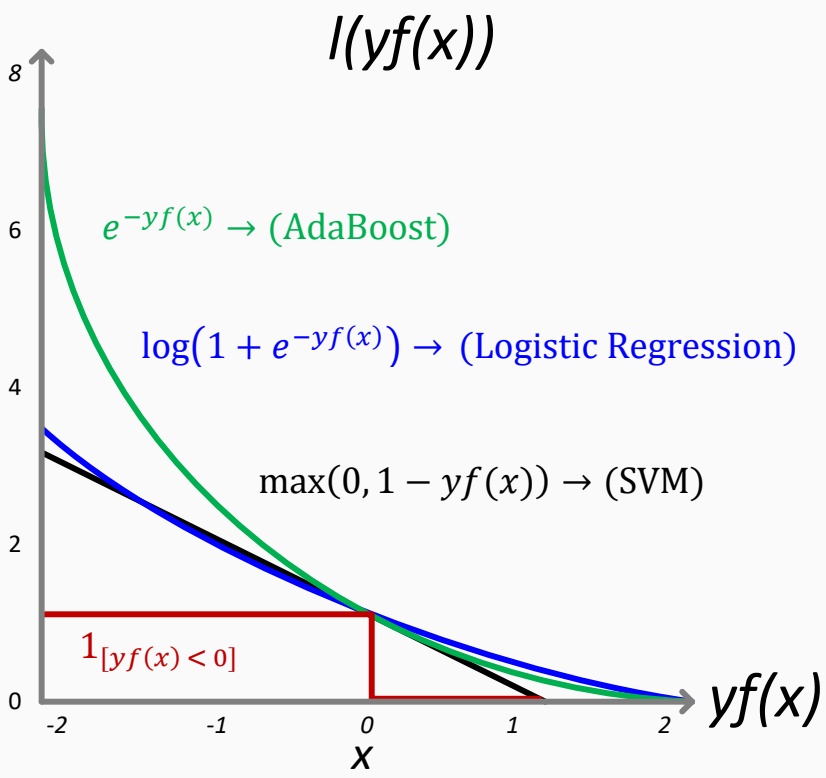
Logical examples of the $yf(x) < 0, yf(x) > 0$:

- If $y > 0$ and $f > 0$, the classification is **correct** and the loss function does **not** penalize.
- If $y < 0$ and $f < 0$, the classification is **correct**, and the loss function does **not** penalize.
- If $y > 0$ and $f < 0$, the classification is **incorrect**, and the loss function **does** penalize.
- If $y < 0$ and $f > 0$, the classification is **incorrect**, and the loss function **does** penalize



Loss Function Intuition:

Fraction of times $sign(f(x_i))$ is not y_i:



$$y_i = \frac{1}{n}\sum_{i=1}^{n}[y_i \neq sign(f(x_i))]$$

$$y_i = \frac{1}{n}\sum_{i=1}^{n}[y_i f(x_i) < 0]$$

$$y_i \leq \frac{1}{n}\sum_{i=1}^{n}\ell(y_i f(x_i))$$

Applying an algorithm attempts to minimize the Loss Function:

$$\min_{\text{models } f} \frac{1}{n}\sum_{i=1}^{n}\ell(y_i f(x_i))$$

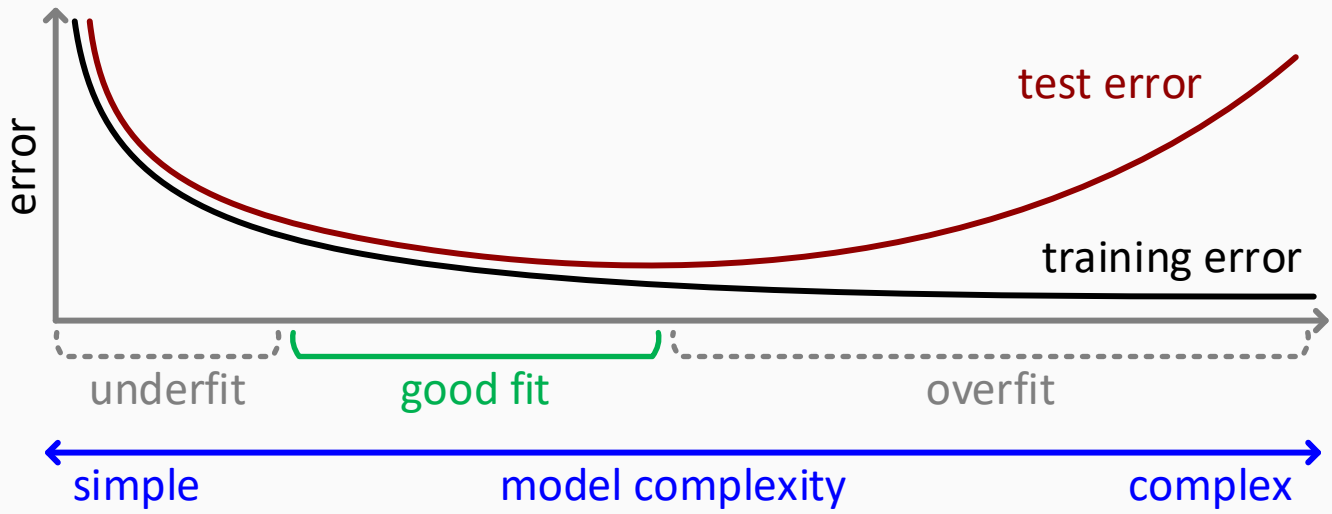
However, the above model fails to **generalize** new examples which key in machine learning.

statistical learning theory for supervised learning

Ockham's Razor states the best models are simple models that fit the data well.

William of Ockham, English fryer and philosopher (1287-1347) said that among hypotheses that predict equally well, choose the one with the fewest assumptions.

The key to understanding Statistical Learning Theory:



Therefore, the goal is obtain a **balance of accuracy and simplicity**.

Most common machine learning methods choose f to **minimize training error and complexity**.

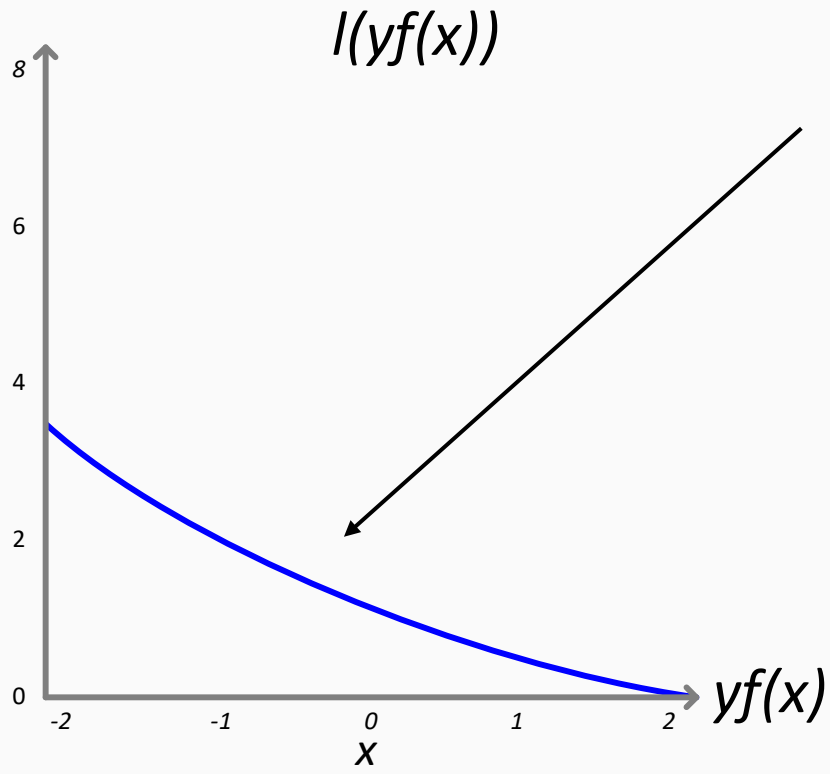
The ultimate goal is to mitigate what is known as the **"curse of dimensionality"**

> The curse of dimensionality is the tendency to overfit a dataset when many features are available, but not enough data to compensate for predictive purpose. Therefore, the amount of data available would need to increase exponentially to prevent models overfitting.

Simplicity is measured in multiple ways and referred often to as **regularization**:

$$\frac{1}{n}\sum_{i=1}^{n}\ell(y_i f(x_i)) + \text{Regulatization}(f)$$

logistic regression ⬍ for machine learning



$$\min_{\text{models } f} \log\left(1 + e^{-yf(x)}\right)$$

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} = \sum_{j=1}^{p} \beta_j x_{ij}$$

$$\min_{\beta_1,\beta_2,\ldots,\beta_p} \frac{1}{n} \sum_{i=1}^{n} \left(1 + e^{-y \sum_{j=1}^{p} \beta_j x_{ij}}\right)$$

Constructing Models with R

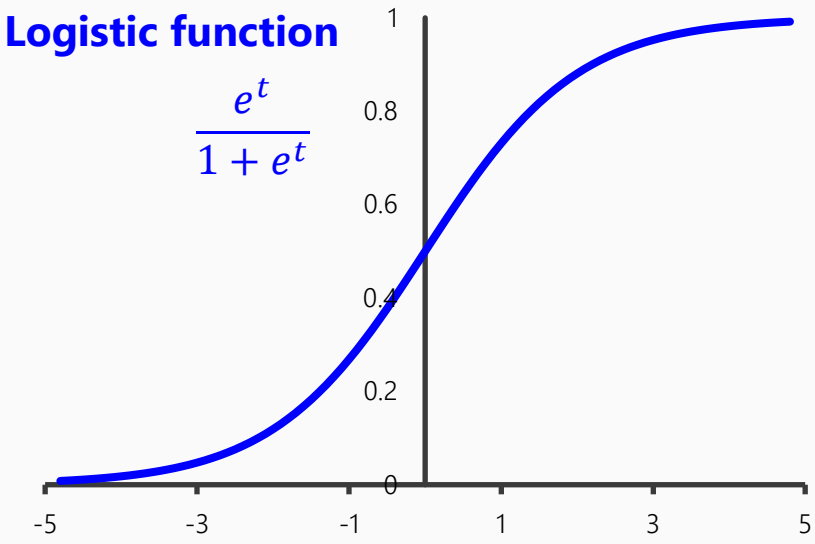
- Wide range of models available in R
- R models are defined with expressive formula language
- A formula is a language within a language
- A formula is ubiquitous across most R model types
- Labels and features can be both numeric or categorical (factor)

R Formula Language

```
lm(x~y, data = df) #The Linear Relationship between x and y
lm(x~y+z, data = df) #Adding another feature
lm(x~., data = df) # ".” indicates the usage of all features
lm(x~. - z, data = df) #To use all features except z
lm(x~0 + y + z, data = df) #The "0" drops the intercept term
lm(x~y + I(y^2) + z + I(z^2), data = df) #Utilizes function I(x)
lm(x~0 + y + z + y:z, data = df) #Adds an Interaction Term y:z
```

maximum likelihood a/b perspective

An expansion on the intuition behind **Logistic Regression**:



The **Logistic Function** models itself through a visible growth and saturation as seen in the illustration.

The function was developed in the mid-19th century by Adolphe Quetelet and his pupil, Pierre Francois Verhulst. The mathematicians were modeling the growth of populations with the intuition when countries become full, the population growth levels off and the population will saturate.

As a property known of probabilities, there can be no value above 1 or less than 0. The function $\frac{e^t}{1+e^t}$ therefore returns a probability measuring the inputs (t) to the model.

Logistic Regression enters when the probability of modeling the function will result as 1 or 0:

Predicting the value of 1: $P(Y_i = 1|x_i, \beta) = \frac{e^{\sum_{j=1}^{p} \beta_j x_{ij}}}{1+e^{\sum_{j=1}^{p} \beta_j x_{ij}}}$

Written in matrix notation: $P(Y_i = 1|x_i, \beta) = \frac{e^{x_i\beta}}{1+e^{x_i\beta}}$

Predicting the value of 1: $P(Y_i = -1|x_i, \beta) = 1 - \frac{e^{x_i\beta}}{1+e^{x_i\beta}}$

Written in simplified notation: $P(Y_i = -1|x_i, \beta) = \frac{1}{1+e^{x_i\beta}}$

The Likelihood of the observations will be calculated:

$$\text{Likelihood}(x_i, y_i) = P(Y_i = y_i|x_i, \beta)$$

Therefore:

$$\text{If } \begin{cases} P(Y_i = -1|x_i, \beta) = 1 - \frac{e^{x_i\beta}}{1+e^{x_i\beta}} = \frac{1}{1+e^{x_i\beta}} = \frac{1}{1+e^{-y_i x_i\beta}} \\ P(Y_i = 1|x_i, \beta) = \frac{e^{x_i\beta}}{1+e^{x_i\beta}} = \frac{1}{1+e^{-x_i\beta}} = \frac{1}{1+e^{-y_i x_i\beta}} \end{cases}$$

$$\text{Likelihood}(x_i, y_i) = P(Y_i = y_i|x_i, \beta)$$

$$P(Y_i = y_i|x_i, \beta) = \frac{1}{1+e^{-y_i x_i\beta}}$$

Finally adding the product property to **Logistic Regression** as follows:

$$\prod_{i=1}^{n} \text{Likelihood}(x_i, y_i) = \prod_{i=1}^{n} P(Y_i = y_i | x_i, \beta)$$

$$\prod_{i=1}^{n} P(Y_i = y_i | x_i, \beta) = \prod_{i=1}^{n} \frac{1}{1 + e^{-y_i x_i \beta}}$$

Ultimately summarized as:

$$\prod_{i=1}^{n} \text{Likelihood}(x_i, y_i) = \prod_{i=1}^{n} \frac{1}{1 + e^{-y_i x_i \beta}}$$

Proceeding to take the logarithm of each side and simplifying appropriately:

$$-\log \prod_{i=1}^{n} \text{Likelihood}(x_i, y_i) = -\log \prod_{i=1}^{n} \frac{1}{1 + e^{-y_i x_i \beta}}$$

$$= \sum_{i=1}^{n} -\log \frac{1}{1 + e^{-y_i x_i \beta}}$$

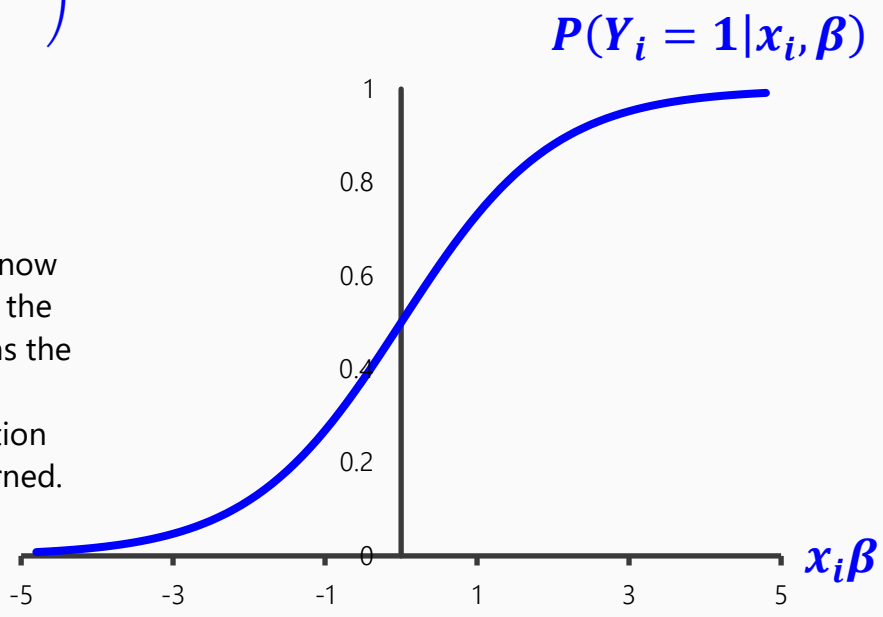
$$= \sum_{i=1}^{n} \log\left(1 + e^{-y_i x_i \beta}\right)$$

The above derivation ultimately returns us to the original **Logistic Function**:

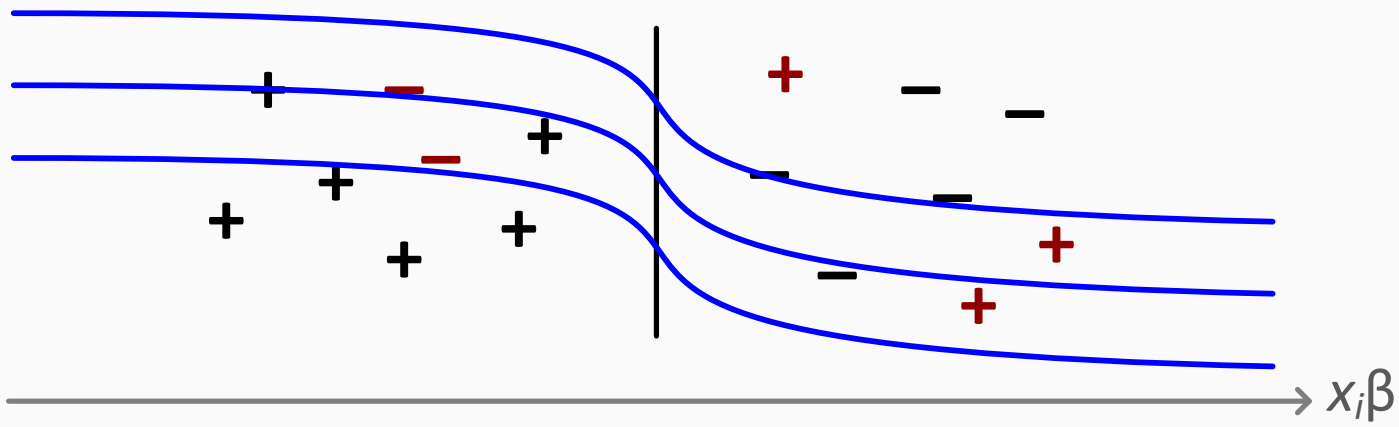
$$\min_{\beta_1, \beta_2, \ldots, \beta_p} \frac{1}{n} \sum_{i=1}^{n} \left(1 + e^{-y \sum_{j=1}^{p} \beta_j x_{ij}}\right)$$

$$P(Y_i = 1 | x_i, \beta) = \frac{e^{x_i \beta}}{1 + e^{x_i \beta}}$$

The derivation of the **Logistic Function** now provides a probabilistic interpretation of the model. Whatever score the model assigns the observation, the model also assigns the probability (likelihood). Both a Classification and Probability of Classification are returned.



The above in **Logistic Function** in geometric notation:



The geometric illustration above demonstrates a high probability received in a prediction of $y = 1$ on the left side of the decision boundary. Conversely the probability logarithmically decreases as the values move towards and over the decision boundary into the right side of the illustration.

Logistic Regression in Summary:

1. Randomly partition the data into training and test sets

2. Estimate the coefficients and train the model:

$$\hat{\beta} = \underset{\beta_1,\beta_2,\dots,\beta_p}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n}\left(1 + e^{-y\sum_{j=1}^{p}\beta_j x_{ij}}\right)$$

3. Score the model: Compute scores for each x_i in the test set

$$f(x_i) = \sum_{j=1}^{p}\beta_j x_{ij}$$

4. Evaluate the model's performance

5. Improving the performance through **Regularization** (discussed later)

$$f^* = \underset{\text{models } f}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n}\ell\big(y_i f(x_i)\big) + C + \text{Regulatization}(f)$$

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \cdots + \beta_p x_{ip}$$

$$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2 \text{ (referred to as } \ell_2)$$

$$\text{Regularization}(f) = |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1 \text{ (referred to as } \ell_1)$$

evaluation methods for classification

A **Confusion Matrix** is constructed to determine the quality of a **Classification Algorithm**:

	Actual Value $\rightarrow$ $y = 1$	**Actual Value** $\rightarrow$ $y = -1$
Predicted Value $\rightarrow$ $\hat{y} = 1$	**True Positive (+)**	**False Positive (Type I Error)**
Predicted Value $\rightarrow$ $\hat{y} = -1$	**False Negative (Type II Error)**	**True Negative (-)**

Misclassification Error (also Misclassification Rate or Accuracy), most used in Machine Learning:

$$\frac{\text{FP} + \text{FN}}{n} = \frac{1}{n}\sum_{i-1}^{n} 1_{[y_i \neq \hat{y}_i]} = \frac{\text{False Positives} + \text{False Negatives}}{\text{Total number of observations}}$$

True Positive Rate TPR (also Sensitivity or Recall):

$$\frac{\text{TP}}{\#\text{Positive}} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\sum_i^n 1_{[y_i = \hat{y}_i \text{ and } y_i = 1]}}{\sum_i^n 1_{[y_i = 1]}} = \frac{\text{True Positives (+)}}{\text{True Positives (+)} + \text{False Negatives}}$$

True Negative Rate TNR (also Specificity):

$$\frac{\text{TN}}{\#\text{Negative}} = \frac{\text{TN}}{\text{TN} + \text{FP}} = \frac{\sum_i^n 1_{[y_i = \hat{y}_i \text{ and } y_i = -1]}}{\sum_i^n 1_{[y_i = -1]}} = \frac{\text{True Negatives (}-\text{)}}{\text{False Postives} + \text{True Negatives (}-\text{)}}$$

False Positive Rate FPR:

$$\frac{\text{FP}}{\#\text{Negative}} = \frac{\text{FP}}{\text{TN} + \text{FP}} = \frac{\sum_i^n 1_{[y_i \neq \hat{y}_i \text{ and } y_i = -1]}}{\sum_i^n 1_{[y_i = -1]}} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives (}-\text{)}}$$

Precision:

$$\frac{\text{TP}}{\#\text{Predicted Positive (+)}} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\sum_i^n 1_{[y_i = \hat{y}_i \text{ and } y_i = 1]}}{\sum_i^n 1_{[\hat{y}_i = 1]}} = \frac{\text{True Positives (+)}}{\text{True Positives (+)} + \text{False Positives}}$$

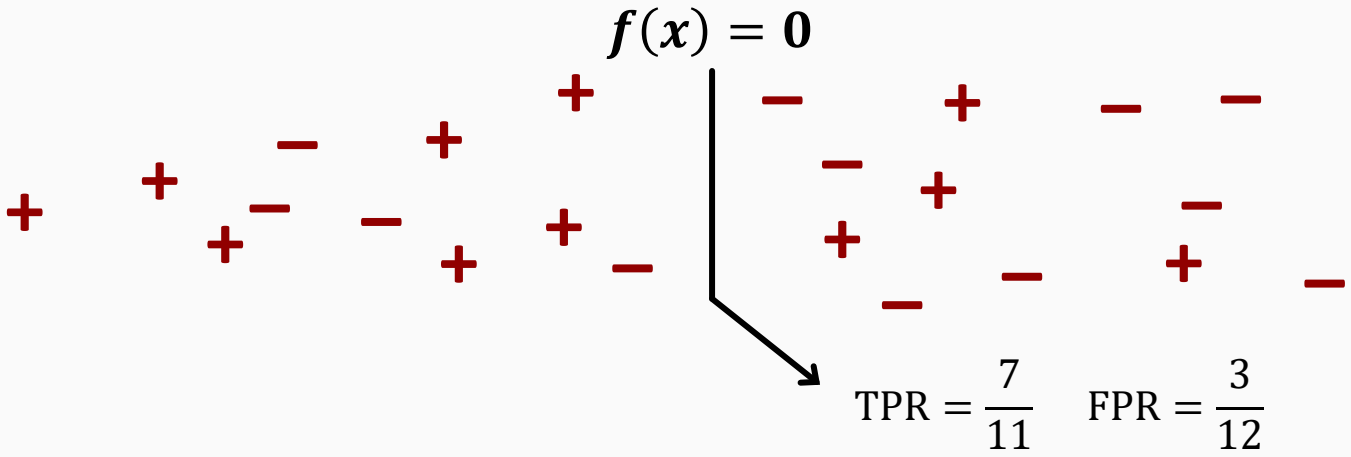
F-1 Score:

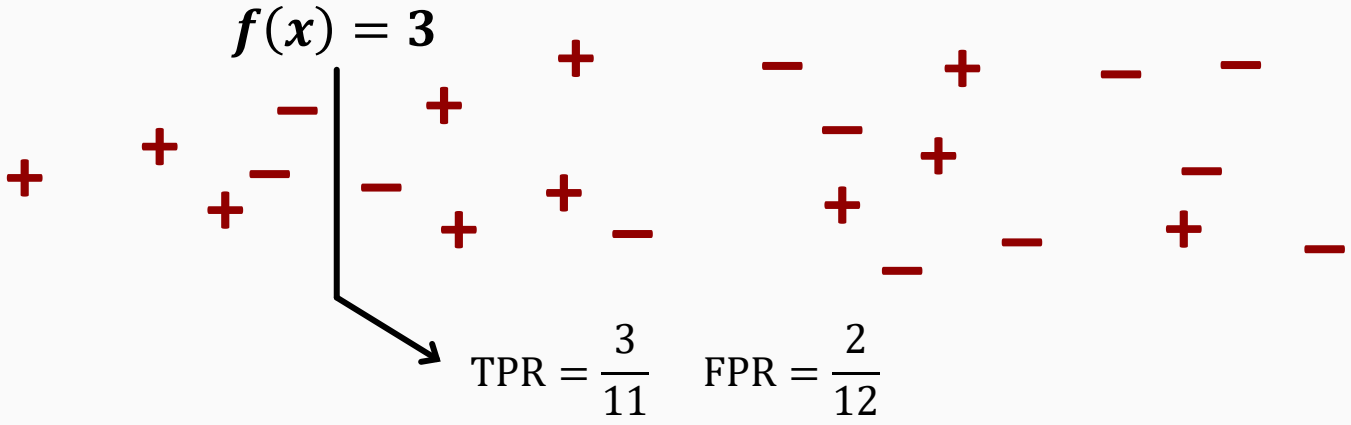
$$F1 = 2\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2\frac{\left(\frac{\text{TP}}{\#\text{Predicted Positive (+)}} = \frac{\text{TP}}{\text{TP} + \text{FP}}\right) \times \left(\frac{\text{TP}}{\#\text{Positive}} = \frac{\text{TP}}{\text{TP} + \text{FN}}\right)}{\left(\frac{\text{TP}}{\#\text{Predicted Positive (+)}} = \frac{\text{TP}}{\text{TP} + \text{FP}}\right) + \left(\frac{\text{TP}}{\#\text{Positive}} = \frac{\text{TP}}{\text{TP} + \text{FN}}\right)}$$

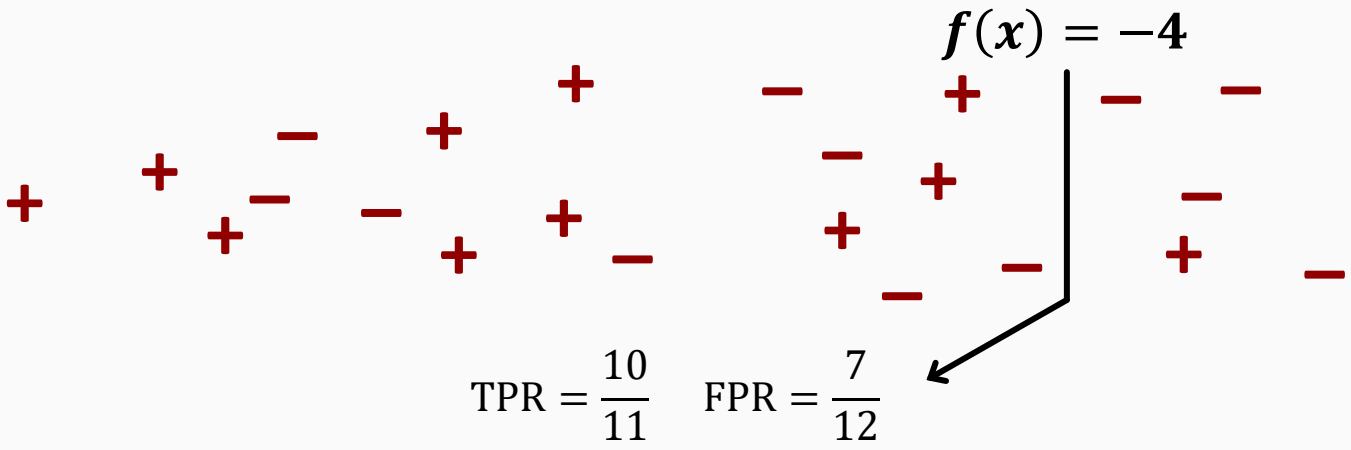
Radio Operating Characteristic (ROC) curves

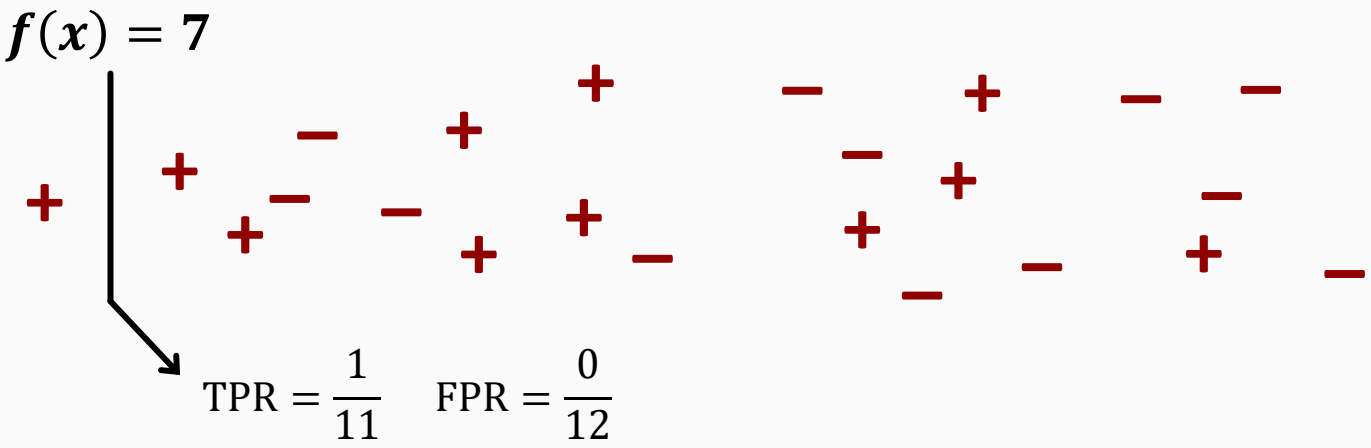
- Started during WWII for analyzing radar signals.
- For a particular False Positive Rate (FPR), what is the True Positive Rate (TPR)?
- FPR = # of negatives that were classified by the ML algorithm as positives / total # of negatives
- TPR = # of positives that were classified by the ML algorithm as positives / total # of positives

Illustrative examples in geometric notation adjusting the **Decision Boundaries** below:

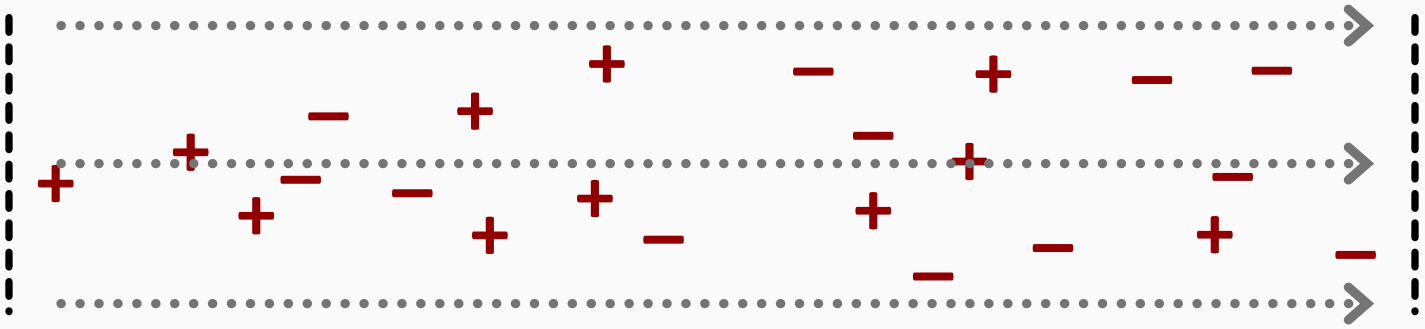






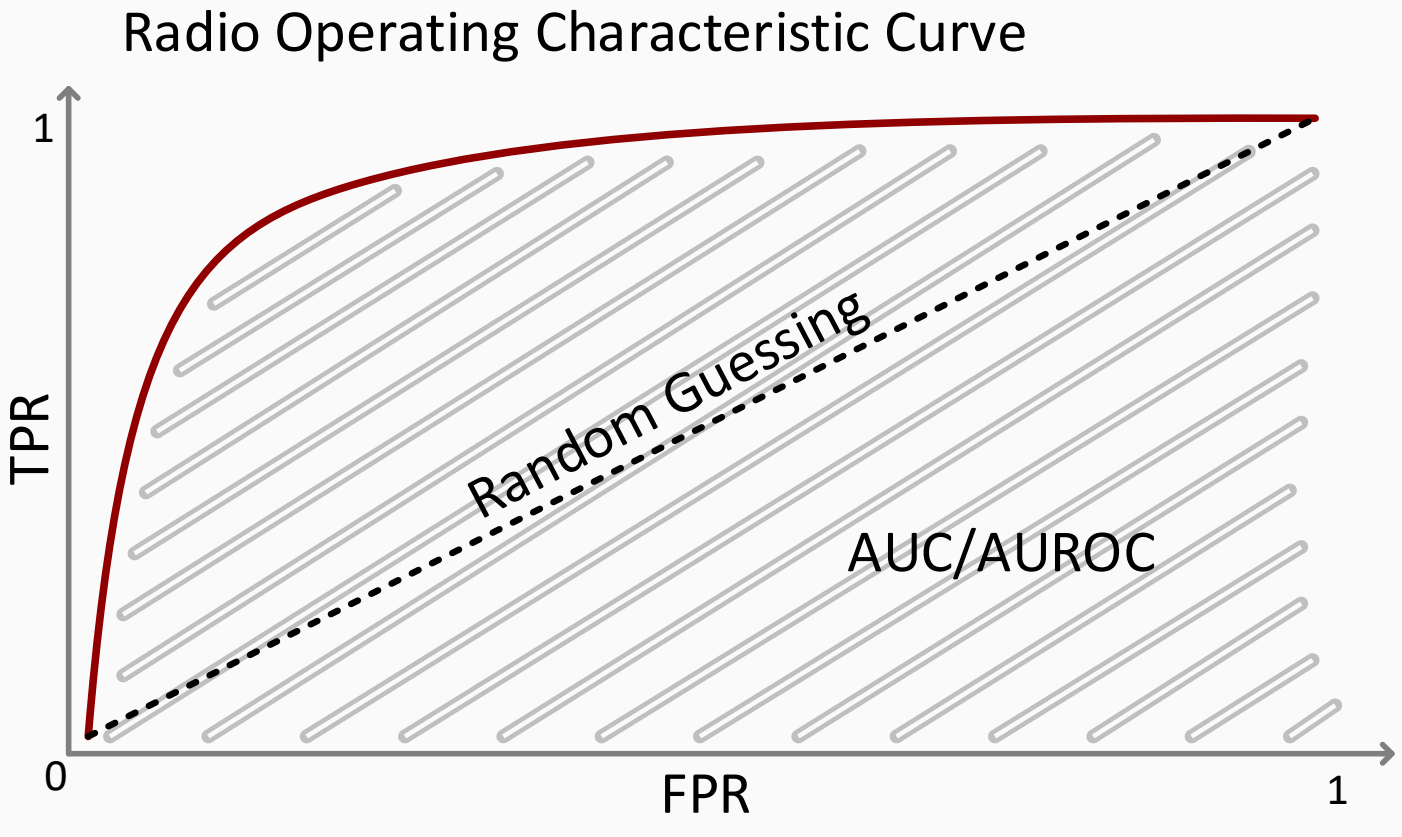


To define what an **ROC Curve** is in effect, image taking the decision boundaries computed above and sweeping the function across the entire plot below. As the boundary computes the evaluation metrics at each point, it will collect and replot a graph to represent the dynamic of evaluation metrics across the latter sweep: **(ROC Curves can be swept against both single classifiers and algorithms)**



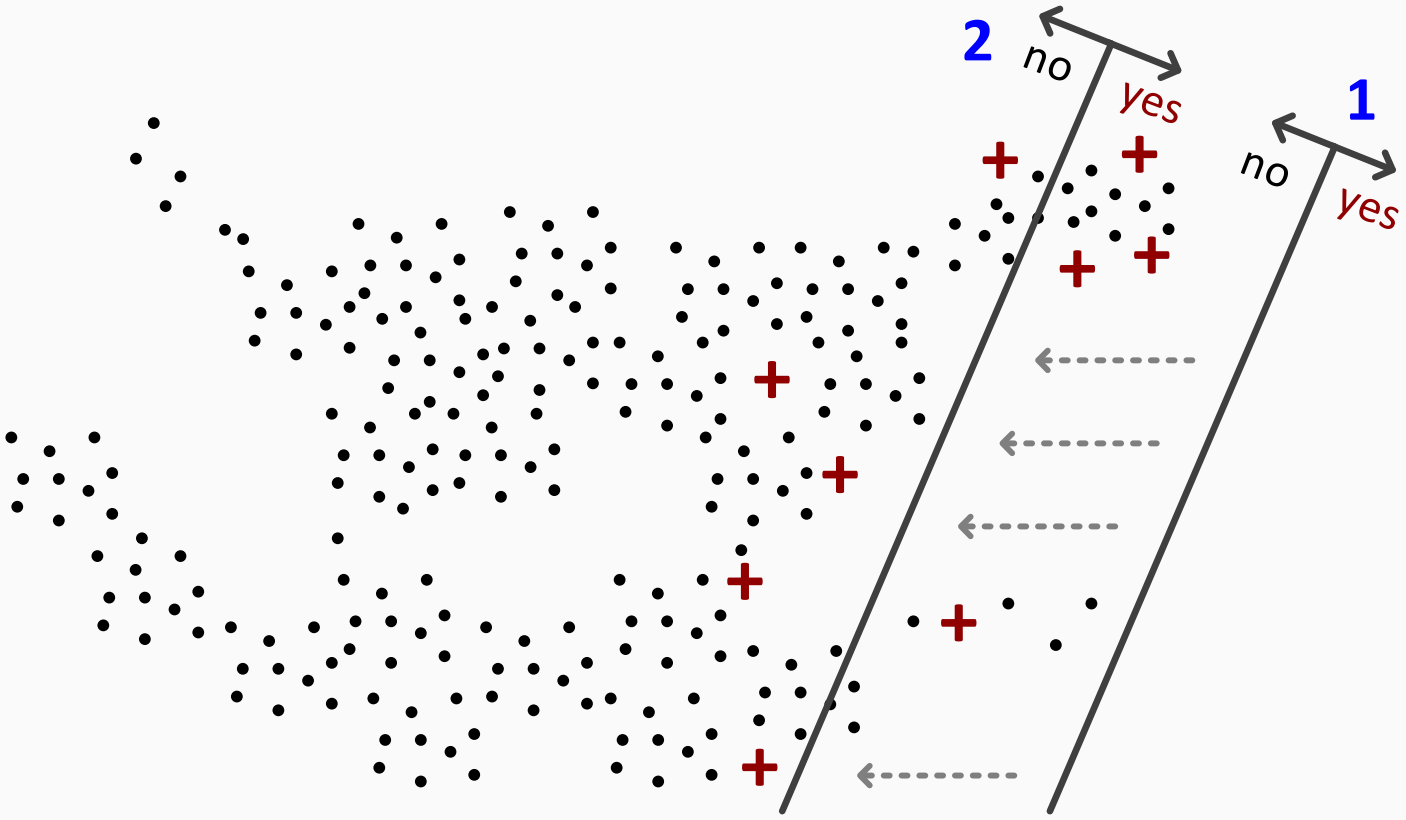
ROC Curve

For a given False Positive Rate (**FPR**), what is the True Positive Rate (**TPR**)?



imbalanced data

The following illustration represents an imbalanced data set and a classification model applied:



The **Classification Model 1** represents a likely model the initial data will produce. It is noted that the predictions **Classifier 1** will make is **"no"** or **0** for every example in the dataset. This appears to be highly flawed, however the resulting model is **99% accurate** considering that only **1%** of the examples exhibit a **"yes"** or **1** response.

Regardless of the accuracy, the model is degenerative or trivial in nature; the model is essentially meaningless because it cannot interpret data to predict anything of value for future observations.

The alternate **Classification Model 2** will obviously have a much higher **Misclassification Rate** and predict a large number of **False Positives (+)**. However, the ability for **Classifier 2** to capture the existence of a positive result is much more valuable than the loss of accuracy in predicting negatives. The scarcity of the positive examples in the dataset prove the value of the latter.

The way to achieve **Classification Model 2** from the resulting dataset is to weight the value of **positive (+)** results appropriately. Weighting the algorithm is achieved through several methods of which **Regularization** is the most widely utilized:

$$\frac{1}{n}\sum_{i=1}^{n}\ell\left(y_i f(x_i)\right) + \text{Regulatization}(f)$$

The above **objective function** will treat the **positives (+)** equally to that of the **negatives (-)** by segregating the two types of labels as follows:

$$\frac{1}{n}\left(\boldsymbol{C} \sum_{\substack{i \text{ positives} \\ i \text{ where } y_i = 1}}^{n} \ell\left(y_i f(x_i)\right) + \sum_{\substack{k \text{ negatives} \\ k \text{ where } y_k = 1}}^{n} \ell\left(y_k f(x_k)\right) \right) + \text{Regulatization}(f)$$

The above notation dictates each **positive (+)** example is weighted as $\boldsymbol{C}$ × **a negative (-)** example.

The Classifier will now favor the **Classification Model 2** as opposed to **Classification Model 1**.

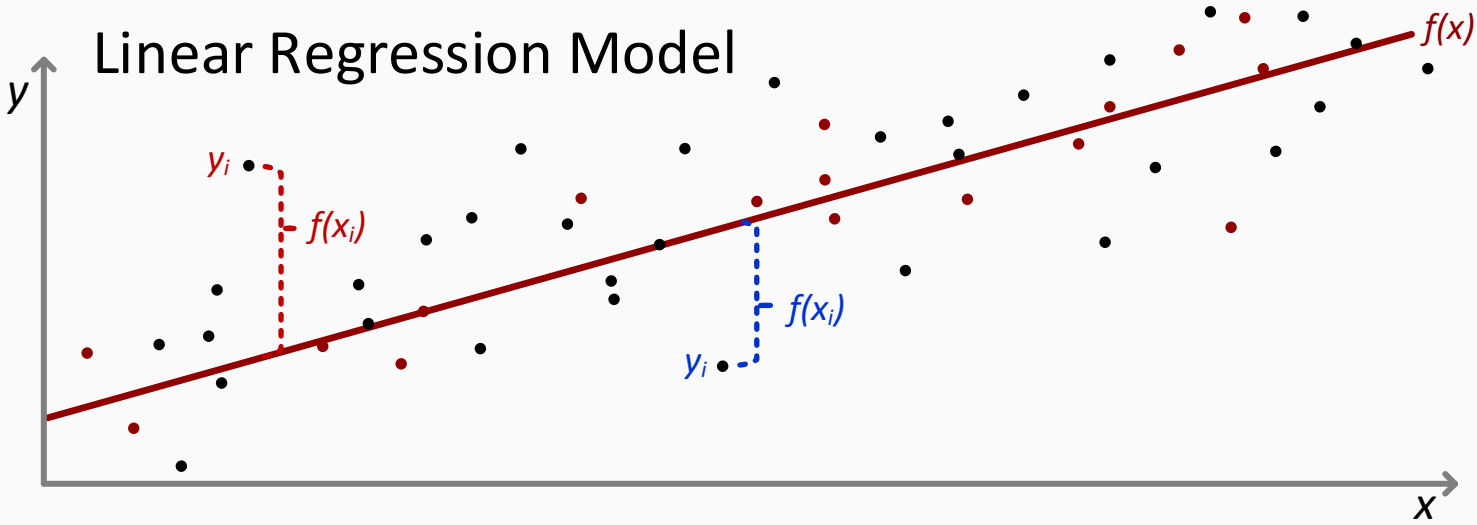
In summary, instead of relying on accuracy as an absolute evaluation metric, adjust **the imbalance parameter** $\boldsymbol{C}$ to obrain an ideal balance between **True Positives (+)** and **False Positives** $= \frac{\mathbf{TP}}{\mathbf{FP}}$.

Additionally to using **the imbalance parameter** $\boldsymbol{C}$ to adjust the model interpretation, the parameter C can equally be used to evaluate the performance of an entire algorithm itself **(discussed later)**.

module2· regression

linear regression 📐 for machine learning

Linear Regression is used to predict real-valued outcomes (continuous variables)



The simplest form of **Linear Regression** is an equation that consists of a single linear coefficient and a constant fit the model data to a line; a function is applied in order to estimate $\boldsymbol{y}$ for each observation of $\rightarrow \boldsymbol{f(x_i) = b_0 + b_1 x_i}$. In order to fit the best model possible to the data, the model needs to be adjusted in response to the error that can be measured.

Error in the above model can be measured as the distance from a point to the linear model prediction. However, there are two orientations of error as seen above: $\boldsymbol{y_i - f(x_i)}$ and $\boldsymbol{f(x_i) - y_i}$. This application is a fallacy considering that either way will not account for the error of the other. An alternative would be to use the absolute error $|\boldsymbol{f(x_i) - y_i}|$, however the latter remains inadequate. The solution to measure the model's error is the **Sum of Least Squares** error $\boldsymbol{\left(y_i - f(x_i)\right)^2}$.

With the **SSE** applied to a linear model, the optimization objective of minimizing error is achieved.

Single Variable Linear Regression:

$$\boldsymbol{f(x_i) = b_0 + b_1 x_i}$$

Choose $\boldsymbol{b_0}$ and $\boldsymbol{b_1}$ to minimize the total error on the training set:

$$\mathrm{SSE}(f) = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2 = \mathrm{SSE}(f) = \sum_{i=1}^{n} \left(y_i - (\boldsymbol{b_0 + b_1 x_i})\right)^2$$

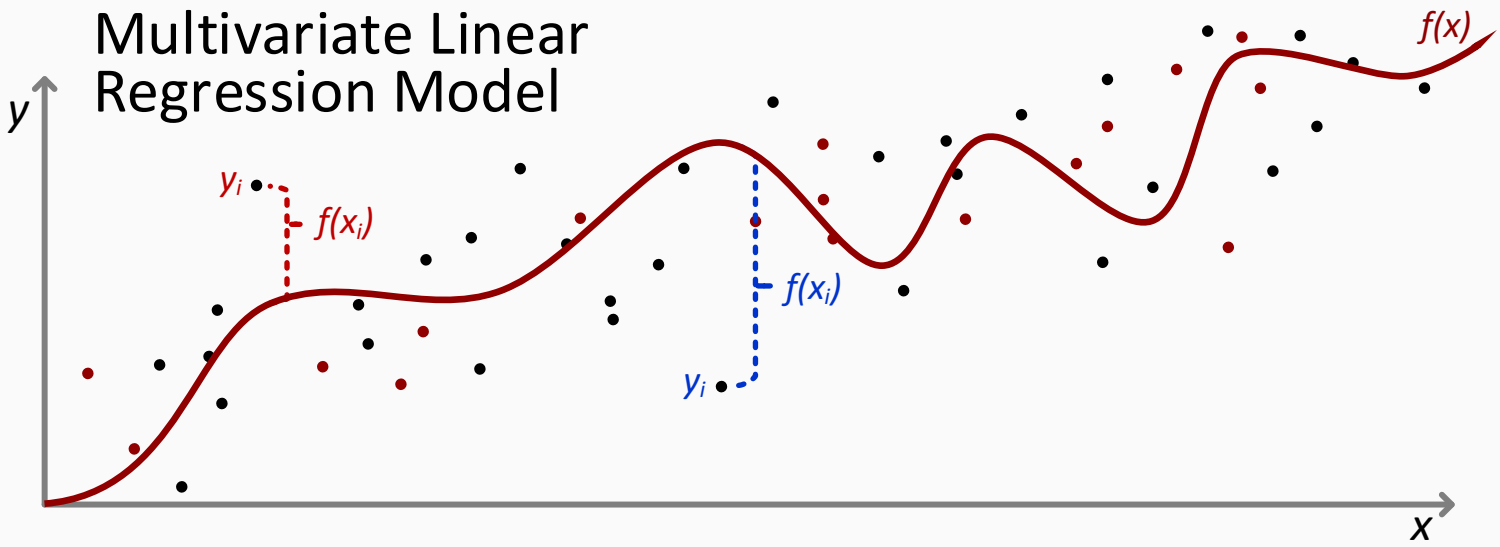
multivariate linear regression for machine learning

Multivariate Linear Regression expands upon simple linear regression when multiple weighted coefficients are added in perpetuity to the algorithm.

$$f(x_i) = b_0 + b_1 x_i \quad \rightarrow \quad \boldsymbol{f(x_i) = b_0 + b_1 x_{i,1} + b_2 x_{i,2} + \cdots + b_p x_{i,p}}$$

Method of Least Squares choosing coefficients to minimize:

$$\text{SSE}(f) = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2 = \text{SSE}(f) = \sum_{i=1}^{n} \left(y_i - \left(\boldsymbol{b_0 + b_1 x_{i,1} + b_2 x_{i,2} + \cdots + b_p x_{i,p}}\right)\right)^2$$

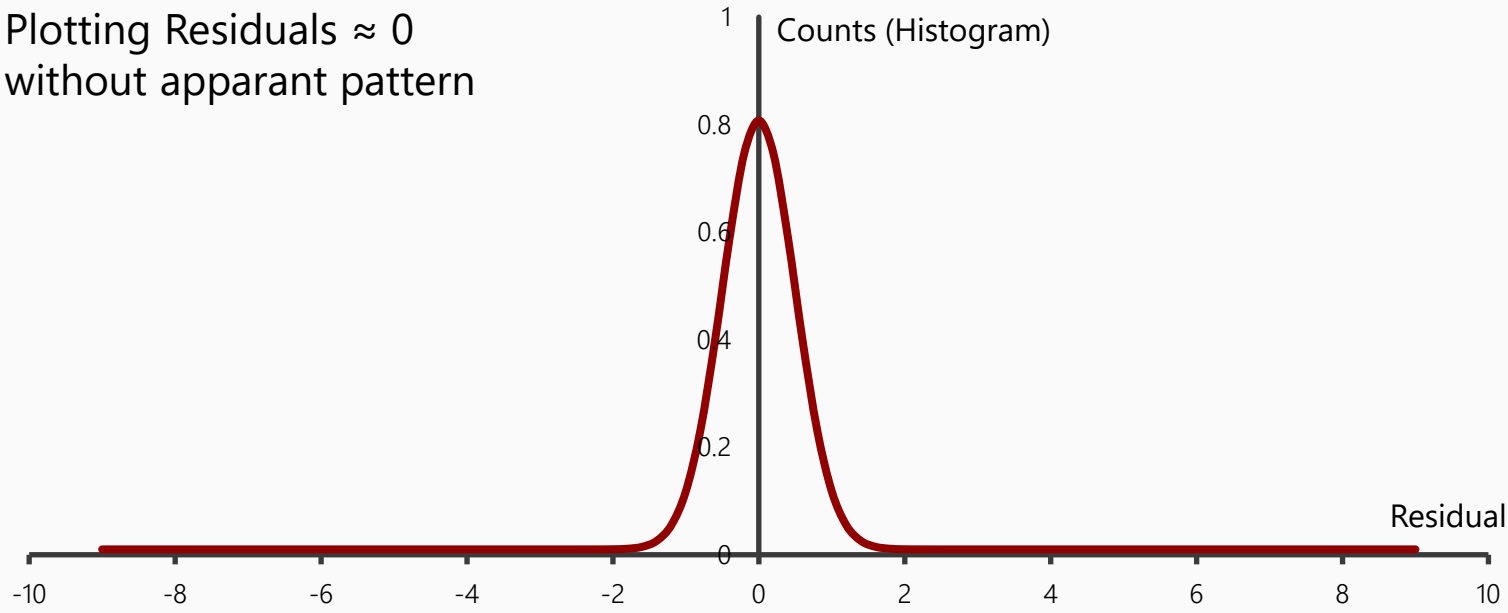


Multivariate Linear Regressive models can fit data in a much more dynamic manner as seen above. This is achieved through the use of polynomial, quadratic, trigonometric, and other various functions.

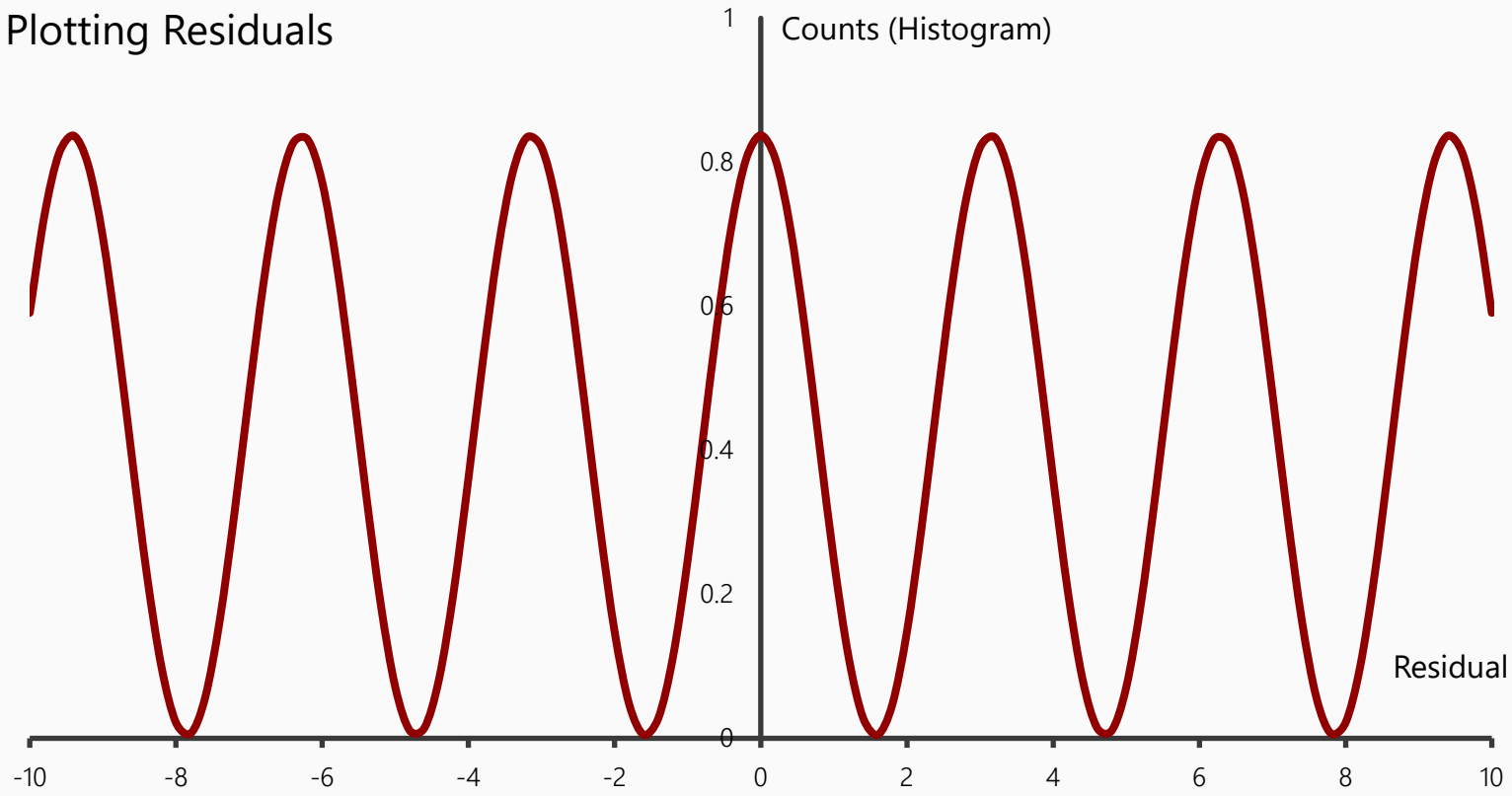
evaluation ⇄ regression models

The **Residual** or **Error** is the difference between the predicted and actual values $f(x_i) - y_i$.

The reported Residuals are ideally ≈ 0 and will lack any specific structure or pattern amongst them:

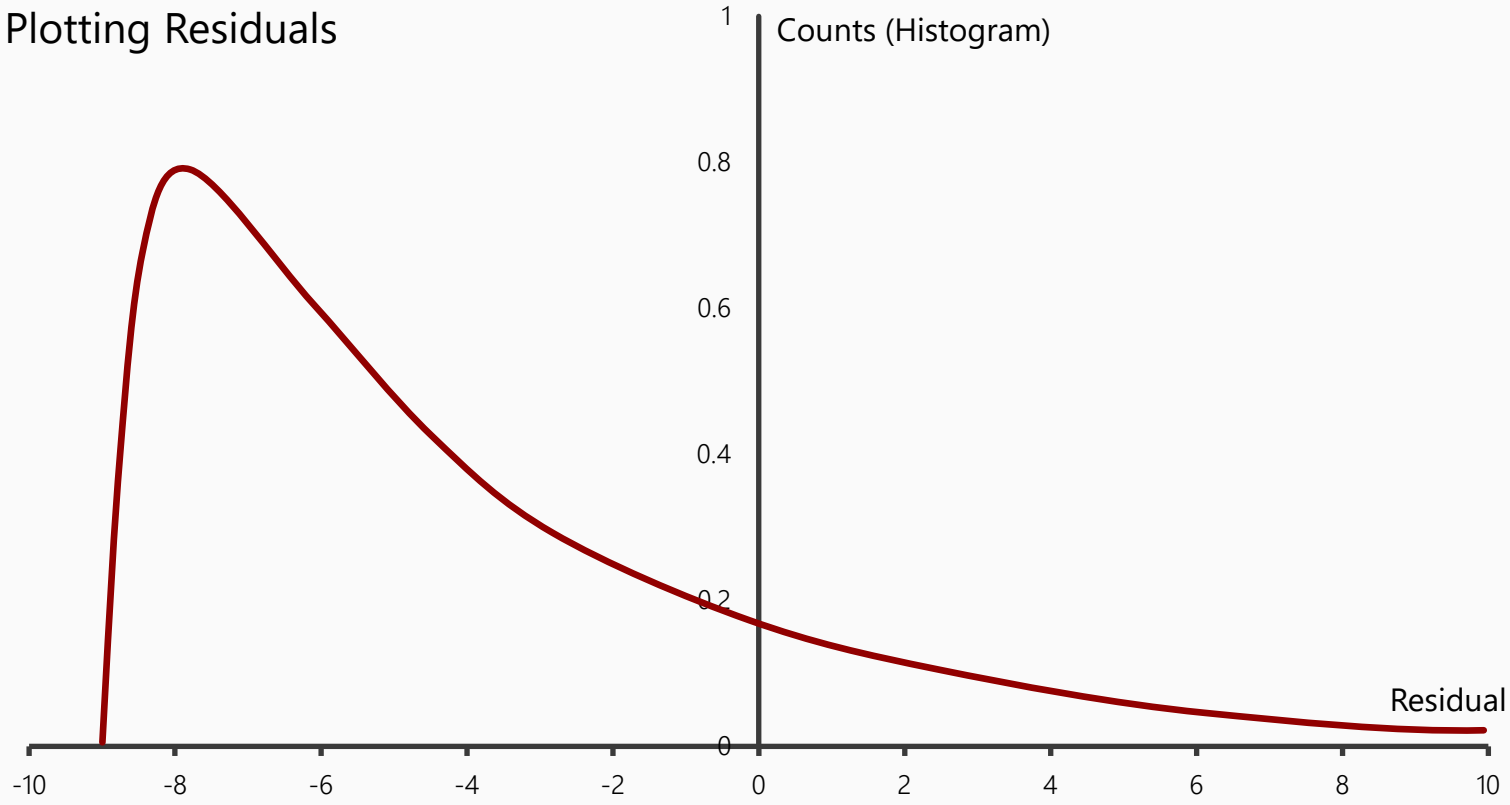


Plotted Residuals displaying an apparent pattern indicates missed properties in modeling the data:



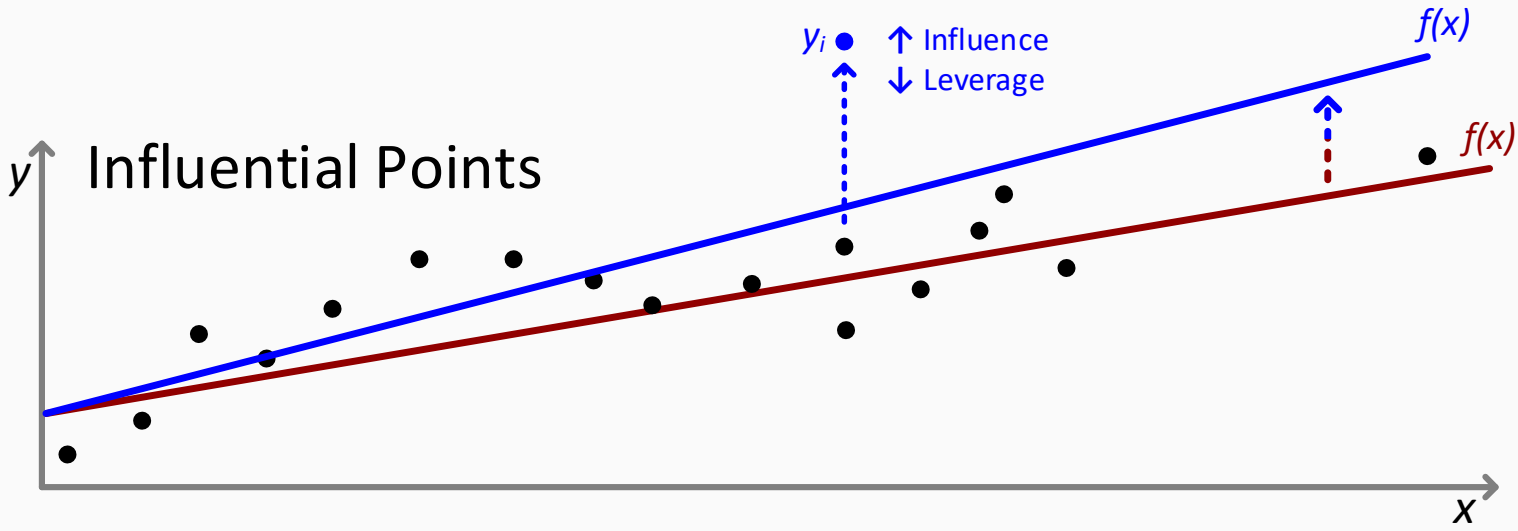
The above model could result from multiple linear features modeled inappropriately causing signal.

Plotted examples displaying a skew in either direction indicates a generally poorly performing model:

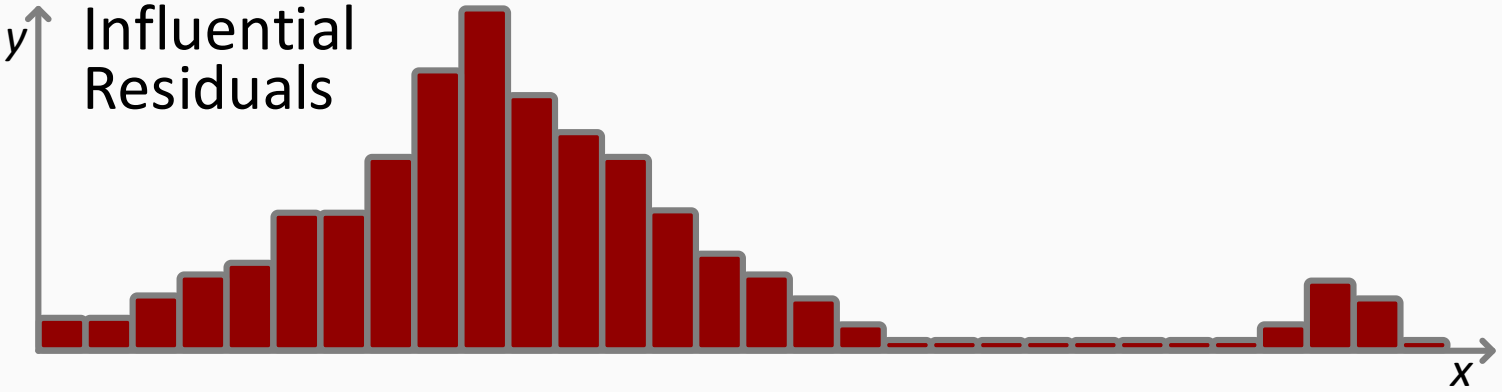


influential ≋ points

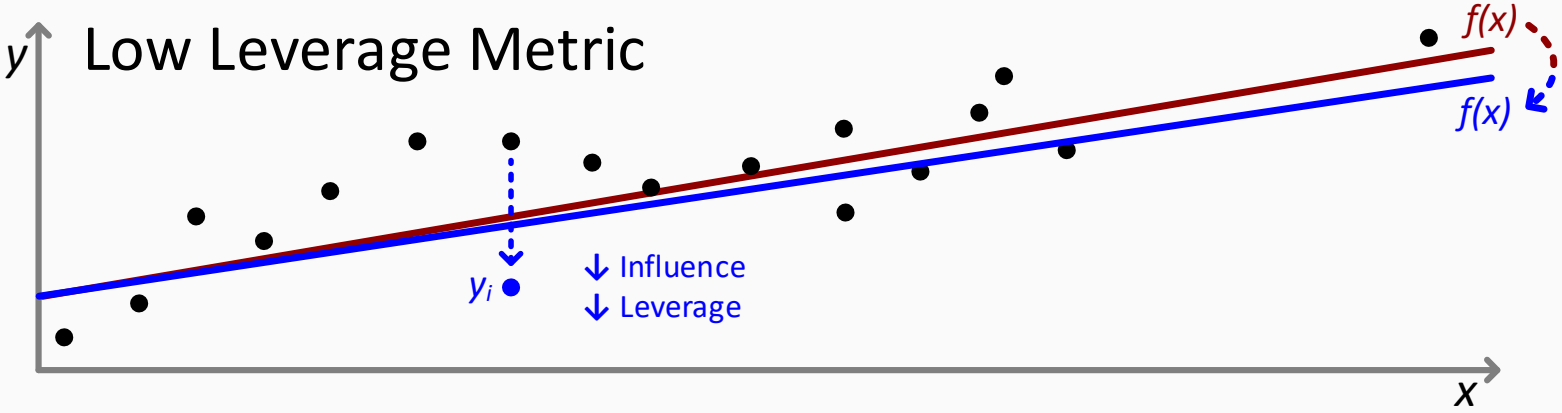
An **Influential Point** changes a model's predicted values significantly when omitted or altered:



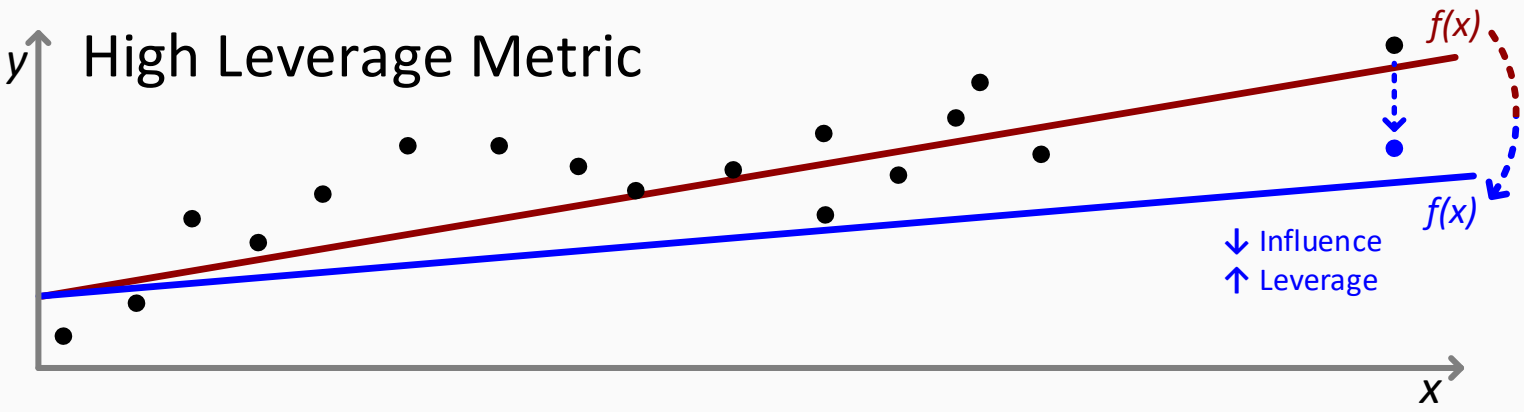
Equally, **Influential Points** significantly impact a plot of residuals seen in the histogram below:



Alternatively, **Leverage** is where a point can be altered in multiple ways without having a significant impact on the model's predictions or the plotted residual distribution as seen in the illustration below:



However, depending on the location of the point in question, **Leverage** can be substially increased. Any alteration of such a point will cause the model to alter in a more significant nature seen below:



Formally, if point x_i is moved, and $f(x_i)$ moves proportionally, the proportionality constant is referred to as the leverage of point i.

Leverage depends on x_i but does not depend on y_i. Leverage depends on how far away x_i is from the mean of the points x_i *'s*.

It is important to note the **Influential Points** do not necessarily have a high amount of **Leverage**. Alternatively, High **Leverage** points are not necessarily **Influential Points**.

Ultimately, the **Leverage** of point x_i measures the impact of x_i on $f(x_i)$.

Investigating the nature of outliers typically involves asking questions such as:

- Is the data correct and represented as such?
- Is the model complete and performing sufficiently?

module3 · evaluation machine learning models

feature ▦ selection

Ockham's Razor states the best models are simple while fitting datasets appropriately.

A balance to achieve between accuracy and simplicity to mitigate the "curse of dimensionality"

More simple models:

- tend to predict better labels
- more interpretable to humans
- easier to make predictions from (considering less calculations)

However, selecting the optimal combinations of features is computationally difficult to apply.

Standard **Feature Selection** methods exist to remove irrelevant and redundant features from models.

Greedy Backward Selection

- Begin with all of the features in a dataset
- Find the most feature that hurts predictive power the least after removed → **remove it**
- Reiterate the process until some determined criterion is met

The process is referred to as "**greedy**" because removed features are never returned in this process.

Greedy Forward Selection

- Begin with none of the features in a dataset
- Find the single most valuable feature towards prediction power → **include it**
- Reiterate the process until some determined criterion is met

The process is effectively the converse of **Greedy Backwards Selection**. Features continually added in the process where the prediction power increase less each time, the result is **diminishing returns**. Thus the two perquisite criterion to select in the process are that of how to select features. This is typically done with the feature that **boosts accuracy** the most. The other is a **stop criterion** resulting from **diminishing returns**; typically measured with **Adjusted R^2**.

Adjusted R^2

R^2 is the measure of how well the model fits the dataset (**correlation**):

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}} = 1 - \frac{\sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \qquad \rightarrow \qquad \textbf{Goodness of fit}$$

If the model is always measured closely to the data, then $\boldsymbol{y_i}$ and $\boldsymbol{f(x_i)}$ are close. Thus $\boldsymbol{y_i - f(x_i)}$ will be $\boldsymbol{\approx 0}$ and $\boldsymbol{R^2}$ will be $\boldsymbol{\approx 1}$ and determine a measure of **goodness of fit**. The denominator of the $\boldsymbol{R^2}$ function does not depend on the model as it is simply a property captured of the data; it has no opinion on the model performance or correlation.

$\bar{R}^2$ is the **Adjusted R^2**. The $\bar{R}^2$ measure penalizes R^2 depending on the number of terms in a model.

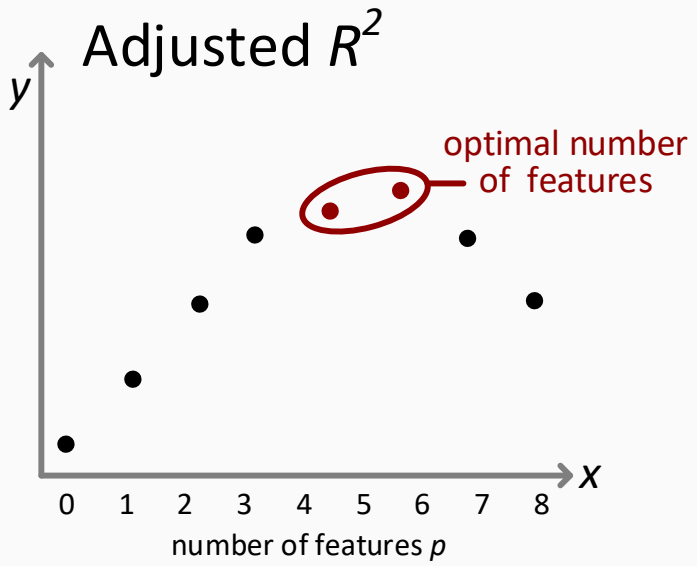
$$\bar{R}^2 = R^2 - (1 - R^2)\frac{p}{n - p - 1} \quad \rightarrow \quad \textbf{Goodness of fit and complexity}$$

Therefore the **Adjusted R^2** both **sparse** and **accurate** models.

For example, if p is **large** → $\bar{R}^2 = R^2 - (1 - R^2) \times$ **Large Penalty** = **Small** $\bar{R}^2$

However, if p is **small** → $\bar{R}^2 = R^2 - (1 - R^2) \times$ **Small Penalty** = **? Depends on** R^2

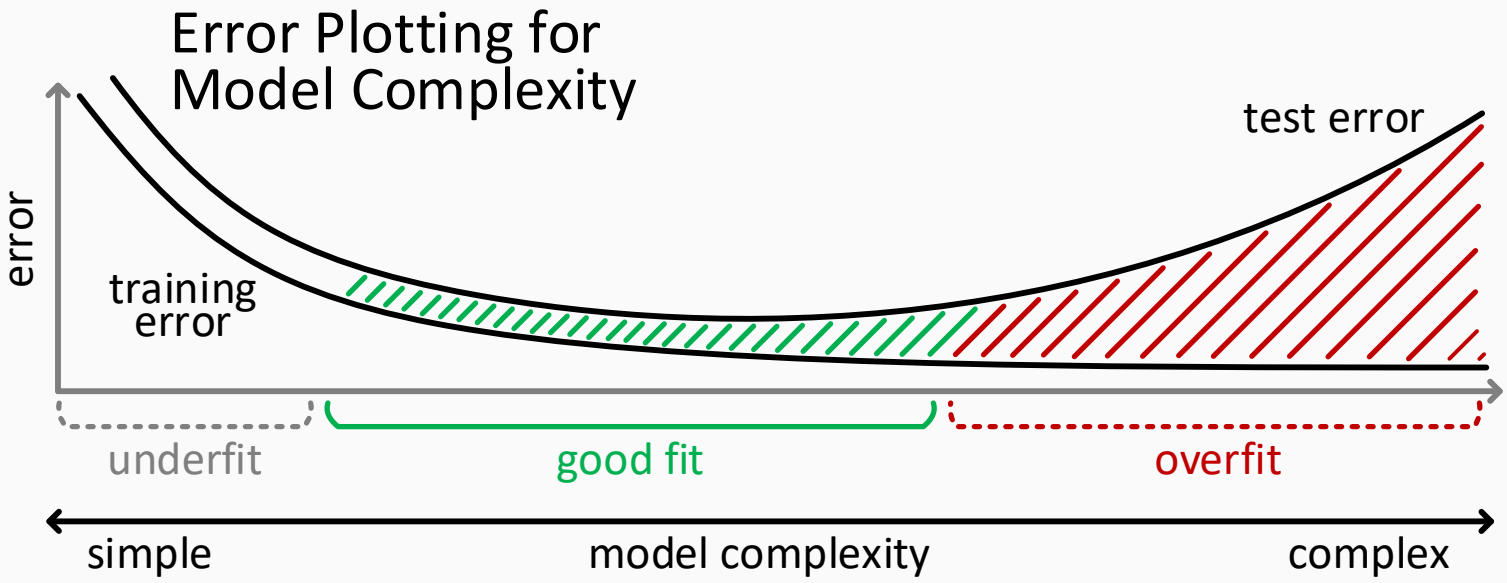
The reasoning influencing the **Adjusted R^2** being ambiguous of a model with few features is because a model will a small amount of features might possible have a small R^2 in the first place, making the measurement of the **Adjusted R^2** potentially equally small; The solutions returns to the concept of **Ockham's Razor** stating that a balance must be achieved between the correlation and the amount of features utilized in a model, in the latter context.



regularization for overfitting algorithms

Regularization is one of the most significant reasons machine learning models can **generalize**.

Statistical Learning Theory dictates that in order to keep the error small on a test dataset, the model needs to be accurate on the training set while maintaining a certain degree of simplicity:



Allowing a model to become **overly complex** to minimize the **training error** leads to **overfitting**.

Regularization in turn, prevents **overfitting** of the training set through limiting model **complexity**.

Simplicity of a given model is measured by the **Regularization Term**; with significant constant $\boldsymbol{C}$.

$$\frac{1}{n}\sum_{i=1}^{n}\ell\big(y_i f(x_i)\big) + \boldsymbol{C} \times \text{Regulatization}(f)$$

The **Regularization Constant**; constant $\boldsymbol{C}$ is the determinate in balancing accuracy and simplicity.

$$\min_{\text{models } f}\frac{1}{n}\sum_{i=1}^{n}\ell\big(y_i f(x_i)\big) + C + \text{Regulatization}(f)$$

Choose a linear model:

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \cdots + \beta_p x_{ip}$$

To **Regularize** (**simplify**):

$\beta_1, \beta_2, \beta_3, \ldots, \beta_p$ should be small

$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2$ (referred to as ℓ_2)

$$\text{Regularization}(f) = |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1 \text{ (referred to as } \ell_1)$$

A Single Dimensional Example:

$$\min_{\text{models } f} \frac{1}{n}\sum_{i=1}^{n} \ell\big(y_i f(x_i)\big) + C + \text{Regulatization}(f)$$

Choose a linear model:

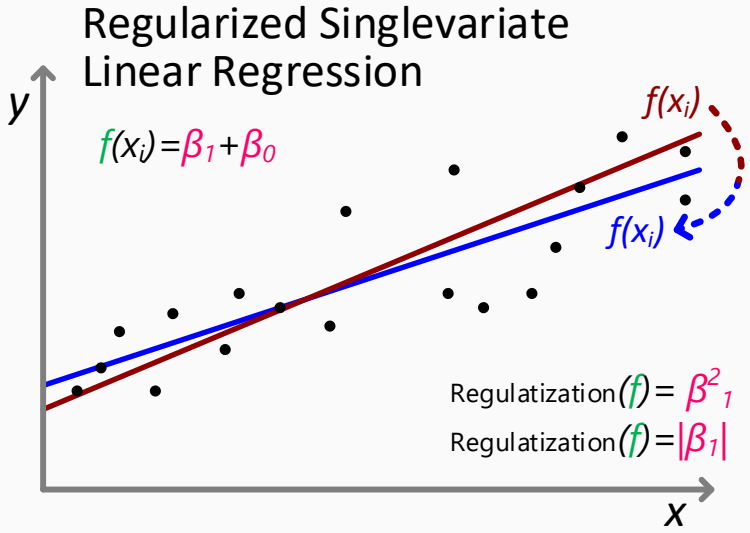
$$f(x_i) = \beta_0 x_i + \beta_1$$

To **Regularize** (**simplify**):

β_1 should be small

$\text{Regularization}(f) = \beta_1^2$ (referred to as ℓ_2)

$\text{Regularization}(f) = |\beta_1|$ (referred to as ℓ_1)



Regularization attempts to flatten the linear model as much as it acceptably can. The intent is to avoid influence by excess variance in the dataset. The **Regularization intuition** is more evident if a higher dimensional polynomial term is used in place of a single variable linear expression.

A Single Dimensional Example with a Polynomial Expression:

Choose a linear (polynomial) model:

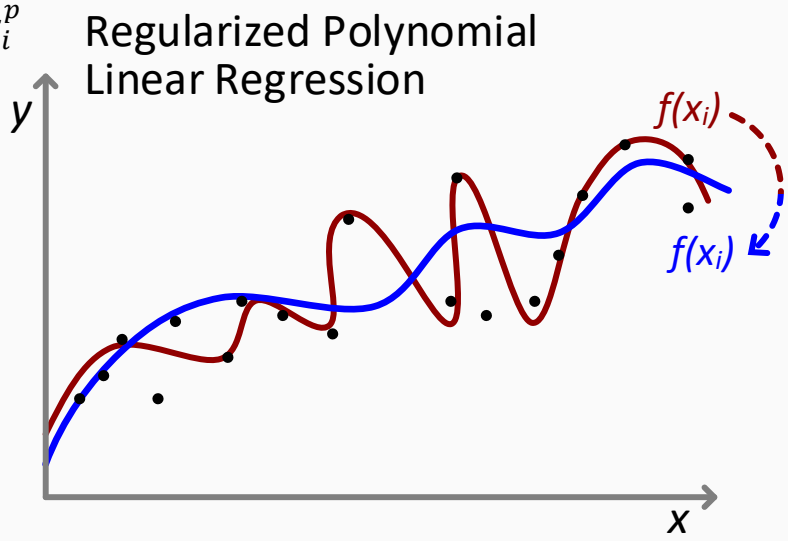
$$f(x_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_i^2 + \beta_3 x_i^3 + \cdots + \beta_p x_i^p$$

To **Regularize** (**simplify**):

$\beta_1, \beta_2, \beta_3, \ldots, \beta_p$ should be small

$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2$ (referred to as ℓ_2)

$\text{Regularization}(f) = |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1$ (referred to as ℓ_1)



The Difference Between ℓ_1 and ℓ_2 Regularization

$$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2 \text{ (referred to as } \ell_2)$$

ℓ_2 Regularization intuitively tends to make all coefficients slightly smaller.

$$\text{Regularization}(f) = |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1 \text{ (referred to as } \ell_1)$$

ℓ_1 Regularization is particularly useful for making sparse solutions; setting many coefficients $= 0$.

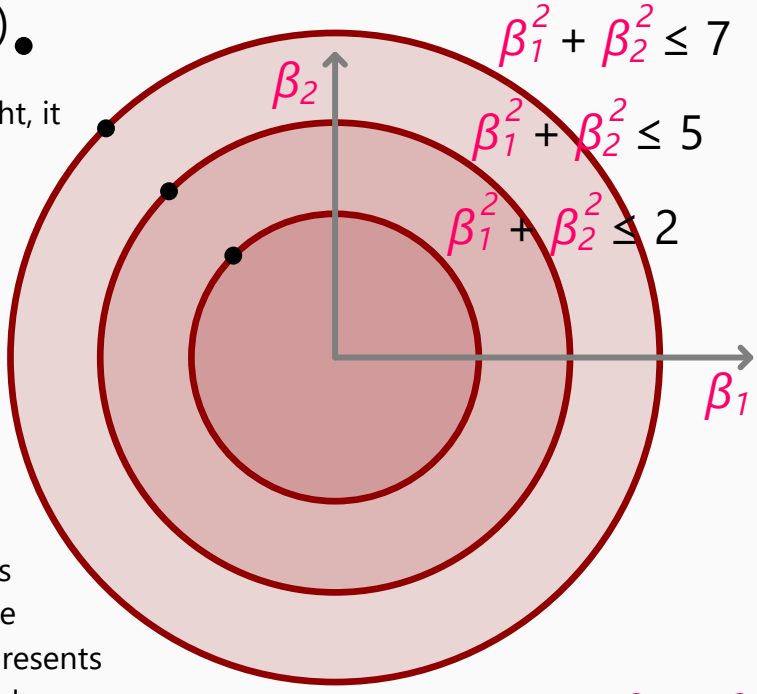
A Geometric 2-Dimensional Example Using $\boldsymbol{\ell_2}$ Regularization:

$$(\beta_1^{TrainingError}, \beta_2^{TrainingError})$$

Plotting both β_1 and β_0 in the illustration to the right, it can be seen how the summation term $\beta_1^2 + \beta_2^2$ attempts to maintain the as close to the origin as possible. For example, the first bound sets the **Regularization term** relatively small; having enough Regularization that the summation is at most **5**. This model is likely **underfitted**.

In this case, the **Regularization term** is attempting to choose the point on the smallest, inner most circle.

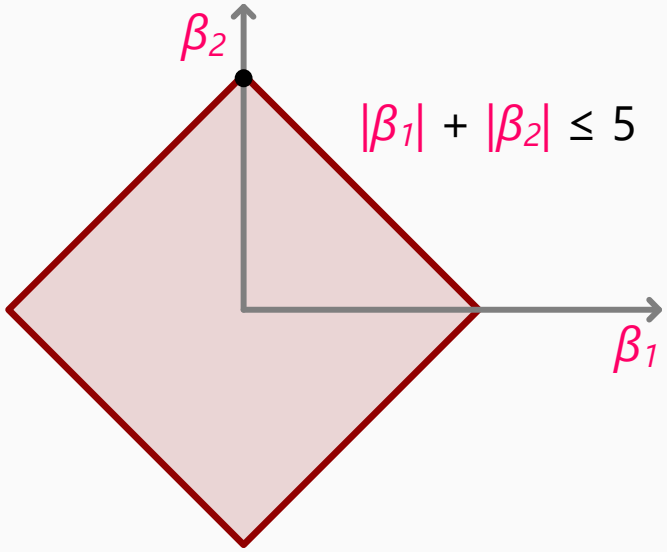
However, the term also prefers to be as **accurate** as possible; illustrating the most accurate model as the point at $\beta_1^{\text{TrainingError}} + \beta_2^{\text{TrainingError}}$. This point represents the model that is the most **overfitted** because it only succeeds at minimizing the **Training Error**. Therefore, the circular gradients ultimately represents regularization options to choose a term that results in a balance between accuracy and simplicity when fitting a model to dataset.



ℓ_2: Regularization(f) = $\beta_1^2 + \beta_2^2$

A Geometric 2-Dimensional Example Using ℓ_1 Regularization:

$(\beta_1^{TrainingError}, \beta_2^{TrainingError})$.



ℓ_1: Regularization(f) = $|\beta_1| + |\beta_2|$

In regards to ℓ_1 **Regularization**, the gradient actually produces a diamond shape as seen in the illustration to the right. This is due to the **Regularization term** consisting of the absolute values of the term $|\beta_1^2| + |\beta_2^2|$. The term that produces the optimal training error is often one of the terms at the outermost points of the diamond. Specifically illustrated in the model is the optimal point being that of term that sets $\beta_1 = 0$.

This is the intuition of ℓ_1 **Regularization**. Specifically, in higher dimension spaces, ℓ_1 **Regularization** tends to prefer sets of coefficients that have many set to 0; the resulting model becomes consequentially sparse with many irrelevant features for making predictions.

Setting up a problem using a linear model and ℓ_1 **Regularization**:

$$\min_{\text{models } f} \frac{1}{n}\sum_{i=1}^{n} \ell\big(y_i f(x_i)\big) + C + \text{Regulatization}(f)$$

where $f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \cdots + \beta_p x_{ip}$

$$\begin{aligned}
\beta_1 &= 0.7 \\
\beta_2 &= 0 \\
\beta_3 &= 0 \\
\beta_4 &= 0 \\
\beta_5 &= 0 \\
\beta_6 &= 5.6 \\
&\vdots \\
\beta_p &= 0
\end{aligned}$$

The solution to the problem exhibits many of the betas being set = 0; the features are automatically selected, leaving the user to consider only β_1, β_1 and whichever other betas are $\neq 0$.

Regularization(f) $= \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2$ (referred to as ℓ_2)

ℓ_2 **Regularization** is also referred to as **Ridge Regression**.

Regularization(f) $= |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1$ (referred to as ℓ_1)

ℓ_1 **Regularization** is also referred to as the **Lasso Penalty**.

Regularization in general is often referred in whole as **shrinkage**.

Regularization in Summary

Regularization is an alternative to feature selection. In summary, regularization forces the weights of less important features toward zero. This process helps to ensure that models will generalize in production. Regularization can be performed on models with very large numbers of features, for which manual feature selection is infeasible.

Analogously with manual feature selection, the largest regularization weights which do not affect model performance are preferred. Larger regularization weights force the model coefficients for less important features toward zero. The lower influence of less important features helps to ensure that the model will generalize.

interpreting features

Selecting the best features is essential to the optimal performance of machine learning models. Only features that contribute to measurably improving model performance should be used.

Using extraneous features can contribute noise to training and predictions from machine learning models. This behavior can prevent machine learning models from generalizing from training data to data received in production.

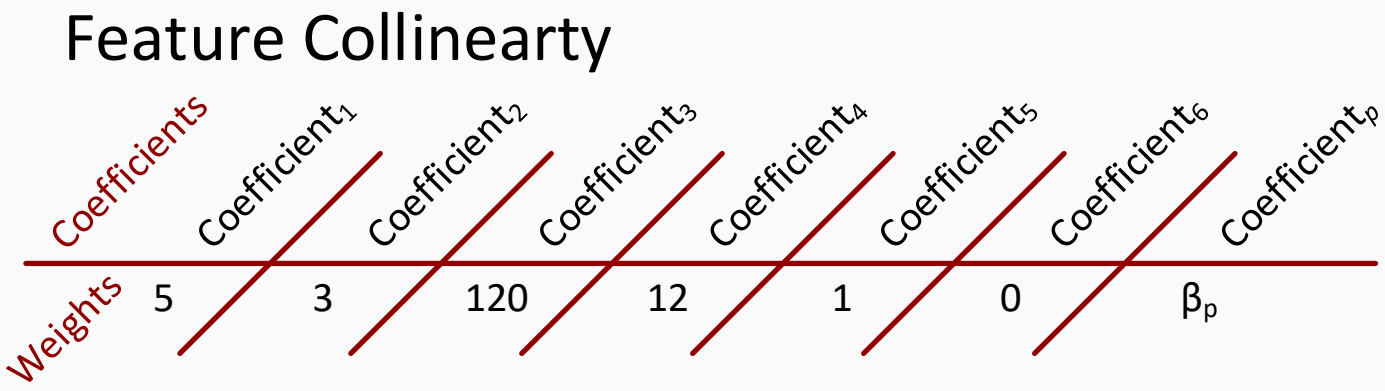
Collinear Features are features in which are highly correlated amongst themselves; they are essentially the same and thus will have comparable predictive power.

> **Collinear Features** are often the product of data scientists engineering new features to learn more information for a given topic or property of a dataset or domain.

However, **Collinear Features** often distort the interpretation of model parameterization (coefficients).

It is important to note that coefficients are not relevant to measuring their importance in a model.

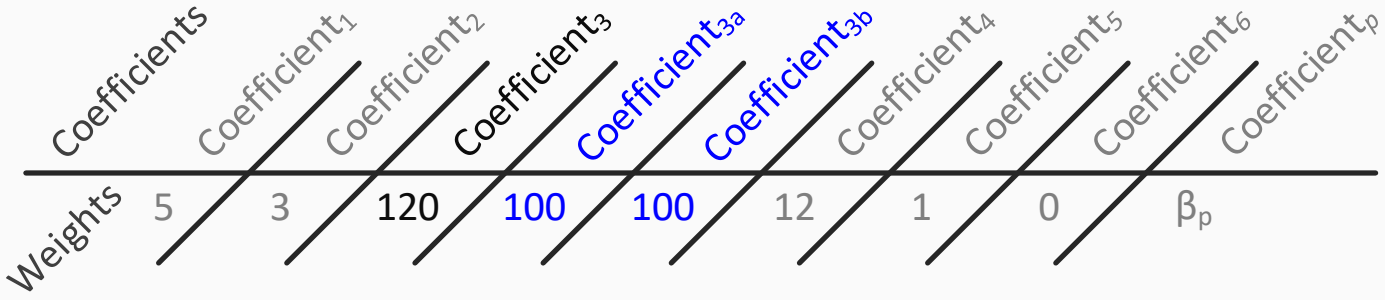
The following hypothetical illustrates **Colinearity** and Feature Selection:



Given the above algorithm parameters $\text{Coefficient}_1, \ldots, \text{Coefficient}_p$ and either **Regularization Term**:

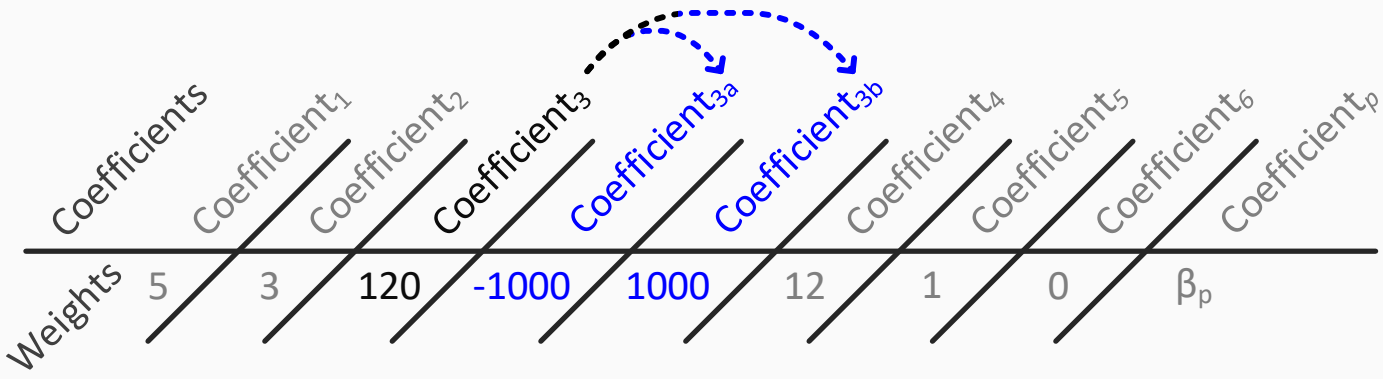
$$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2 \text{ (referred to as } \ell_2)$$

$$\text{Regularization}(f) = |\beta_1^2| + |\beta_2^2| + |\beta_3^2| + \cdots + |\beta_p^2| = \|\beta\|_1 \text{ (referred to as } \ell_1)$$



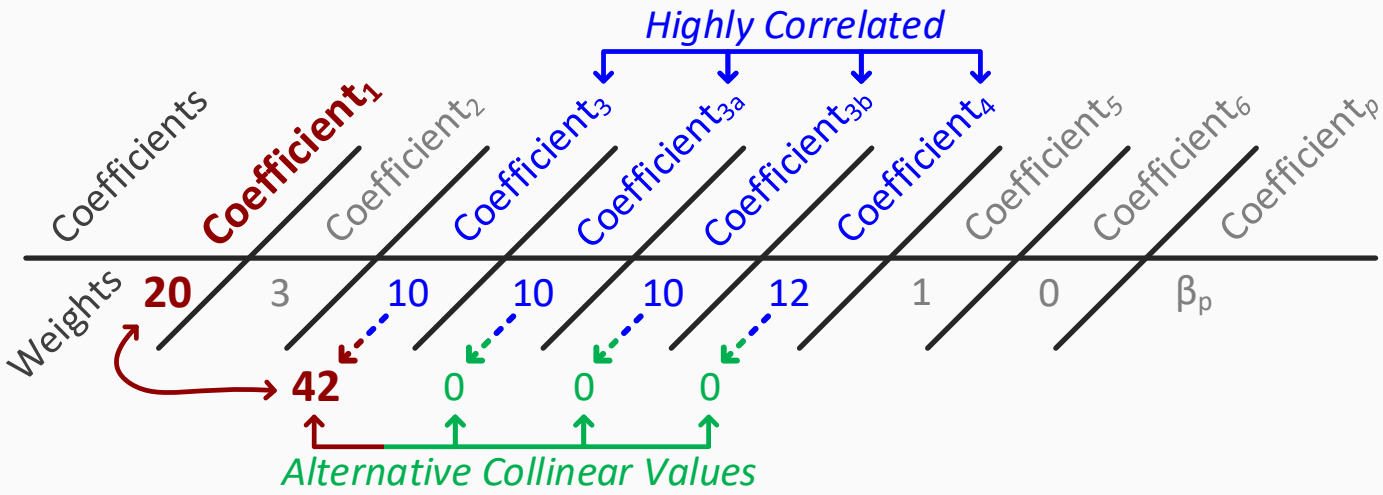
Assuming $\textbf{Coefficient}_3$ is determined to be valuable, a data scientist might derive multiple features (feature engineering) from the original as seen above Coefficient_{3a} and Coefficient_{3b}. The goal is to

achieve a higher rate of predictive power. However, the derived features are **highly correlated** amongst themselves and might lack independent predictive power on the model when evaluated.



Assuming Coefficient_{3a} and Coefficient_{3b} are assigned weights of -1000 and 1000 respectively. It might appear that the engineered features are highly valuable to the model. However this is a **learning fallacy** considering the two features effectively net to 0 and cancel each other out.

Again, **Coefficients** are **NOT** related to **feature importance** in a predictive model.

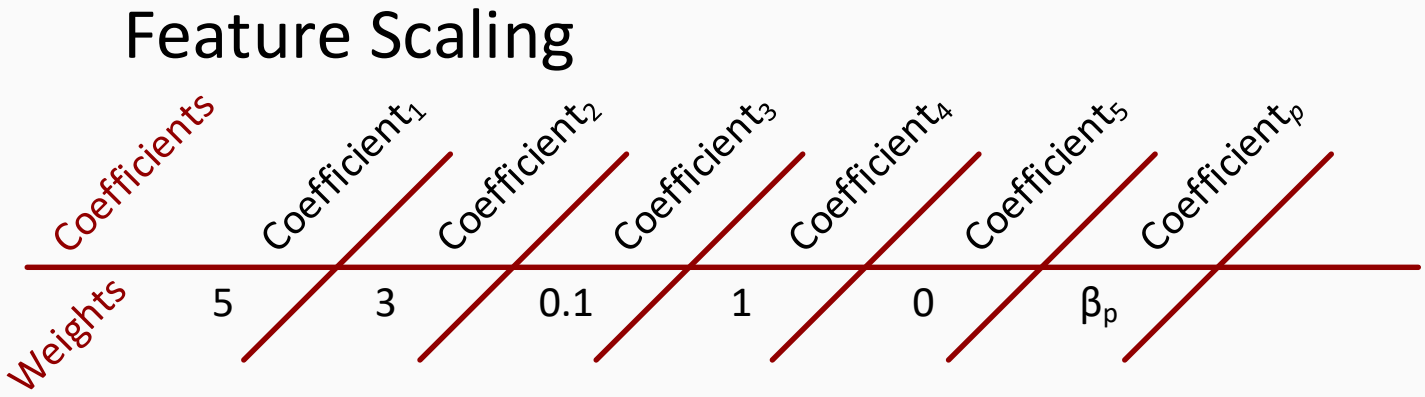


Assuming the above **Coefficient Weights**, it appears that the **four blue features** are **highly correlated**. Therefore, the most valuable feature with the given weights appears to be $\mathbf{Coefficient_1}$ with a given weight of **20**. However, because the **four blue features** are **highly correlated** with each other, they could have equally been weighted with the given **Alternative Values** instead. In the latter scenario, $\mathbf{Coefficient_3}$ appears to hold the highest value in predicting new labels from a dataset.

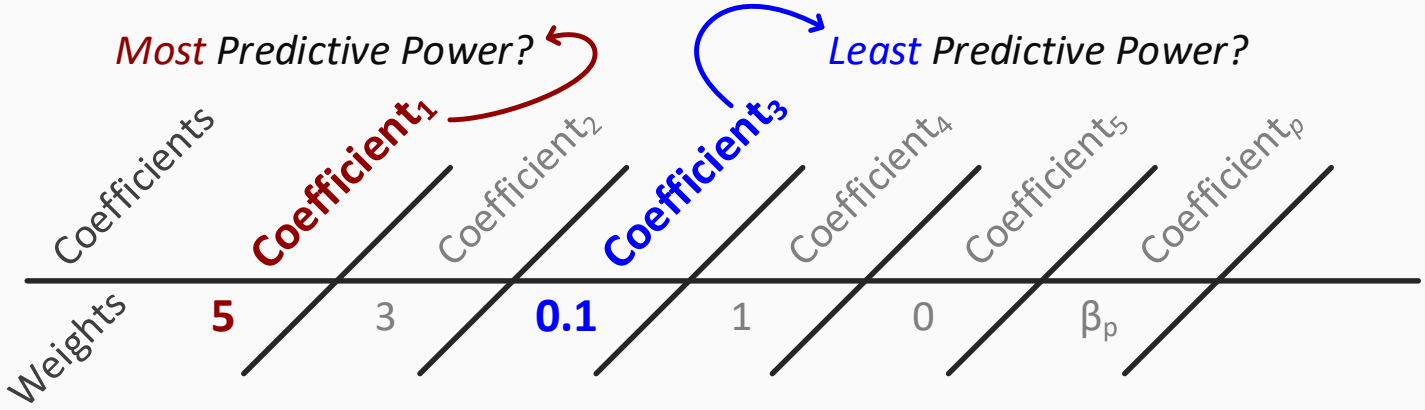
feature scaling

Alternatively, another method for **Feature Selection** is the process of **Feature Scaling**.

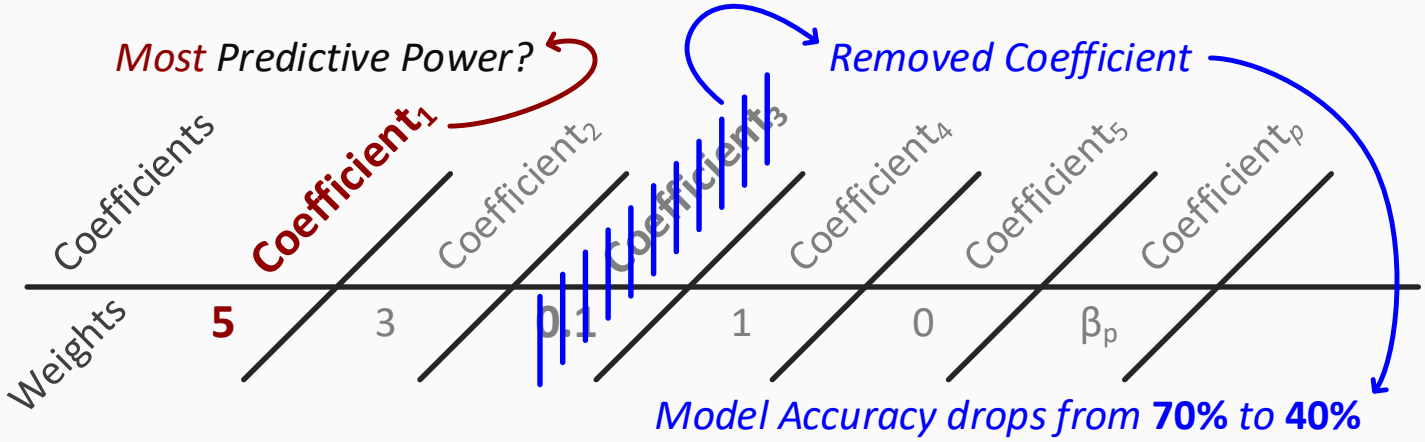
The following hypothetical illustrates **Feature Scaling**:

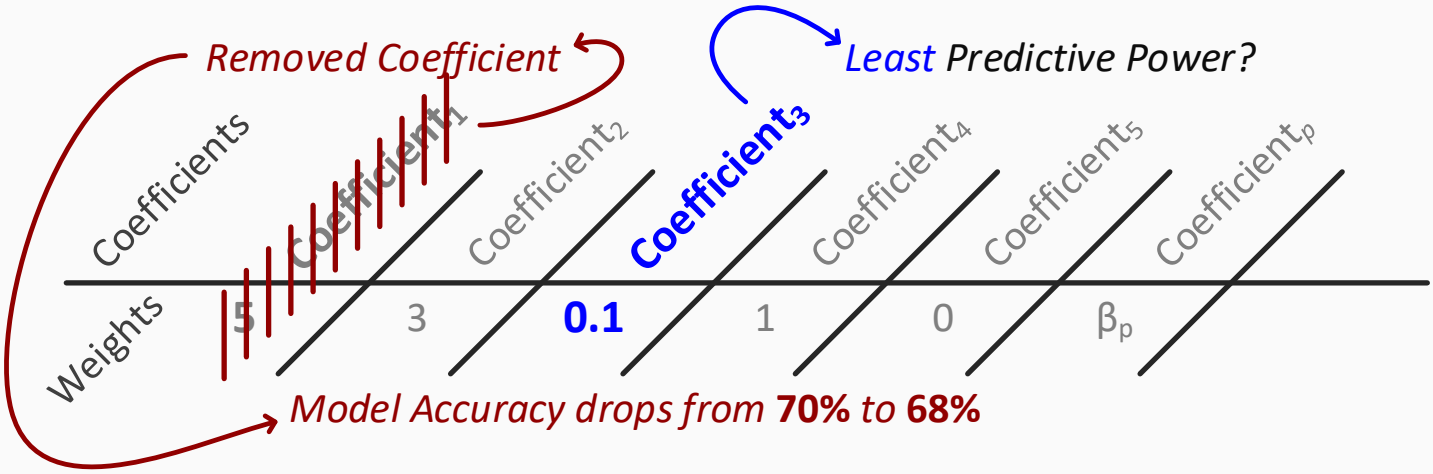


Assuming the above weights assigned to a given machine learning algorithm,



It can be noted that $\mathbf{Coefficient_1}$ appears with have the **most** predictive power with $\mathbf{Coefficient_3}$ having the **least** predicative power. With the latter intuition, it is expected to remove $\mathbf{Coefficient_3}$. However the decision to drop the low-valued parameter is a **learning fallacy**. $\mathbf{Coefficient_3}$ could potentially be a highly valuable parameter to model performance illustrated as follows:





Furthering the illustration above, intentionally (or unintentionally) removing the **most** valuable $\mathbf{Coefficient_1}$ would expect a high penalty on the model's performance. Conversely, the result could be less severe, proving the **appearance** of $\mathbf{Coefficient_1}$ having high importance is a **learning fallacy**.

The above outcome could be explained by $\mathbf{Coefficient_1}$ being measures on a **Range of 0 to 1**, contributing $\leq \mathbf{5\%}$ of the model's score. Additionally, $\mathbf{Coefficient_3}$ could be responsible for contributing $\leq \mathbf{14.5\%}$ to the model's score and measured on a **range of 0 to 145**.

Poor scaling equally distorts the interpretation of coefficients and has a negative effect on regularization. **Regularization** applies pressure on the larger coefficients to become smaller, meaning the features with the smallest values will be reduced; the latter is **not the intent of the data scientist**.

For example, when attempting to **regularize** all coefficients equally:

Given the application of $\boldsymbol{\ell_2}$ **Regularization** and two equally important features that are measured on **different scales**, the value of one coefficient is considerably larger than the other. The values of the first feature are small (0 or 1), therefore the coefficient is large. Conversely, the values of the second feature are small (up to 145), therefore the coefficient is small. **Regularization** is applied by reducing the coefficient of each feature by 20% as seen in the illustration. The impact on the aggregate **Regularization Term** is highly disparate depending upon the chosen coefficient to reduce. The example illustrates the tendency for the **Regularization Term** to prefer reduction of the first coefficient opposed to the second, regardless of the features having **<u>equal importance</u>**. Ensuring the features are on a relative scale will prevent regularization from biasing features for arbitrary reasons.

$$\begin{aligned}
\text{Regularization}(f) &= \beta_1^2 + \beta_2^2 \\
&5^2 + 0.10^2 \quad \rightarrow \quad 25.01 \\
\text{Reduce Feature}_1 \text{ by } 20\% \rightarrow\ &4^2 + 0.10^2 \quad \rightarrow \quad 16.01 \\
\text{Reduce Feature}_2 \text{ by } 20\% \rightarrow\ &5^2 + 0.08^2 \quad \rightarrow \quad 25.0064
\end{aligned}$$

Improperly Scaled features will also have a negative impact towards **distance metrics**. Concretely, **regularization** can have an impact on the **loss function** itself. If a loss function depends on Euclidean distances between $\boldsymbol{x_i}'\mathbf{s}$, the same problem arises as with the above illustration; the algorithm may inappropriately determine certain dimensions more valuable over others.

An appropriate solution would be to scale each feature to be measured **between -1 and 1** (or between 0 and 1). The latter is achieved through the following expression, however, there are various methods to effectively scale features in a dataset. It is important to take these measures as a part of **preprocessing** data prior to constructing a functioning model.

$$x^{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

sweeping parameters » tuning model hyperparameters

The **Tune Model Hyperparameters** module in Microsoft AzureML is used to automatically search for optimal combinations of machine learning model hyperparameters.

The hyperparameter space can be searched in several ways; on a regular or randomized grid of the hyperparameter space. Alternatively, the hyperparameter space can be randomly sampled. In either case, the search of hyperparameter space can be computationally intensive, since the model must re-compute and evaluate the model. Fortunately, most commonly used machine learning models are not particularly sensitive to the choice of hyperparameters.

cross validation

The machine learning culture revolves around effective evaluation of predictive models. ***Out-of-sample*** testing in the form of **Cross Validation** is heavily relied on when training and testing a model.

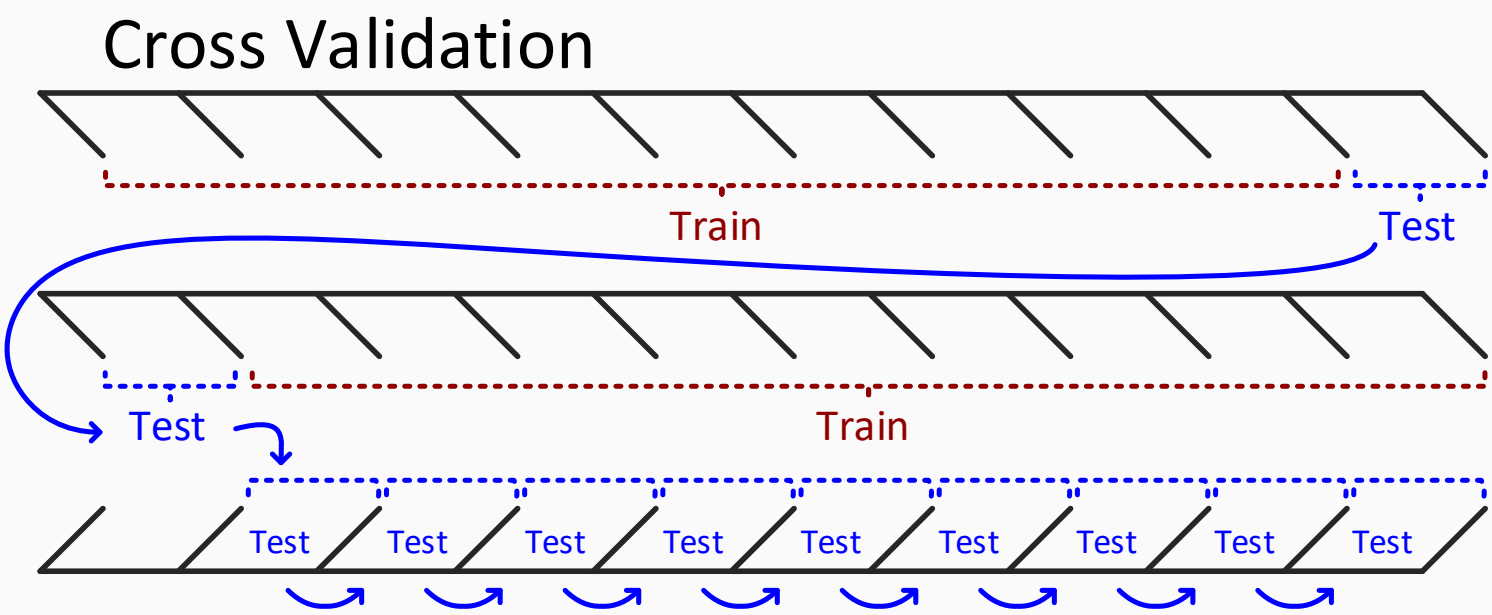
Cross Validation is the most widely used method of machine learning algorithm evaluation on data.

Performing **Cross Validation** requires the follows:

- A dataset
- An algorithm
- An evaluation metric for the quality of result
- Often metrics are squared error between the predicted and actual values

Cross Validation is thus executed as follows:

- Dataset is partitioned into approximately 10 equally-sized "**folds**".
- The algorithm is trained on 9 folds, with the evaluation metric computed on the 10^{th} fold.
- The training sequence reiterates on the proceeding fold until all folds are evaluated.
  - The process is ultimately repeated 10 times, with each fold in turn as the test fold.
- The mean and standard deviation of the evaluation metric are reported over the 10 folds.



The algorithm with the highest measured performance (**average out-of-sample performance across the 10 test folds**) is then applied to the model. Additionally, significance tests can be computed against the performance across the folds.

nested cross validation

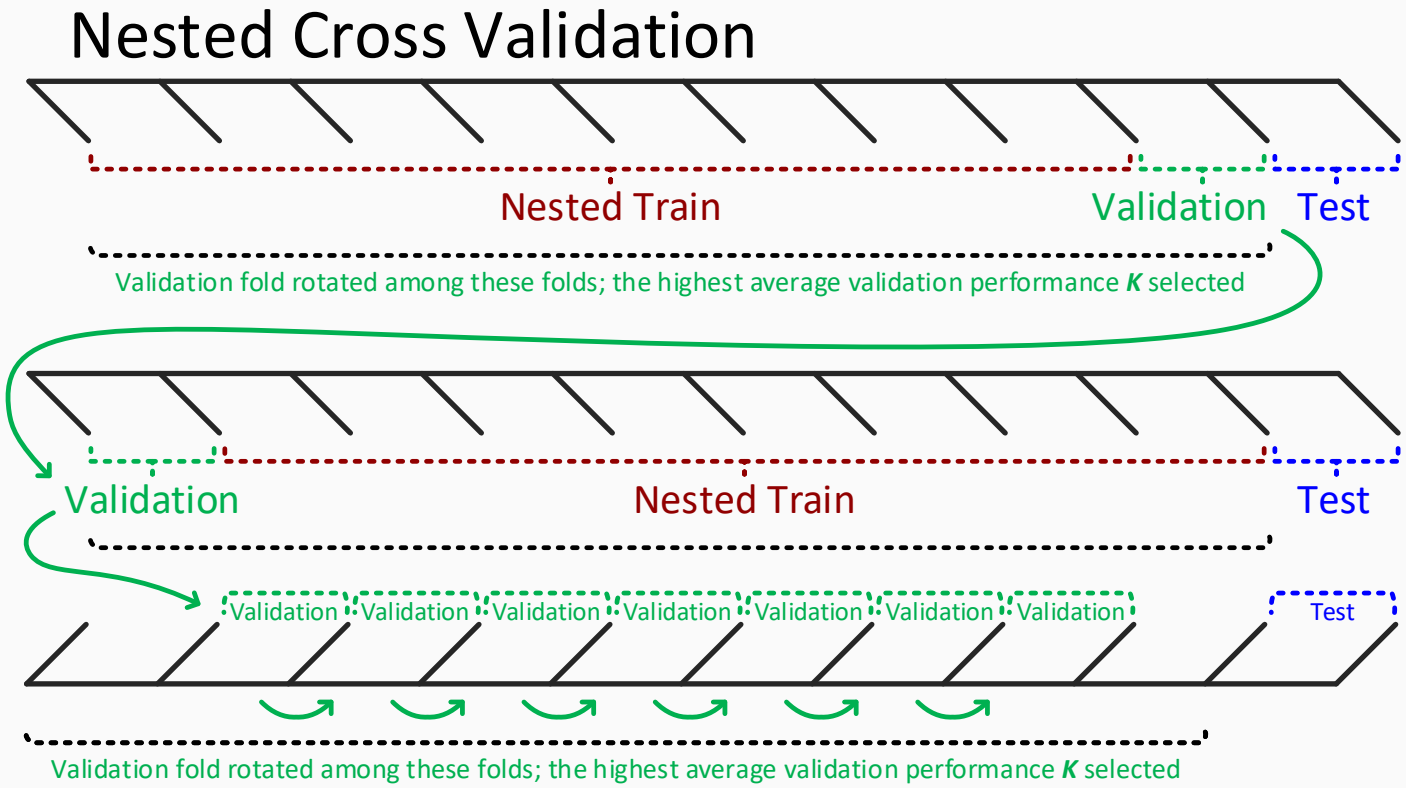
Nested Cross Validation is specifically the most used method of tuning parameters in an algorithm.

Performing **Nested Cross Validation** requires the follows:

- A dataset
- An algorithm
- An evaluation metric for the quality of result (**additionally a parameter for tuning**)
- For illustration, the parameter will is K and will be assigned values of 1, 10 ,100, 1000, or 10000.

Cross Validation is thus executed as follows:

- Dataset is partitioned into approximately 10 equally-sized "**folds**", reserving one for **test**.
- Additionally, a fold will be reserved for **validation**.
- For $K = 1, 10, 100, 1000, 10000$, the algorithm is trained on the 8 remaining folds, with the evaluation metric computed on the **validation** fold; **5 measurements** are computed.
- The training sequence reiterates on the proceeding fold until all folds are evaluated.
  - The process is ultimately repeated 9 times, rotating which training fold is for **validation**.
  - The resulting computations are 9*5 metrics (**9 folds x 5 K's**).
- K that minimizes that average training error over the 9 folds in chosen and used to evaluate on the test dataset.
- The process is repeated 10 times from the second step, using each fold in turn as the test fold.
- The mean and standard deviation of the evaluation metric are reported over the 10 test folds.



The algorithm with the highest measured performance (**average out-of-sample performance across the 10 test folds**) where **Nested Cross Validation** was applied is then assigned to the model. Additionally, significance tests can be computed against the performance across the folds.

Nested Cross Validation can be highly computationally expensive

(10 test sets*10 validation sets*number of parameter settings being considered).

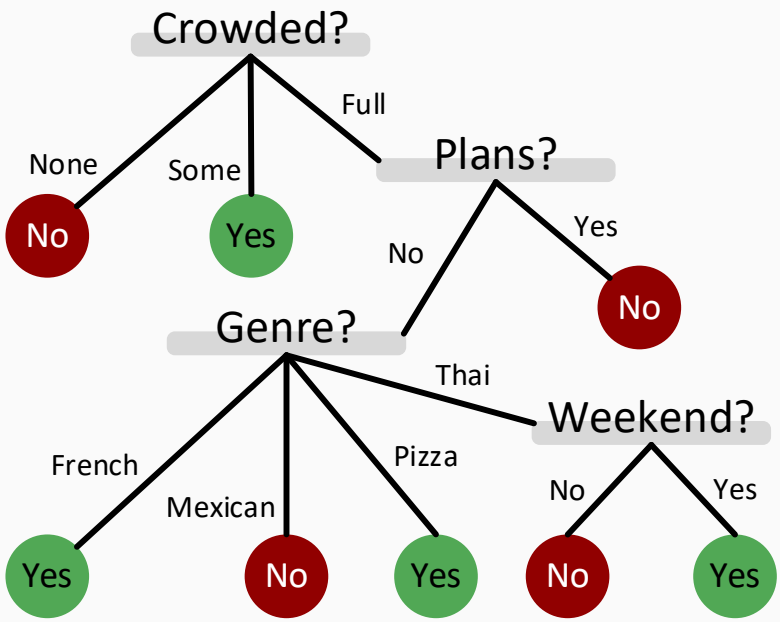
module4 · tree and ensemble methods

decision tress

Example: Will the customer wait for a table at a restaurant?

- OthOptions: Other options, True if there are restaurants nearby.
- Weekend: This is true if it is Friday, Saturday or Sunday.
- Area: Does it have a bar or other nice waiting area to wait in?
- Plans: Does the customer have plans just after dinner?
- Price: This is either $, $$, $$$, or $$$$
- Precip: Is it raining or snowing?
- Genre: French, Mexican, Thai, or Pizza
- Wait: Wait time estimate: 0-5 min, 6-15 min, 16-30 min, or 30+
- Crowded: Whether there are other customers (no, some, or full)

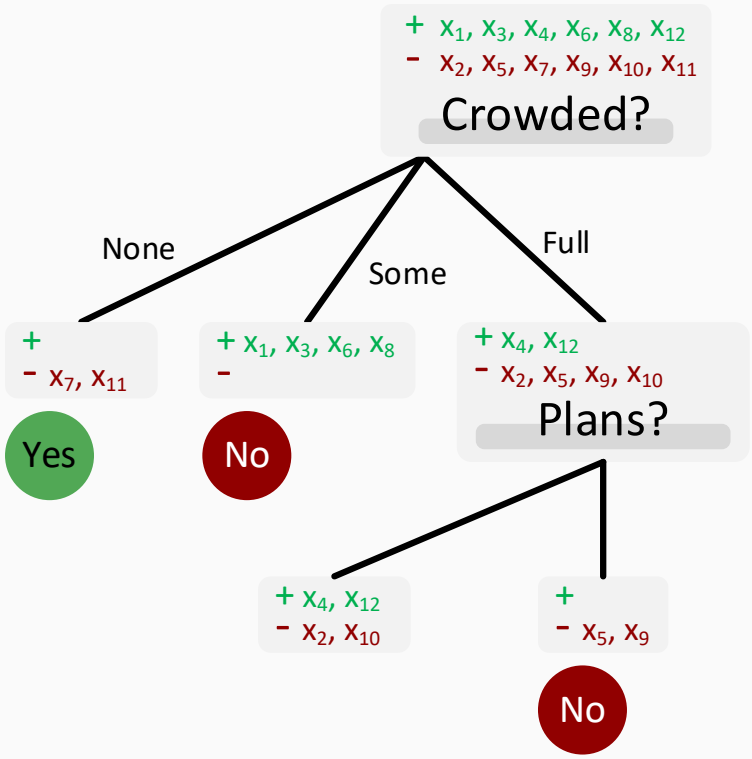
	OthOptions	Weekend	Area	Plans	Price	Precip	Genre	Wait	Crowded	Stay?
x_1	Yes	No	No	Yes	$$$	No	French	0-5	some	Yes
x_2	Yes	No	No	Yes	$	No	Thai	16-30	full	No
x_3	No	No	Yes	No	$	No	Pizza	0-5	some	Yes
x_4	Yes	Yes	No	Yes	$	No	Thai	6-15	full	Yes
x_5	Yes	Yes	No	No	$$$	No	French	30+	full	No
x_6	No	No	Yes	Yes	$$	Yes	Mexican	0-5	some	Yes
x_7	No	No	Yes	No	$	Yes	Pizza	0-5	none	No
x_8	No	No	No	Yes	$$	Yes	Thai	0-5	some	Yes
x_9	No	Yes	Yes	No	$	Yes	Pizza	30+	full	No
x_{10}	Yes	Yes	Yes	Yes	$$$	No	Mexican	6-15	full	No
x_{11}	No	No	No	No	$	No	Thai	0-5	none	No
x_{12}	Yes	Yes	Yes	Yes	$	No	Pizza	16-30	full	Yes

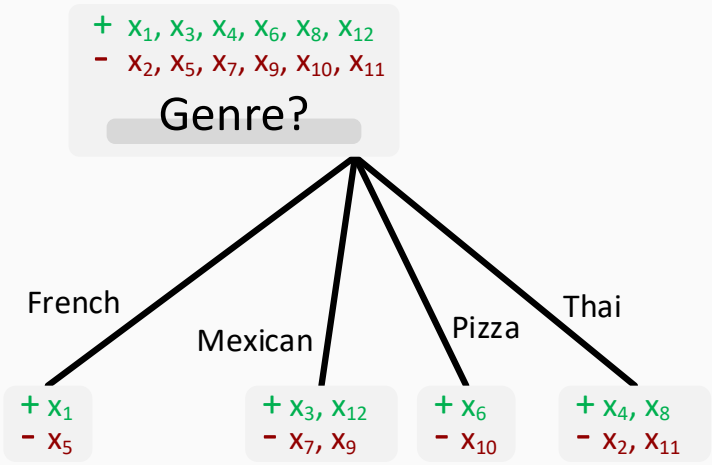


As illustrated in the example, **decision trees** apply a hierarchy of **nested logic** with associated **probabilities** of occurrence to determine labels. The rule-based logic applied to the table above produces an answer to the predicted value on the **Stay?** column. The **decision tree** itself is illustrated in the graphic to the **left** showing the progression from the first, to the last question determining the ultimate binary response to the problem set. **Decision Trees** are generally interpretable, straight forward, and applicable to an array of varying logical problems applied to a dataset.

constructing decision tress

In determining which features to split within a **Decision Tree**, it is ideal to spilt upon the feature that provides the most information about that questions; **"will the customer have to wait?"**:





The example illustrates properties such as:

- If there is **no crowd**, no one (**2**) waits for a table
- If there is **some crowd**, everyone (**4**) waits
- If the restaurant is **fully crowded**, **2** leave; **4** wait
- If the menu is **French**, half leave and half wait
- If the menu is **Mexican**, half leave and half wait
- If the menu is **Pizza**, half leave and half wait
- If the menu is **Thai**, half leave and half wait

Examining the **Decision Tree** above displays clearly that splitting on the **genre** feature offers no predictive power; the outcomes of each property are equally likely given the training data.

what is information?

Information from observing the occurrence of an event = the number of bits needed to encode the probability of the event.

If an event has probability $p \to -\log_2(p)$ **bits** are needed.

A coin flip encodes 1 **bit** of information:

$$p = \frac{1}{2}, \text{and thus } -\log_2\left(\frac{1}{2}\right) = 1 \textbf{ bit}$$

For an event with probability 1, 0 bits are needed:

$$p = 1, \text{and thus } -\log_2(1) = 0 \textbf{ bits}$$

Given many events, with probabilities $[\boldsymbol{p_1}, \ldots, \boldsymbol{p_m}]$, the probabilities' mean information is as follows:

$$E_{p\sim[p_1,\ldots,p_m]}I(p) = \sum_{j=1}^{j} p_j I(p_j) = -\sum_{j=1}^{j} p_j \log_2 p_j = \boldsymbol{H([p_1, \ldots, p_m])} \to \textbf{Entropy}$$

Any probabilities that are ~ 0 will have a corresponding **entropy** ~ 0, synonymous to applying the definition of an average. The information is simply weighted by their probabilities of occurrence.

Given the definition above, if only 2 events (**binary**) with probabilities p and $1-p$ exist:

$$H([p, 1-p]) = -p\log_2(p) - (1-p)\log_2(1-p)$$

If the probabilities are $\frac{1}{2}$ and $\frac{1}{2}$:

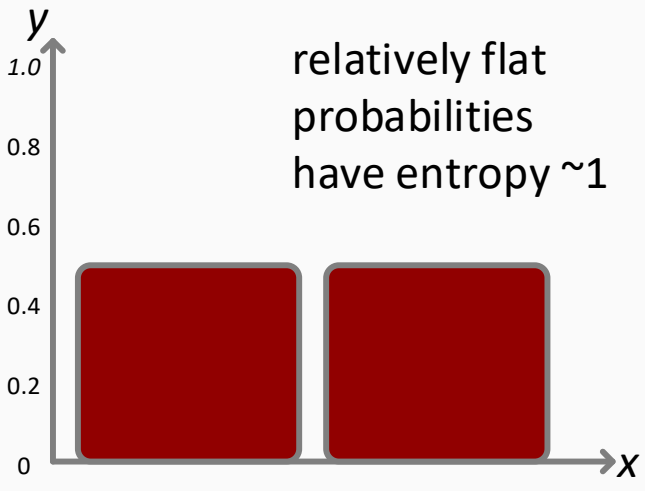
$$H\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right) = -2\frac{1}{2}\log_2\left(\frac{1}{2}\right) = 1$$

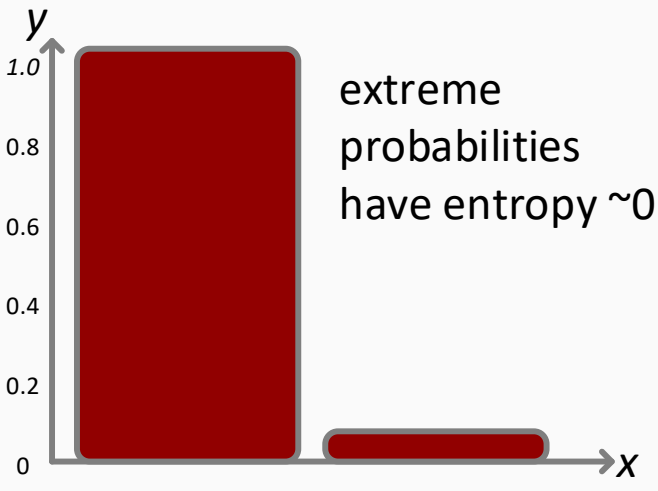
If the probabilities are 0.99 and 0.01:

$$H([0.99, 0.01]) = 0.08 \text{ bits}$$

Entropy Intuition

In relation to probability, higher **entropy** correlated with flat probabilities (uniform) that are spread out. **Rare events** conversely result in lower **entropy**:





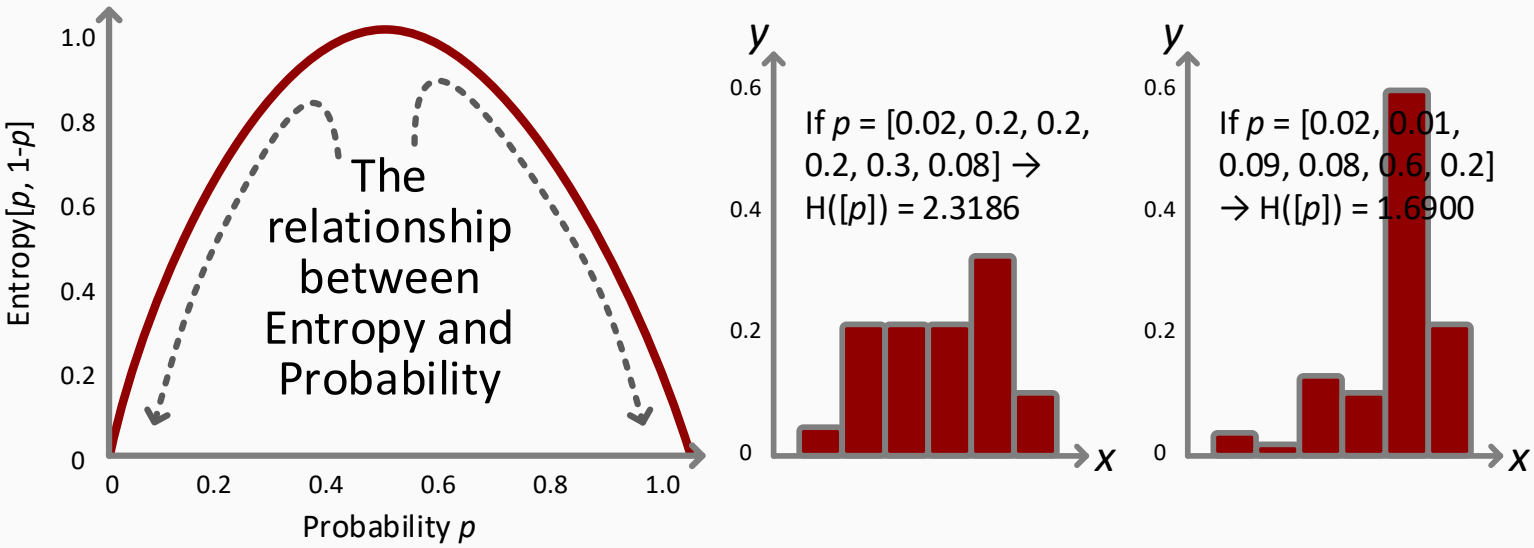
If the probabilities are $\frac{1}{2}$ and $\frac{1}{2}$:

$$H\left(\left[\frac{1}{2}, \frac{1}{2}\right]\right) = -2\frac{1}{2}\log_2\left(\frac{1}{2}\right) = 1$$

If the probabilities are 0.99 and 0.01:

$$H([0.99, 0.01]) = 0.08 \text{ bits}$$

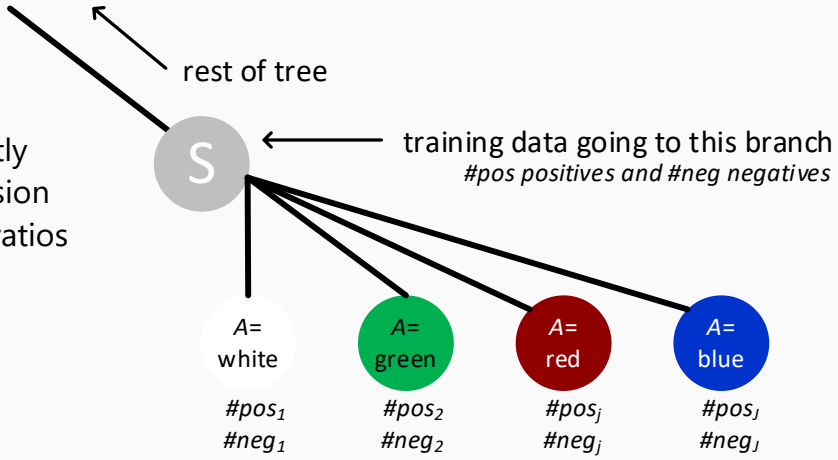
For example:



splitting criteria for decision trees ⚠ information gain

Given the above definition of **information**, with **entropy** being the average of **information**, the definition of **information gain** is as follows:

Support Vector Machines begin by addressing the question of which decision boundary to select when each point is correctly classified. The right illustration splits the decision on a **color** variable. The positive to negative ratios can be computed for each branch:



The training probabilities for branch$_j$ are:

$$\left[\frac{\#\text{pos}_j}{\#\text{pos}_j + \#\text{neg}_j}, \frac{\#\text{neg}_j}{\#\text{pos}_j + \#\text{neg}_j}\right]$$

... noting that $\frac{[0.08, 0.99] \rightarrow \text{is good}}{[0.50, 0.50] \rightarrow \text{is bad}}$

Extreme probabilities at either end of the spectrum (near **0** and **1)** have strong predictive power; the predicted labels will essentially be known for new observations in the dataset. Flat probabilities are poor predictors.

Thus, **Information Gain** for a particular subset of Data $\boldsymbol{S}$, in a particular branch of variable $\boldsymbol{A}$ is the original **entropy** before the branch minus the entropy after the branch:

$$\begin{aligned}\text{Gain}(\boldsymbol{S},\boldsymbol{A}) &= \text{expected reduction in entropy due to branching on attribute } \boldsymbol{A}\\ &= \text{original entropy} - \text{entropy after branching}\\ &= H\left(\left[\frac{\#\text{pos}}{\#\text{pos}+\#\text{neg}},\frac{\#\text{neg}}{\#\text{pos}+\#\text{neg}}\right]\right) - \sum_{j=1}^{j}\frac{\#\text{pos}_j+\#\text{neg}_j}{\#\text{pos}+\#\text{neg}}H\left[\frac{\#pos_j}{\#pos_j+\#neg_j},\frac{\#neg_j}{\#pos_j+\#neg_j}\right]\end{aligned}$$

For example, the variable $\boldsymbol{A}$ is split with half of the data being positive and half of the data being negative (**Entropy** – 1). The entropy after splitting will be much lower, proving information gain to be substantial:

$$\text{Gain}(\boldsymbol{S},\boldsymbol{A}) = H\left(\left[\frac{1}{2},\frac{1}{2}\right]\right) - \left[\frac{2}{12}H([0,1]) + \frac{4}{12}H([1,0]) + \frac{6}{12}H\left(\left[\frac{2}{6},\frac{4}{6}\right]\right)\right] \approx 0.541 \textbf{ bits}$$

A less successful example can be seen with variable **B**:

$$\text{Gain}(\boldsymbol{S},\boldsymbol{B}) = 1 - \left[\frac{2}{12}H\left(\left[\frac{1}{2},\frac{1}{2}\right]\right) + \frac{2}{12}H\left(\left[\frac{1}{2},\frac{1}{2}\right]\right) + \frac{4}{12}H\left(\left[\frac{2}{4},\frac{2}{4}\right]\right) + \frac{4}{12}H\left(\left[\frac{2}{4},\frac{2}{4}\right]\right)\right] \approx 0 \textbf{ bits}$$

Standard Procedure for Building Decision Trees

- Begin at the top of the Tree
- Grow the Tree by "**splitting**" the features (**C4.5** Algorithm displayed above, **CART** alternatively uses Gini Index) one by one. Determine where to split by examining how "impure" the node is.
- Assign leaf nodes the majority vote in the leaf
- Return back through the Tree to **prune** leaves and reduce overfitting of the model to the data.
  - The elimination of subtrees that do not have a high cost-to-value for the model
  - CART uses the "minimal cost complexity" pruning method

Decision Trees are generally used for **classification** but do exist in ways for **regression**.

Decision Tree Advantages

- Handles nonlinearities due to being logical models
- Decision Trees are generally interpretable (especially CART)
- Greedy models starting from the top and not regressing backwards to reevaluate, thus fast
- Can easily handle imbalanced data by reweighting the points

Decision Trees Disadvantages

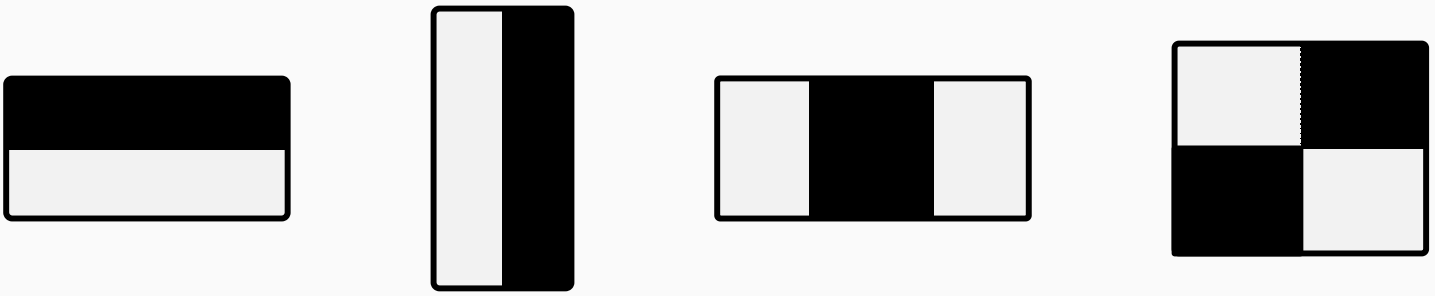
- Not very interpretable without trading off in accuracy (complexity increases model accuracy)
- Greedy models will be fast but will trade off in accuracy
- Decision Trees are a heuristic algorithm, thus lacking computational elegance considering that they are not optimized for any specific purpose or application
- Tend to perform poorly on imbalanced data, even after adjusting the parameters.

boosting

The motivation behind **Boosting** originated from a question asked by Michael Kearns: "**Can a 'weak' learning algorithm** (slightly better than random guessing) **be turned into a 'strong' learning algorithm** (error rate arbitrarily close to 0)**?**"

Schapire and French worked on the above problem, examine ways to make the algorithm create numerous classifiers and determine the best method of combination. The struggled existed with having a **single dataset** being able to create different classifiers. The answer was to **reweight the data in many ways** and feed them into the algorithm; creating a **weak classifier** for each (**reweighted**) dataset. Ultimately, a **weighted average** would be computed of the weak classifiers.

One of the most noted application of **combining weak classifiers** was that of Viola and Jones in their image detection experiments. The concept of weak classifiers for facial recognition is as follows:



The above are representative of matrix filters that scan images to determine contrasts in the pixel densities between the light and darker segments of each filter. The pixel densities are subtracted to determine features in an image. The filters are applied in all different shapes, sizes and orientations. The intuition behind these weak classifiers is covered in detail in the Applied Machine Learning Documentation. For this purpose, the motivation behind combined many weak classifiers to create a strong learning algorithm to classify new labels in a dataset is the intent of the illustration.

The following application of **weak classifiers** above will be in the context of the **AdaBoost** algorithm.

adaboost

AdaBoost functions by reweighting the dataset in many iterations to compute different results out of the weak learning algorithm each time around.

AdaBoost Pseudocode

Assign observation i the weight of $d_{1i} = \frac{1}{n}$ (equal weights)

For t=1:T

Train weak learning algorithm using data weighted by $d_{1i} = \frac{1}{n}$ ...

...producing weak classifier h_t.

Choose coefficient α_t.

Update weights:

$y_i h_t(x_i) = 1$ if correct, -1 if incorrect

$$d_{t+1,i} = \frac{d_{t,i}\exp(-\alpha_t(y_i h_t(x_i))}{Z_t}$$ with Z_t serving as a normalization factor

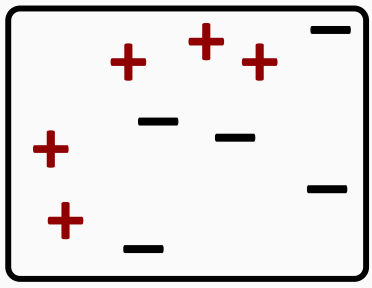
End

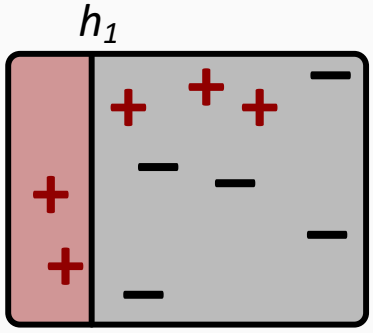
Output the final classifier: $H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x_i)\right)$

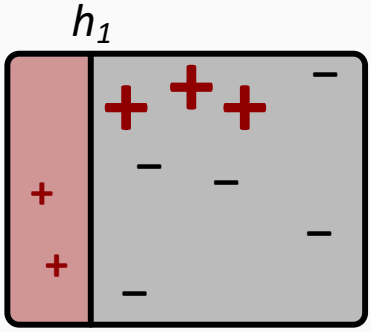
Princeton Boosting Survey

The following illustrative example assumes any decision boundaries are **horizontal** or **vertical** lines:

Each point begins the algorithm with equal weights:

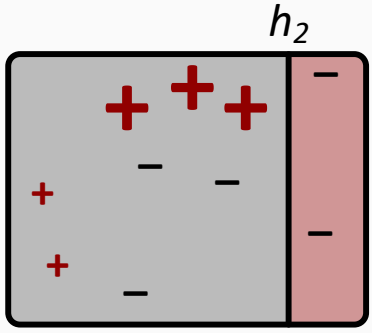


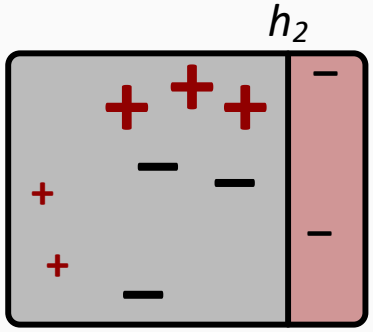




The **first Boosting** iteration (**weak classifier**) is run and the coefficient weight is chosen → $\boldsymbol{\alpha_1 = 0.42}$

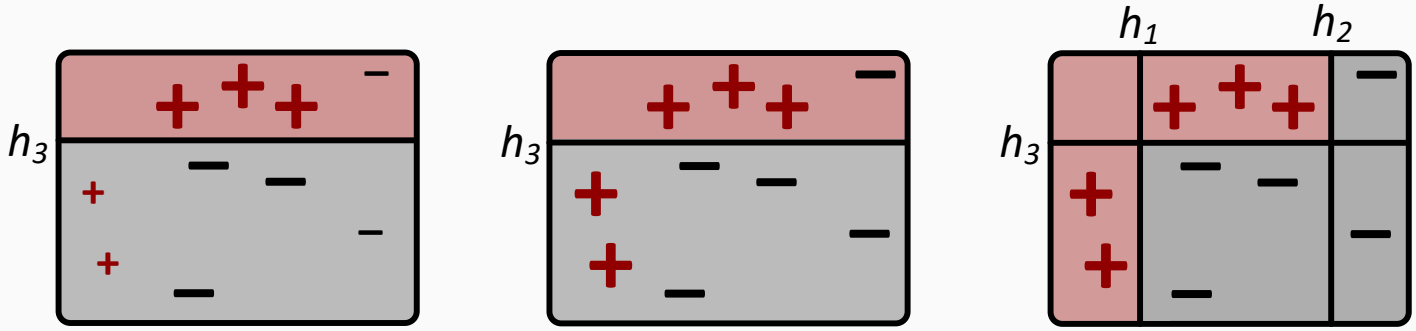
The weights of the **misclassified points** are **increased**; **decrease correctly classified points** weights.





The **second Boosting** iteration is run and the coefficient weight is chosen → $\boldsymbol{\alpha_1 = 0.66}$

The weights of the **misclassified points** are **increased**; **decrease correctly classified points** weights.



The **third Boosting** iteration is run and the coefficient weight is chosen → $\boldsymbol{\alpha_1 = 0.66}$

The weights of the **misclassified points** are **increased**; **decrease correctly classified points** weights.

Finally, output the **weighted average classifier** $H(x) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(x_i))$:

$$H = \text{sign}\left(0.42 \square + 0.66 \square + 0.93 \square\right)$$

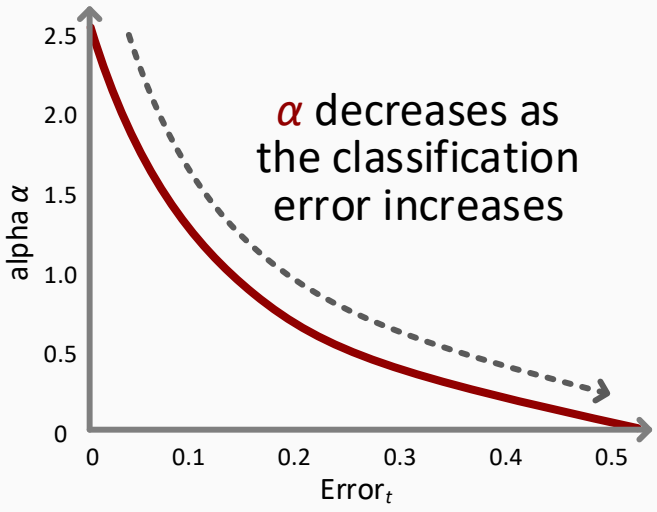
AdaBoost Coeffisicients Update

The alpha α examines how well the classifier performs at each iteration; the **classification error**:

$$\text{Error}_t = \sum_{i\,:\,h_t(x_i) \neq y_i} d_t = \text{sum of weights of misclassified points}$$

Intuitively, if the error rate is large, alpha α will conversely be assigned a small value to minimize the classifiers impact on the model. Plotting alpha α against the error illustrates the latter intuition:

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1 - Error_t}{Error_t}\right)$$

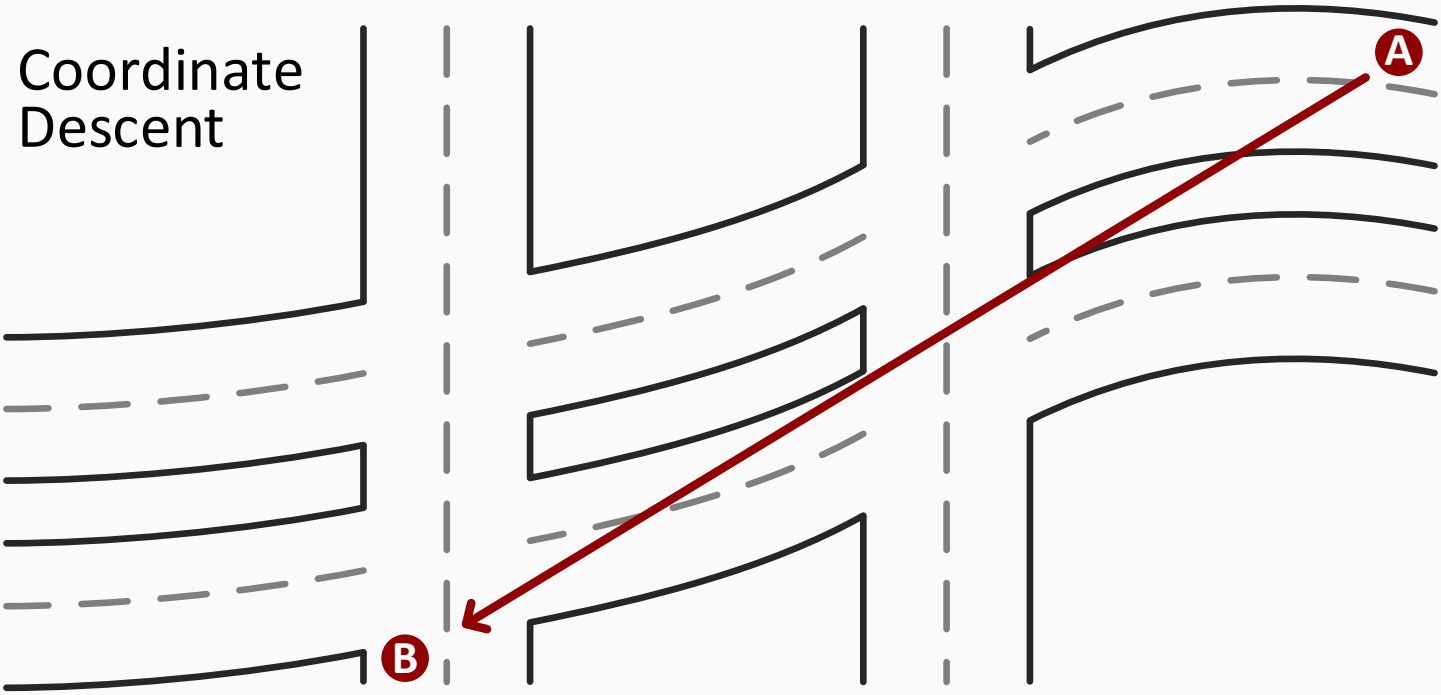


coordinate descent

Coordinate Descent has synonymous logic to **gradient descent** focusing on **exponential loss** applied through combining weak classifiers into a powerful learning algorithm (**boosting**).

In the context of **AdaBoost**, the above illustrations are oriented towards general **binary classifiers**.

Understanding the origins of the formula and coordinate weighting functions can be illustrated using **coordinate descent** geometrically as shown below:



Assuming a starting point at **A**, the natural intuition of arriving at point **B** is to travel in a straight line. This, however, is not possible in the illustration above considering the available paths restrict the directions and distances point **A** can travel at a time. **Coordinate Descent**, in turn, has the objective of minimizing the function to determine the best path to the final point **B** from starting point **A**.

Coordinate Descent Pseudocode

For $t = 1{:}T$
    Choose direction h_t
    Choose the distance to travel in direction α_t
End

Output the final position: $f(x) = \sum_{t=1}^{T} \alpha_t h_t(x_i)$

Therefore, each **Weak Classifier** (**decision boundary**) $h_1, h_2, \dots, h_t$ can be thought of as the **direction traveled** above.

In abstract terms, h_t represents the chosen directions and α_t represents how far traveled in the determined direction for each iteration of the **AdaBoost** algorithm.

AdaBoost Pseudocode

Assign observation i the weight of $d_{1i} = \frac{1}{n}$ (equal weights)

For $t = 1:T$

Train weak learning algorithm using data weighted by $d_{1i} = \frac{1}{n}$ …

…producing weak classifier h_t.

Choose coefficient α_t.

Update weights:

$y_i h_t(x_i) = 1$ if correct, -1 if incorrect

$$d_{t+1,i} = \frac{d_{t,i}\exp(-\alpha_t(y_i h_t(x_i))}{Z_t}$$ with Z_t serving as a normalization factor

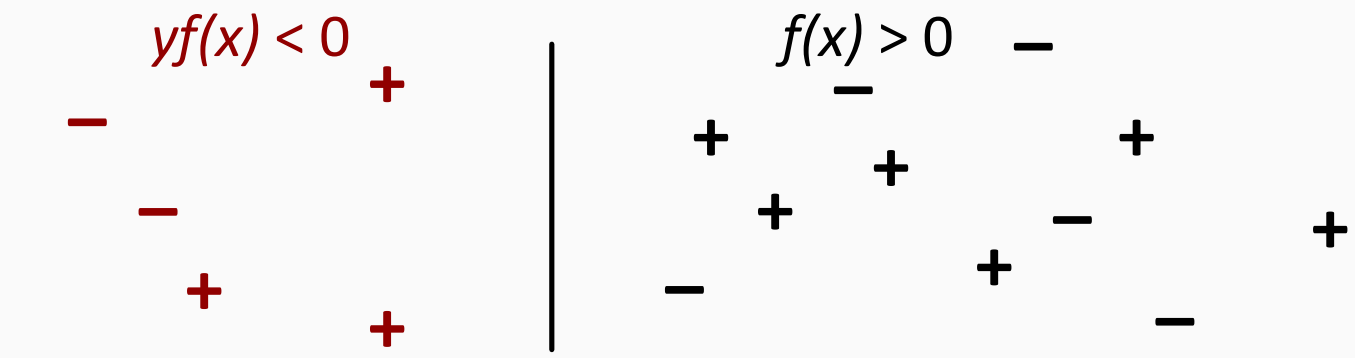
End

Output the final classifier: $H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x_i)\right)$

The above representation of **AdaBoost** can be explained logically in steps as follows:

- The **Weak Learning Algorithm** is essentially the most promising chosen direction of travel
- The assignment of **coefficient α_t** tells the algorithm know how far to travel in that direction
- The **weight update** essentially aids in selecting the next direction of travel in the next iteration
- The **final classifier** determines the total amount of distance travels in all directions

The final component to **Coordinate Descent** is the formula used to determine the hill to travel upon:

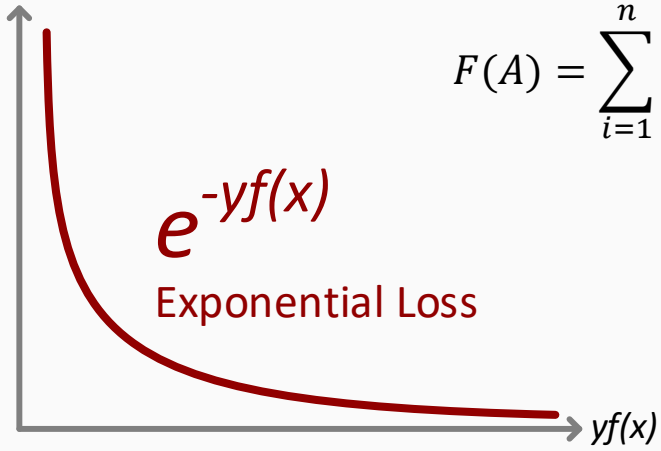


Returning to the geometrics representation of **classification** above, the left frame represents **misclassified points** and the right frame represents **correctly classified** points.

The objective is to choose **loss functions** that heavily penalize any and all **misclassified points** seen the left frame. This is achieved through the **Exponential Loss Function** as illustrated in the example to the right:

AdaBoost Objective Function:

$$F(A) = \sum_{i=1}^{n} e^{-yf(x)}$$



AdaBoost Objective Function:

$$F(A) = \sum_{i=1}^{n} e^{-yf(x)}$$

The direction of travel chosen will represent the **steepest descent**:

$$j_t = \underset{j}{\operatorname{argmax}} \left| -\left(\frac{dF(A + \alpha l_j)}{d\alpha} \right) \right|_{a=0} = \underset{j}{\operatorname{argmax}} \sum_{\substack{i \text{ that are misclassified} \\ y_i h_j(x_i) = -1}} d_{t,i}$$

The direction is that of the **lowest weighted error**.

In the circumstance that the optimal direction is not chosen, the algorithm as flexible and will correct.

The above is the concept of applying coefficient a_t as a weighting function to **AdaBoost**:

$$0 = \left| \frac{dF(A + \alpha l_j)}{d\alpha} \right|_{\alpha_t} \rightarrow \alpha_t = \frac{1}{2} \ln \left(\frac{1 - Error_t}{Error_t} \right)$$

Lastly, the weights will be updated to ensure the steepest descents are computed in each reiteration.

The algorithm essentially will continue travelling along the descent through reiterations until the benefit is **substantially decayed**. Therefore, a **1 dimensional optimization problem** is being solved; where the minimized function along that dimension is chosen.

The Difference Between Boosted Decision Trees and Decision Forests

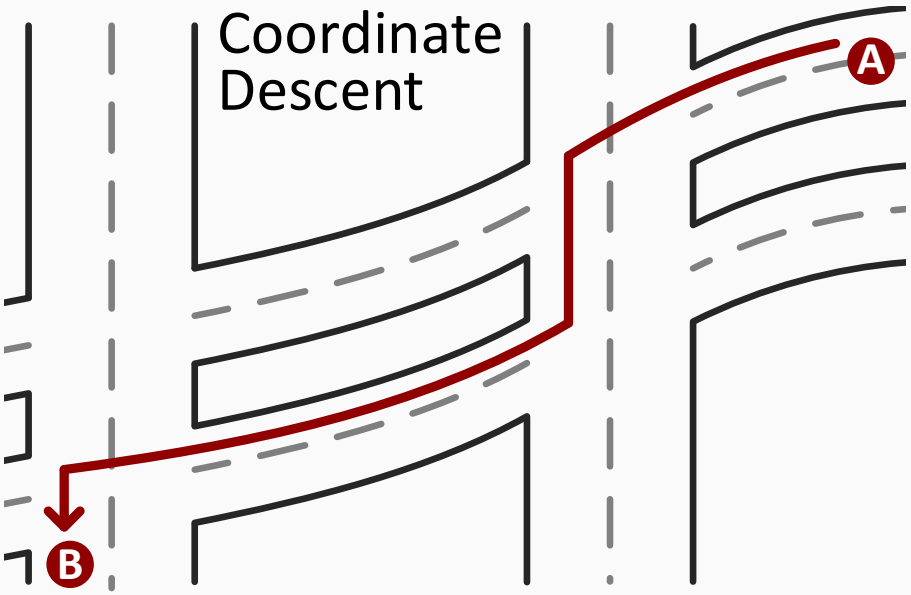
Decision Forests

- Computing many trees from different subsets of data and features
- Average the results (**bagging**)

Boosted Decision Trees

- Reweight the data to generate different trees
- The combination of multiple weights minimizes the training error (**coordinate descent performed on exponential lost**)

The differences coverage as the result of each is essentially a multitude of **overfitted, heuristic decision trees** that are combined in some determined way. The philosophy behind **decision forests** is to **average** everything to **reduce variance**. The approach behind **boosting** tries to **minimize bias** and make the model **more accurate**, but ends up **reducing variance** anyway because the trees that it generates tend to be diverse from continuous reweighting.



decision forests

Decision Forests are another type of ensemble method where weak classifiers are combined to build a stronger algorithm.

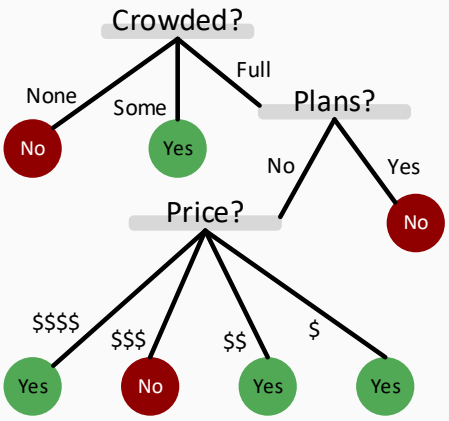
- **Decision Forests** are a powerful prediction tool that are highly complex
- They are another type of **black-box** algorithm where interpretability is low
- Similar logic to **Boosted Decision Trees**; averaging many uncorrelated, yet accurate, models reduces overall variance

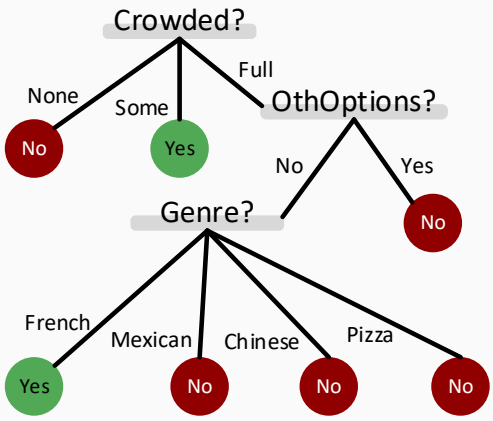
Returning to the illustrative restaurant scenario from the Decision Tree example:

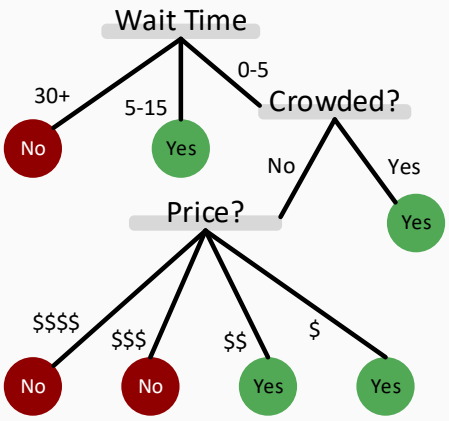
Example: Will the customer wait for a table at a restaurant?

- OthOptions: Other options, True if there are restaurants nearby.
- Weekend: This is true if it is Friday, Saturday or Sunday.
- Area: Does it have a bar or other nice waiting area to wait in?
- Plans: Does the customer have plans just after dinner?
- Price: This is either $, $$, $$$, or $$$$
- Precip: Is it raining or snowing?
- Genre: French, Mexican, Thai, or Pizza
- Wait: Wait time estimate: 0-5 min, 6-15 min, 16-30 min, or 30+
- Crowded: Whether there are other customers (no, some, or full)

	OthOptions	Weekend	Area	Plans	Price	Precip	Genre	Wait	Crowded	Stay?
x_1	Yes	No	No	Yes	$$$	No	French	0-5	some	Yes
x_2	Yes	No	No	Yes	$	No	Thai	16-30	full	No
x_3	No	No	Yes	No	$	No	Pizza	0-5	some	Yes
x_4	Yes	Yes	No	Yes	$	No	Thai	6-15	full	Yes
x_5	Yes	Yes	No	No	$$$	No	French	30+	full	No
x_6	No	No	Yes	Yes	$$	Yes	Mexican	0-5	some	Yes
x_7	No	No	Yes	No	$	Yes	Pizza	0-5	none	No
x_8	No	No	No	Yes	$$	Yes	Thai	0-5	some	Yes
x_9	No	Yes	Yes	No	$	Yes	Pizza	30+	full	No
x_{10}	Yes	Yes	Yes	Yes	$$$	No	Mexican	6-15	full	No
x_{11}	No	No	No	No	$	No	Thai	0-5	none	No
x_{12}	Yes	Yes	Yes	Yes	$	No	Pizza	16-30	full	Yes







A new observation is applied to the models above:

Genre: Mexican
Price: $$
Crowded: Full
Wait Time: 5-15
Plans: No
OthOptions: No

Applying the observation to the models above, the labels respectively assigned are **yes**, **yes**, **no**.

Therefore, the combination of the models apply the majority vote as the predicted label: **yes**

More precisely, the method utilizing **bootstrapping** of samples. A **bootstrap sample** of size n: A sample of n points drawn with replacement at random from the training data. Each point is taken one at a time and then replaced prior to selecting another. Thus some points will be repeated as expected.

Decision Forest Pseudocode

For $t = 1{:}T$

Draw a **Bootstrap Sample** of size n from the training dataset

Build a **Decision Tree** $\mathbf{tree}_t$ using the splitting and stopping procedure:

Choose m features at random (out of p)

Evaluate the splitting criteria on all features m

Split on the most valuable feature

Repeat until the node has $< n_{\min}$ observations, then cease splitting

Output all trees (noting that only no pruning takes place, only splitting)

To predict on a new observation x, the majority vote of the trees is applied to x

Decision Forests Compared to Decision Trees

- **Decision Trees** use all sample data; **Decision Forests** use bootstrapped resamples to compute many decision trees.
- **Decision Trees** are allowed to use all of the features for consideration of each split; **Decision Forests** only consider m randomly chosen features for each split.
- **Decision Forests** do not utilize pruning, making trees fit more tightly and reduce bias.
- **Decision Forests** apply majority vote to make predictions; **Decision Forests** use a single tree.

Measuring Variable Importance with Decision Forests

In measuring the importance of feature $\boldsymbol{j}$:

The data **not used** to construct $\mathbf{tree}_t$ is taken: referred to here as "out-of-bag" $\mathbf{OOB}_t$

The $\mathbf{error}_t$ will be computed on the $\mathbf{OOB}_t$ data using the model $\mathbf{tree}_t$

The $\mathbf{OOB}_t$ data is then taken and randomly permutated on the $\boldsymbol{j^{th}}$ feature values:

$$\mathbf{OOB_t} \rightarrow \begin{bmatrix} x_{11} & \boldsymbol{x_{12}} & x_{13} \\ x_{21} & \boldsymbol{x_{22}} & x_{23} \\ x_{31} & \boldsymbol{x_{32}} & x_{33} \end{bmatrix} \rightarrow \text{randomly permuted} \rightarrow \begin{bmatrix} x_{11} & \boldsymbol{x_{32}} & x_{13} \\ x_{21} & \boldsymbol{x_{12}} & x_{23} \\ x_{31} & \boldsymbol{x_{22}} & x_{33} \end{bmatrix} \rightarrow \mathbf{OOB}_{t,\mathbf{permuted}}$$

The logic behind a random permutation is explained hypothetically: if the feature $\boldsymbol{j}$ was unimportant originally, randomly arranging the feature values would have little impact.

Then, the $\mathbf{error}_{t,\mathbf{permuted}}$ will be computing using model $\mathbf{tree}_t$ on data $\mathbf{OOB}_{t,\mathbf{permuted}}$

The "**raw importance**" of variable $\boldsymbol{j}$ is thus the average over trees of the difference:

$$\frac{1}{T}\sum_{\text{trees } t}\left(\text{error}_t - \text{error}_{t,\text{permutated}}\right)$$

Decision Forests for Regression

For $\boldsymbol{t = 1{:}T}$

Draw a **Bootstrap Sample** of size $\boldsymbol{n}$ from the training dataset

Build a **Decision Tree** $\mathbf{tree}_t$ using the splitting and stopping procedure:

Choose $\boldsymbol{m}$ features at random (out of $\boldsymbol{p}$)

Evaluate the splitting criteria on all features $\boldsymbol{m}$

Split on the most valuable feature

Repeat until the node has $< \boldsymbol{n}_{\mathbf{min}}$ observations, then cease splitting

Output all trees (noting that only no pruning takes place, only splitting)

To predict on a new observation $\boldsymbol{x}$, the ~~majority~~ **average** vote of the trees is applied to $\boldsymbol{x}$

Summary of Decision Forests

Advantages

- Complex and powerful prediction tool that is highly nonlinear

Disadvantages

- Black-box algorithm that is difficult to interpret
- Prone to overfitting unless tuned carefully (not intuitive with R package)
- Slow running algorithm with high computational cost

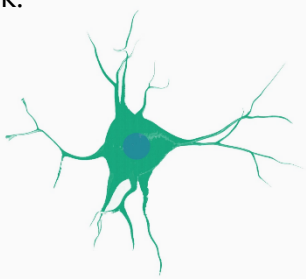
Decision Forests are essentially a collection of overfitted heuristic models that are averaged over their individual majority votes to predict new labels in a dataset.

module5 · optimization-based methods

neural networks

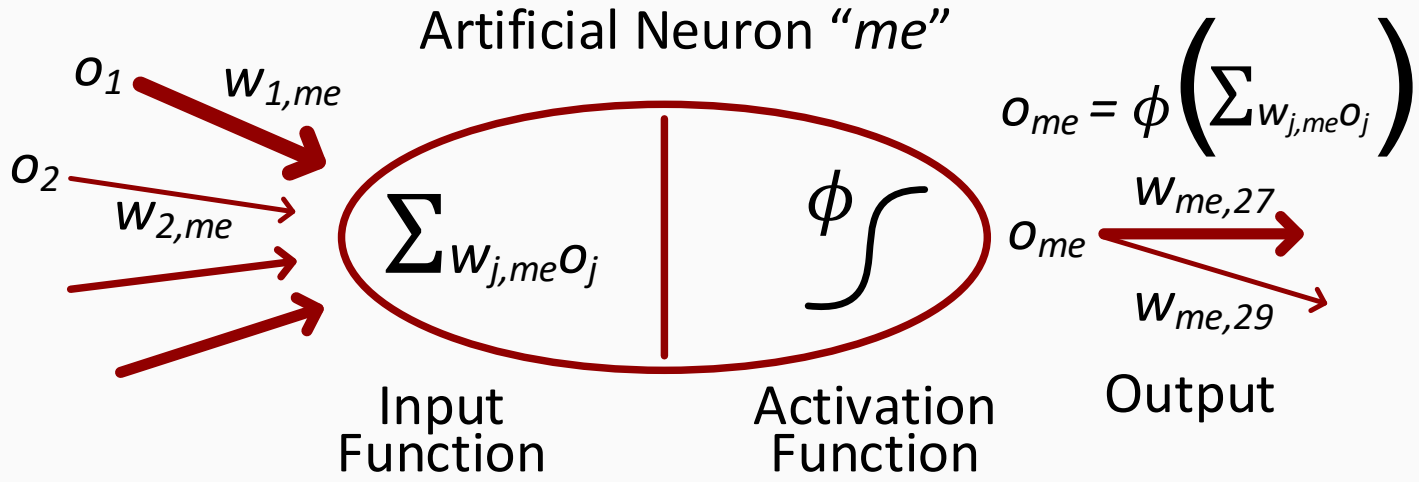
Neural Networks are particularly powerful for computer vision problems. **Artificial Neural Networks** are not the same as **real Neural Networks**:

- There are $\mathbf{10^{11}}$ neurons in a human brain with $\mathbf{10^{14}}$ synapses (connections).
- **Signals** are the electrical potential spikes that travel through the network.
- Thus, the order of magnitude is much larger in a **real Neural Network**.
- **Artificial Neural Networks**, in turn, compute like calculators.



McCulloh-Pitts "Neuron"

Illustrating an **Artificial Neuron** named "***me***":



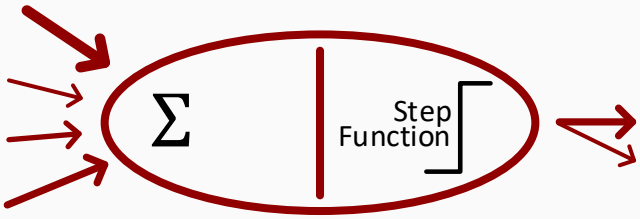
All other outputs $\boldsymbol{o_j}$ of the predecessor neurons are fed into "***me***" and **weighted** by their **connectivity**. The thickness of the predecessor outputs is representative of the magnitude (**weight**) of connectivity. The neuron "***me***" consequentially takes the sum of all the outputs from previous neurons as weighted by their relative connectivity: $\sum_j \boldsymbol{w_{j,me} o_j}$.

The summation feeds into an **Activation Function** which is typically between 0 and 1; appearing as a **threshold function** $\boldsymbol{\phi}\left(\sum_j \boldsymbol{w_{j,me} o_j}\right)$ → the value of the activation function = the **output** of the neuron $\boldsymbol{o_{me}}$. Once computed, the output of neuron "***me***" feeds into all the successive neurons afterwards, also weighted by the connectivity strength.

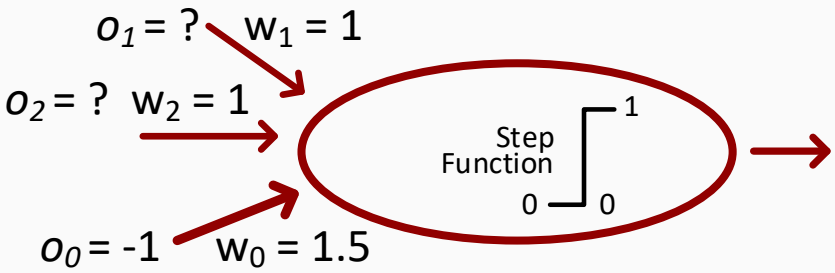
Assuming that the activation function $\boldsymbol{\phi}\left(\sum_j \boldsymbol{w_{j,me} o_j}\right)$ is simply a **step function** for the threshold:

Therefore, enough electrical input from prior neurons will make the sum $\sum_j w_{j,me} o_j$ large;

Ultimately firing when the **activation function** is triggered and returning the higher value (**1**) as opposed to the lower value (**0**).



Applying Weights to Compute an Artificial Neuron's Output



o_1	o_2	output
0	0	0
1	0	0
0	1	0
1	1	1

$$o_1 = 0, o_2 = 0 \rightarrow \phi\left(\sum_j w_{j,me} o_j\right) \rightarrow (0 * 1) + (0 * 1) + (-1 * 1.5) = -1.5 < 0 \rightarrow \sout{\text{activate}} \rightarrow o_{me} = 0$$

$$o_1 = 1, o_2 = 0 \rightarrow \phi\left(\sum_j w_{j,me} o_j\right) \rightarrow (1 * 0) + (0 * 1) + (-1 * 1.5) = -0.5 < 0 \rightarrow \sout{\text{activate}} \rightarrow o_{me} = 0$$

$$o_1 = 0, o_2 = 1 \rightarrow \phi\left(\sum_j w_{j,me} o_j\right) \rightarrow (0 * 1) + (1 * 1) + (-1 * 1.5) = -0.5 < 0 \rightarrow \sout{\text{activate}} \rightarrow o_{me} = 0$$

$$o_1 = 1, o_2 = 1 \rightarrow \phi\left(\sum_j w_{j,me} o_j\right) \rightarrow (1 * 1) + (1 * 1) + (-1 * 1.5) = 0.5 > 0 \rightarrow \text{activate} \rightarrow o_{me} = 1$$

The above illustration outputs the logic **where if one of the outputs from prior neurons = 0, then the output of the "*me*" neuron =0; if both outputs = 1, then the output of "*me*" = 1**.

In other words, the above Artificial Neuron computes the **logical table AND function**.

Additionally, Artificial Neural Networks can equally compute the logical table **OR** and **NOT** functions.

backpropagation

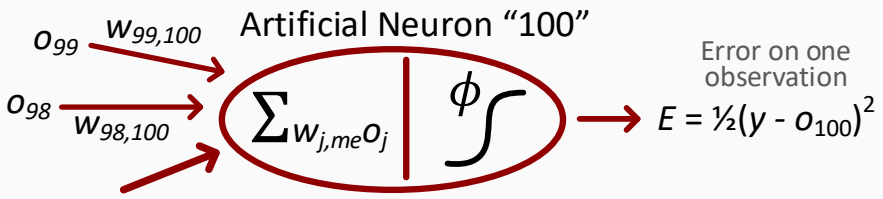
The **Activation Function** used in **Artificial Neural Networks** introduced above represents a threshold; the function is applied in a "smooth" nature due to being differentiable:

$$\phi\left(\sum_j w_{jme}\alpha_j\right) = \frac{\mathbf{1}}{(\mathbf{1}+\boldsymbol{e}^{-x})} \rightarrow \textbf{Sigmoid Function used in Logistic Regression}$$

Single Layer Neural Network

An **Artificial Neural Network** learns by examining a particular input in **supervised learning** and comparing the predicted output; the Neuron's **error** on one observation: $E = \frac{1}{2}(y - o_{100})^2$

In a human brain, the synapses strengthen and weaken in order to learn; the same applies above. Therefore, the **error** will be minimized with respect to the weighted outputs from prior neurons.



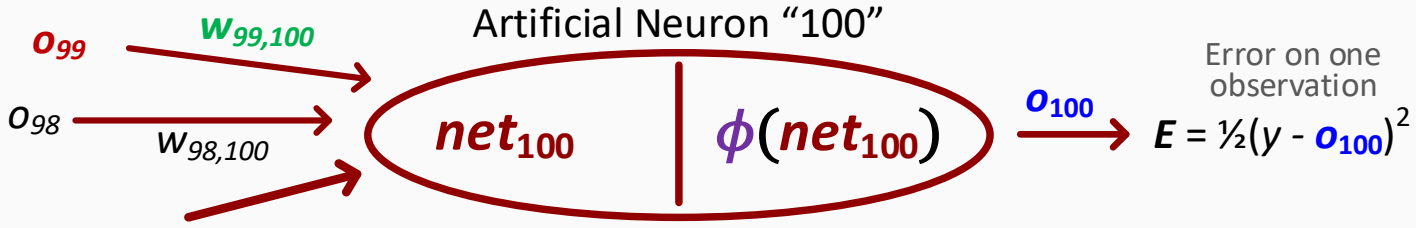
Backpropagation Algorithm

Minimization of the Error in an Artificial Neuron is the objective of **Backpropagation**.

- An algorithm that trains the weights of a neural network
- Require information to propagate backwards through the network, then forwards, then backwards, etc.; reiterating as such
- Backwards Propagation = the chain rule from calculus

Backpropagation through a Single Layer Neural Network

$\boldsymbol{net_{100}}$ is simply the weighted sum from prior neurons weighted by their respective connectivity. The objective is to adjust the weight $\boldsymbol{w_{99,100}}$ in order to minimize training error on the Neuron. The derivative of the error will be taken with respect to the weight $\boldsymbol{w_{99,100}}$ in order to take steps along the gradient and reduce the error:



Below represents the derivative of error $\boldsymbol{E}$ using the **chain rule** from calculus:

$$\frac{dE}{dw_{99,100}} = \frac{dE}{do_{100}}\frac{do_{100}}{dnet_{100}}\frac{dnet_{100}}{dw_{99,100}}$$

$$\phi(z) = \frac{1}{1+e^z}$$

$$\phi'(z) = \frac{d\phi(z)}{dz} = \phi(z)\big(1-\phi(z)\big)$$

The above represents the derivative of $\boldsymbol{E}$ with respect to the output $\boldsymbol{o_{100}}$, the derivative of the output $\boldsymbol{o_{100}}$ with respect to the net $\boldsymbol{net_{100}}$, and the derivative of the net $\boldsymbol{net_{100}}$ with respect to the weight $\boldsymbol{w_{99,100}}$.

In order to calculate at least one of the terms from the chain rule, a factor must be applied as the activation $\boldsymbol{\phi}$. The derivative of phi $\boldsymbol{\phi}$ shown above is phi $\boldsymbol{\phi(z)}$ itself times 1 minus phi $\boldsymbol{\left(1-\phi(z)\right)}$.

Stepping through the Derivative of the Error with Respect to the Weight

$$\frac{dE}{d\boldsymbol{o_{100}}} = \frac{1}{2}2(y-\boldsymbol{o_{100}})(-1) = -(y-\boldsymbol{o_{100}})$$

$\boldsymbol{E}$ is a function of $\boldsymbol{o_{100}}$ and thus the derivative of $\boldsymbol{E}$ with respect to $\boldsymbol{o_{100}}$ is applied appropriately.

$$\frac{d\boldsymbol{o_{100}}}{d\boldsymbol{net_{100}}} = \frac{d\boldsymbol{\phi}(\boldsymbol{net_{100}})}{d\boldsymbol{net_{100}}} = \boldsymbol{\phi}'(\boldsymbol{net_{100}})\left(1-\boldsymbol{\phi}(\boldsymbol{net_{100}})\right) = \boldsymbol{o_{100}}(1-\boldsymbol{o_{100}})$$

The derivative of $\boldsymbol{o_{100}}$ with respect to $\boldsymbol{net_{100}}$is related through phi $\boldsymbol{\phi}$; thus requiring the derivative of phi $\boldsymbol{\phi}$ and $\boldsymbol{net_{100}}$represented through $\boldsymbol{o_{100}}$, appropriately, as illustrated above.

$$\frac{d\boldsymbol{net_{100}}}{d\boldsymbol{w_{99,100}}} = \frac{d\left(\boldsymbol{w_{99,100}o_{99}} + w_{98,100}o_{98} + w_{97,100}o_{97} + \cdots\right)}{d\boldsymbol{w_{99,100}}} = \boldsymbol{o_{99}}$$

$\boldsymbol{net_{100}}$ is simply the sum of the weighted outputs from prior Neurons, with only one containing the weight $\boldsymbol{w_{99,100}}$. Thus, the numerator represents the weighted sum of outputs with one occurrence of $\boldsymbol{w_{99,100}}$. The derivative of $\boldsymbol{net_{100}}$ with respect to $\boldsymbol{w_{99,100}}$ is simply the output of neuron "99" $\boldsymbol{o_{99}}$.

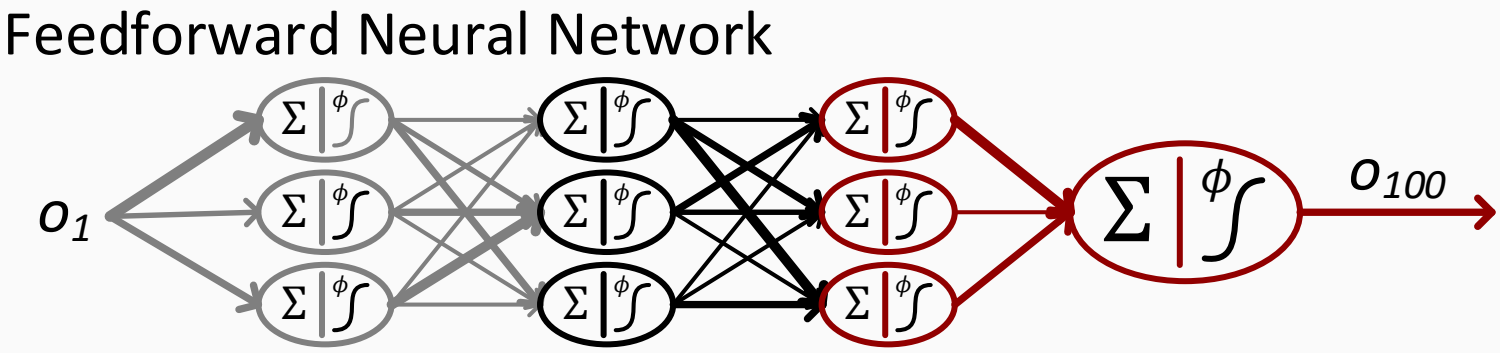
The result of the calculation above is as follows:

$$\frac{dE}{d\boldsymbol{w_{99,100}}} = \frac{dE}{d\boldsymbol{o_{100}}}\frac{d\boldsymbol{o_{100}}}{d\boldsymbol{net_{100}}}\frac{d\boldsymbol{net_{100}}}{d\boldsymbol{w_{99,100}}} = -(y-\boldsymbol{o_{100}})\boldsymbol{o_{100}}(1-\boldsymbol{o_{100}})\boldsymbol{o_{99}}$$

Adjusting the notation for application for the term that only depends on **node "100"**:

$$\frac{dE}{d\boldsymbol{o_{100}}}\frac{d\boldsymbol{o_{100}}}{d\boldsymbol{net_{100}}} = \boldsymbol{\delta_{100}} \rightarrow \frac{dE}{d\boldsymbol{w_{99,100}}} = \boldsymbol{\delta_{100}o_{99}}$$

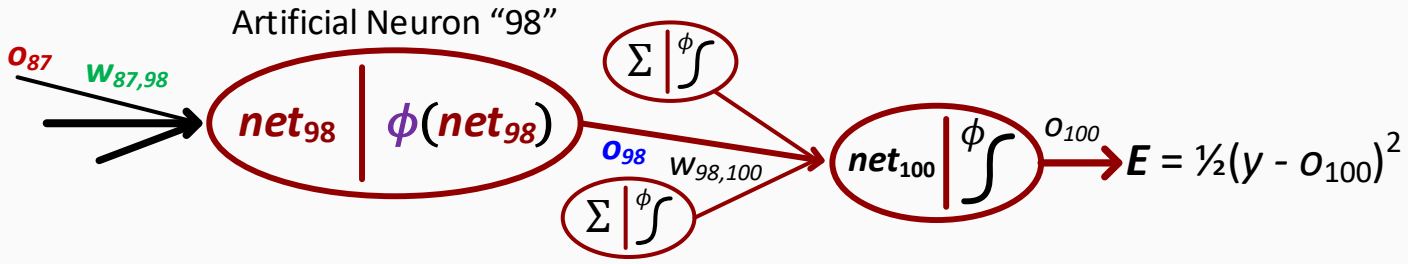
The above illustration represents the calculations for the final layer of an **Artificial Neural Network**:



Each layer is composed of series of Neurons that contribute weighted outputs to the final layer.

backpropagation through neural network hidden layer

The following illustrates calculations passing back through an **Artificial Neural Network** layer:



Below represents the derivative of error E using the **chain rule** from calculus:

$$\frac{dE}{dw_{87,98}} = \frac{dE}{do_{98}} \frac{do_{98}}{dnet_{98}} \frac{dnet_{98}}{dw_{87,98}}$$

$$\phi(z) = \frac{1}{1+e^z}$$

$$\phi'(z) = \frac{d\phi(z)}{dz} = \phi(z)\big(1-\phi(z)\big)$$

Computing the **last** derivative term:

$$\frac{dnet_{98}}{dw_{87,98}} = \frac{d\big(w_{87,98}o_{87} + w_{86,98}o_{86} + w_{85,98}o_{85} + \cdots\big)}{dw_{87,98}} = o_{87}$$

Computing the **middle** derivative term:

$$\frac{do_{98}}{dnet_{98}} = \frac{d\phi(net_{98})}{dnet_{98}} = \phi'(net_{98})\big(1-\phi(net_{98})\big) = o_{98}(1-o_{98})$$

Computing the **first** derivative term:

$$\frac{dE}{do_{98}} = \frac{dE}{dnet_{100}} \frac{dnet_{100}}{do_{98}}$$

since $\frac{dE}{dnet_{100}} = \delta_{100}$ **and** $\frac{dnet_{100}}{do_{98}} = \frac{d\big(w_{99,100}o_{99} + w_{98,100}o_{98} +\big)}{do_{98}} = w_{98,100}$

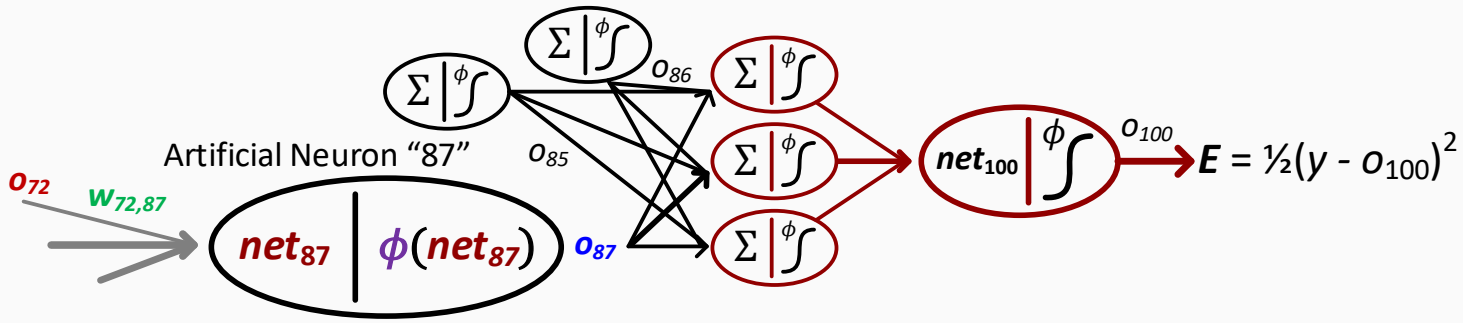
then $\frac{dE}{do_{98}} = \delta_{100}w_{98,100}$

The **result** of the calculation above is as follows:

$$\frac{dE}{dw_{87,98}} = \frac{dE}{do_{98}} \frac{do_{98}}{dnet_{98}} \frac{dnet_{98}}{dw_{87,98}} = \delta_{100}w_{98,100}o_{98}(1-o_{98})o_{87}$$

Backpropagation through another Hidden Layer in a Neural Network

The following illustrates calculations passing back through another **Artificial Neural Network** layer:



Below represents the derivative of error $\boldsymbol{E}$ using the **chain rule** from calculus:

$$\frac{dE}{dw_{72,87}} = \frac{dE}{do_{87}} \frac{do_{87}}{dnet_{87}} \frac{dnet_{87}}{dw_{72,87}}$$

$$\phi(z) = \frac{1}{1+e^{z}}$$

$$\phi'^{(z)} = \frac{d\phi(z)}{dz} = \phi(z)\big(1-\phi(z)\big)$$

Computing the **last** derivative term:

$$\frac{dnet_{87}}{dw_{72,87}} = \frac{d\big(w_{72,87}o_{72} + w_{71,87}o_{71} + w_{70,87}o_{70} + \cdots\big)}{dw_{72,87}} = o_{72}$$

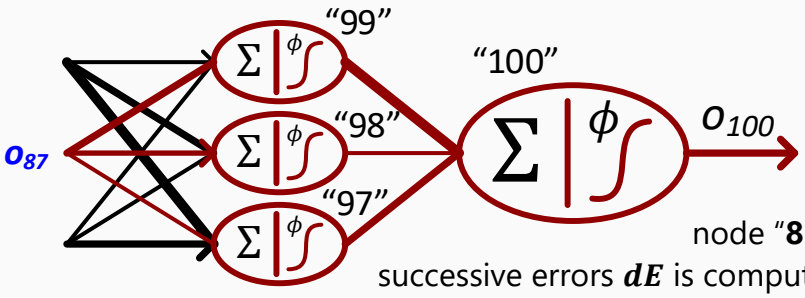
Computing the **middle** derivative term:

$$\frac{do_{87}}{dnet_{87}} = \frac{d\phi(net_{87})}{dnet_{87}} = \phi'(net_{87})\big(1-\phi(net_{87})\big) = o_{87}(1-o_{87})$$

Computing the **first** derivative term:

$$\frac{dE}{do_{87}} = \frac{dE}{dnet_{99}}\frac{dnet_{99}}{do_{87}} + \frac{dE}{dnet_{98}}\frac{dnet_{98}}{do_{87}} + \frac{dE}{dnet_{97}}\frac{dnet_{97}}{do_{87}} + \cdots$$

As the layers become deeper in the **Artificial Neural Network**, the computation of the **first derivative term** in calculating the weight for a given node's output becomes increasingly complex. This is due to the output of nodes within the hidden layer being dependent of the derivative of the error $\boldsymbol{dE}$ of all successive nodes to the final output o_{100} as computed and illustrated below:

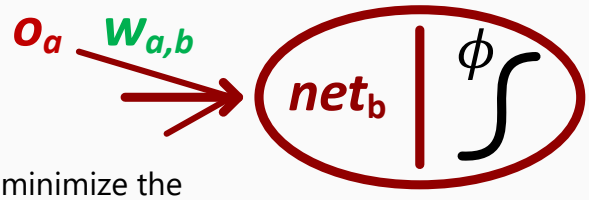


$$\delta_{99}w_{87,99} + \delta_{98}w_{87,98} + \delta_{97}w_{87,97} + \cdots = \sum_{\ell \in L} \delta_{87}w_{87,\ell}$$

Therefore, the derivative of the output of node "**87**" → o_{87} with respect to the derivative of all successive errors $\boldsymbol{dE}$ is computed as the sum of all delta terms from the prior calculations in the **Backpropagation Algorithm** → $\sum_{\ell \in L} \delta_{87}$ times the connectivity (**weight**) of the node "**87**" downstream in the current layer→ $w_{87,\ell}$.

Writing the above computation in a more general way to understand **Backpropagation Intuition**:

$$\frac{dE}{dw_{a,b}} = \frac{dE}{do_b}\ \frac{do_b}{dnet_b}\ \frac{dnet_b}{dw_{a,b}}$$



Assuming Neuron "***a***" connected to Neuron "***b***". The **optimization objective** is to adjust the **weight** $w_{a,b}$ to minimize the derivative of the error with respect to the **weight** $\frac{dE}{dw_{a,b}}$; using the **chain rule**:

$$\frac{dE}{dw_{a,b}} = \frac{dE}{do_b}\ \frac{do_b}{dnet_b}\ \frac{dnet_b}{dw_{a,b}} = \frac{dE}{do_b}\ \frac{do_b}{dnet_b}\ o_a$$

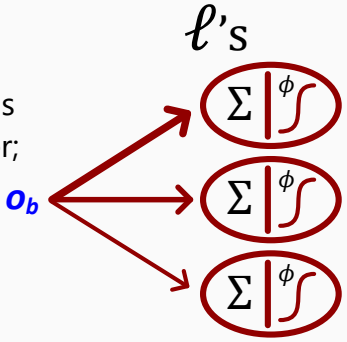
The **last** derivative term is always computed as the output of the prior node o_a.

$$\frac{dE}{dw_{a,b}} = \frac{dE}{do_b}\ \frac{do_b}{dnet_b}\ \frac{dnet_b}{dw_{a,b}} = \frac{dE}{do_b}\ o_b(1-o_b)o_a$$

The **middle** derivative term is always computed as the derivative of phi $\phi \rightarrow o_b(1-o_b)$.

The **first** derivative term is the sum of the deltas δ from all of the downstream nodes and the downstream weights $\left(\sum_{\ell \in L} \delta_\ell\, w_{b,\ell}\right)$

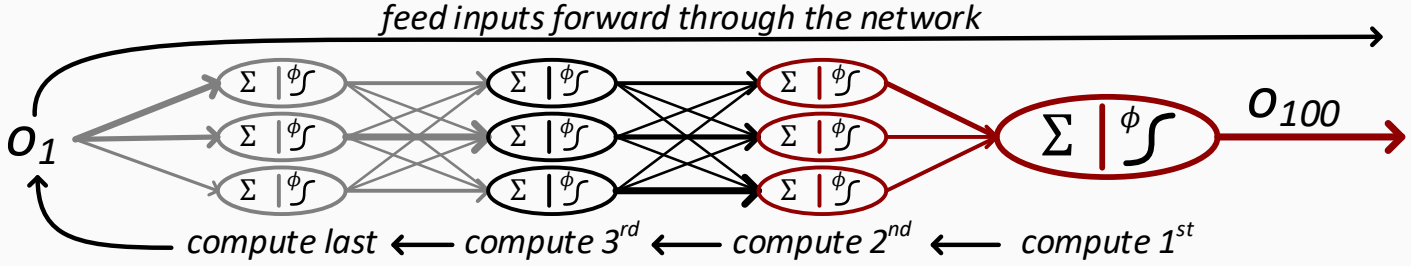
$$\frac{dE}{dw_{a,b}} = \frac{dE}{do_b}\ \frac{do_b}{dnet_b}\ \frac{dnet_b}{dw_{a,b}} = \left(\sum_{\ell \in L} \delta_\ell\, w_{b,\ell}\right) o_b(1-o_b)o_a$$



It is important that the L's represent the indices of **downstream neurons**. It is necessary to compute all of the δ_ℓ's subsequent to the current node in a layer; working backward along the neural network in order to compute. The computation as a whole represents the **gradient**:

$$\left(\sum_{\ell \in L} \delta_\ell\, w_{b,\ell}\right) o_b(1-o_b)o_a \rightarrow \textbf{Gradient Computation}$$

In other words, the implication of the above intuition is that the **gradients** are computed with respect to the **final output layer**, then the **gradients** are computed with respect to the **second last hidden layer**, then the **gradients** are computed with respect to the **third last hidden layer**, and so on... using the deltas ***δ's*** each time that were computed downstream of the current layer.



Given an understanding to compute the derivative $\boldsymbol{dE}$ with respect to the weight $\boldsymbol{w_{a,b}}$ for all weights $\boldsymbol{w_{a,b}}$'s, the following explains the mathematics behind **gradient descent**:

$$\boldsymbol{w_{a,b}} \longleftarrow \boldsymbol{w_{a,b}} - \boldsymbol{\alpha}\frac{dE}{d\boldsymbol{w_{a,b}}}$$

Each weight $\boldsymbol{w_{a,b}}$ is taken, with a **step-sized** $\boldsymbol{\alpha}$ **weight** $\boldsymbol{w_{a,b}}$ taken down the **gradient**.

In standard **Backpropagation**, $\boldsymbol{\alpha}$ is between **0** and **1** and referred to as the **learning rate**.

If the **learning rate** $\boldsymbol{\alpha}$ is too **low**, the neural network will learn very slowly. Conversely, if the **learning rate** $\boldsymbol{\alpha}$ is too **high**, the weights can cause an objective to **diverge** from the **optimal minimum**.

The general logic is that the weight changes depending on how **steep** the **gradient** is at that point.

Once the **errors** are **propagated** back through the neural network and the **weights** have been updated, the **error** must be updated; achieved by feeding the **input forward** through the network.

Neural Network Summarized

> **Backpropagation**: Repeat the process going **backwards** (**gradient** calculation), adjusting the **weights**, and feeding the **input forward** (**error** calculation) by many interactions for **learning**

Advantages of Neural Networks

- Highly expressive nonlinear models
- Advanced in computer vision and speech that is unmatched through competing methods.
- Capable of capturing latent structure with the hidden network layers

Disadvantages of Neural Networks

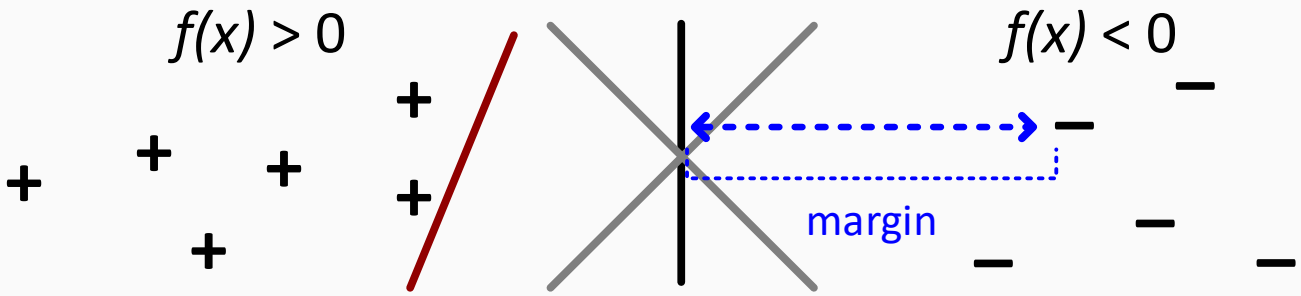
- Prone to becoming stagnant in local optima and produce poor solutions
- Another black-box algorithm that requires extensive parameter tuning (network structure).

support vector machines ⇄ (SVM)

Support Vector Machines are very principles algorithms that can be applied in an elegant manner.

- They arose directly out of Statistical Learning Theory
- The utilize optimization techniques for efficient and structured application
- They leverage kernels to allow for powerful nonlinear classification problem application

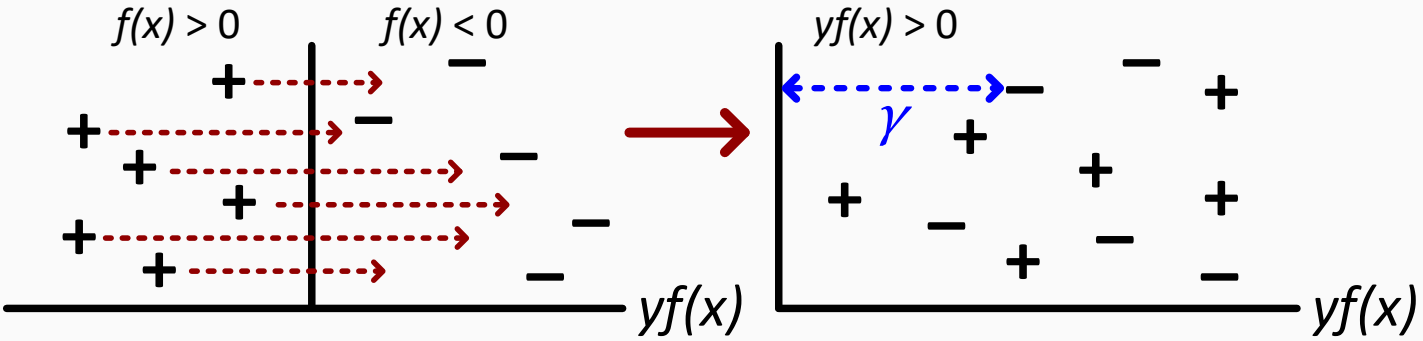
Support Vector Machines by addressing the questions of which **decision boundary** to select when all points in a dataset are classified correctly:



All of the above decision boundary assignments are equally plausible and equally 100% accurate. However, the **grey** boundaries and the **red** boundary are poor choices. They each fail at achieving the motivation behind **Support Vector Machines**.

The motivation behind **SVM** is to have all points as far away from the decision boundary as possible. As illustrated above, a decision boundary can be 100% accurate in classifying the points, but a poorly plotted decision boundary could cause a model to fail ay **generalization** to new data introduced.

The motivation above is achieved through examine the **margin** of the decision boundary; the distance between each data point and the decision boundary itself. In other words, the **minimum margin** in the model should ideally be **large**:



The **SVM optimization objective** is that all **points** $\boldsymbol{i}$ are farther than **gamma** $\boldsymbol{\gamma}$ from the decision **boundary**:

$y_i f(x_i) \geq \gamma, i = 1, \ldots, n$ → with the objective to **maxmimize** $\boldsymbol{\gamma}$

In formal terms, the objective is to **maximize** $\boldsymbol{\gamma}$, subject to the margins being **at least** $\boldsymbol{\gamma}$.

The first step to applying an **SVM**:

$\text{maximize } \boldsymbol{\gamma} \text{ such that } y_i f(x_i) \geq \boldsymbol{\gamma}, \quad i = 1, \ldots, n$

The issue existing with the first step is the ability to extract value from $\boldsymbol{\gamma}$ and $\boldsymbol{f(x_i)}$. The above step suffers from a scaling issue illustrated in example below:

$$y_i f(x_i) \times 100{,}000 \geq \boldsymbol{\gamma} \times 100{,}000, \qquad i = 1, \dots, n$$

Multiplying either side by a large number results in both $\boldsymbol{\gamma}$ and $\boldsymbol{f}(\boldsymbol{x_i})$ being arbitrarily large. Consequentially, $\boldsymbol{\gamma}$ will be forced into maintaining a relative size of $\boldsymbol{f}$:

$$y_i f(x_i) \geq \boldsymbol{\gamma} \times \text{"size } f\text{"}, \qquad i = 1, \dots, n$$

Although gamma $\boldsymbol{\gamma}$ is now meaningful, it remains arbitrary to which $\boldsymbol{f}$ is selected to apply. Meaning the equation will hold regardless of which equation $\boldsymbol{f}$ is selected ($\boldsymbol{2f}$ and $\boldsymbol{f}$ are equally good).

To address the arbitrary nature of f illustrated above, the expression will be set to 0:

$$y_i f(x_i) \geq \boldsymbol{\gamma} \times \text{size f} = 1 \;\longrightarrow \text{therefore } \boldsymbol{\gamma} = \frac{1}{\text{"size } f\text{"}}$$

The optimization objective after addressing the above issues:

$$\text{maximize } \gamma \text{ such that } y_i f(x_i) \geq \boldsymbol{\gamma} \times \text{size f} = 1$$

$$\text{maximize } \frac{1}{\text{"size } f\text{"}} \text{ such that } y_i f(x_i) \geq 1$$

The optimization problem now states to maximize 1 over the value of the size of f with the constraint that the minimum margin value is at least 1.

In regard to the function $f(x_i)$, a linear model will be chosen:

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}$$

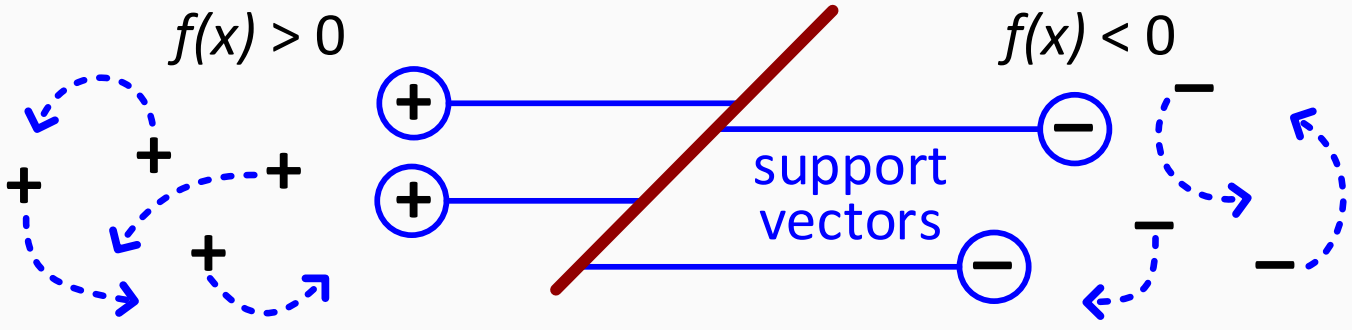
$$\text{size f} = \beta_1^2 + \beta_2^2 + \cdots + \beta_2^p = \|\beta\|_2^2 \longrightarrow \ell_2 \text{ normailization}$$

Applying the function f to the **SVM** objective:

$$\text{maximize } \frac{1}{\|\beta\|_2^2} \text{ such that } y_i f(x_i) \geq 1$$

$$\textbf{minimize } \|\beta\|_2^2 \text{ such that } y_i f(x_i) \geq 1$$

The **Support Vector Machine** optimization problem is solved through standard techniques in convex optimization (Lagrange multipliers, KKT conditions). Fortunately, the machinery designed to handle problems as such is fairly standard, with a wide array of toolboxes available specifically for optimization problems like **Support Vector Machines**.



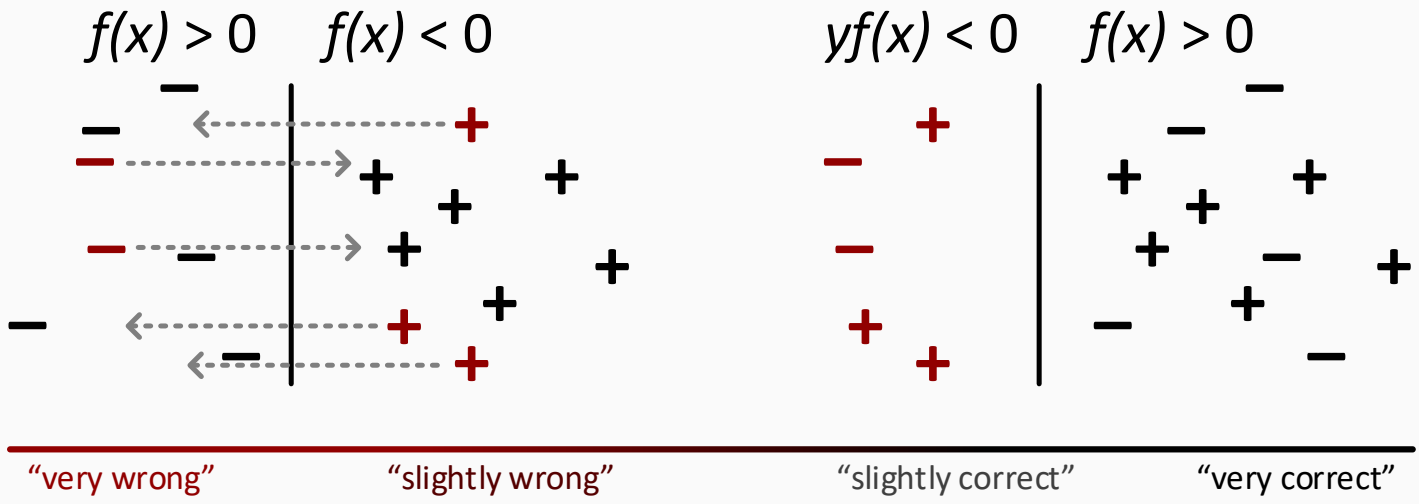
An important and interesting property regarding the solution to a Support Vector Optimization Problem is that only certain points will end up in the final solution for computing the β's. Meaning shifting the points on either side of the decision boundary around will have **no effect** on the final

solution. The special points that create the final solution are referred to as the **Support Vectors**. The **Support Vectors** are thus the points that are closest to the decision boundary on either side.

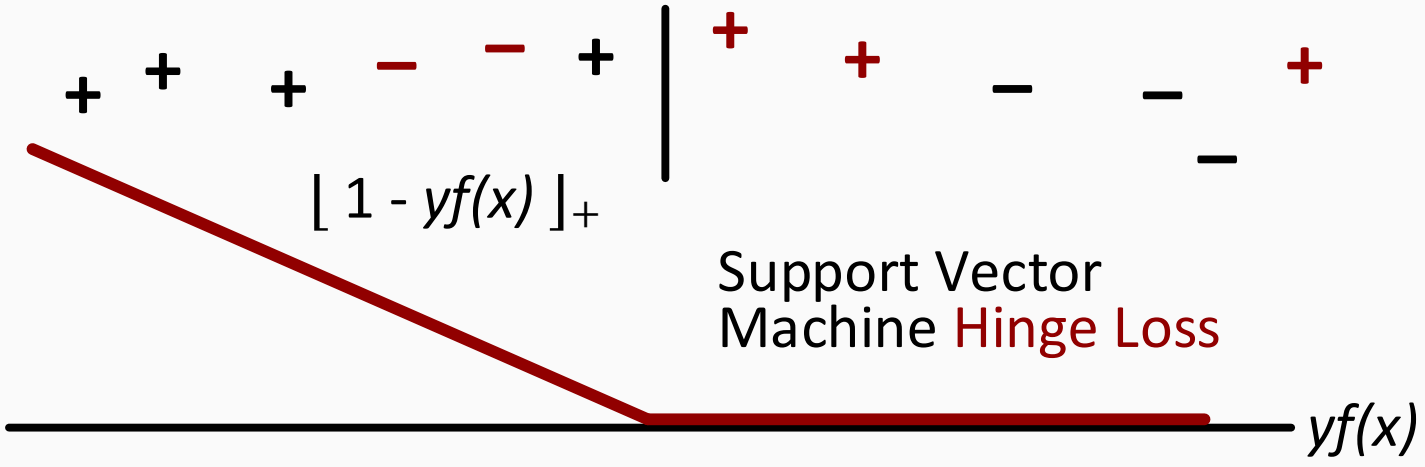
The Nonseparable Case

The problem illustrated above focused on all cases where the dataset can be separated perfectly by a given linear decision boundary. However, it is common for the dataset to contain much noise:

Returning to the geometric illustration familiar from before where the misclassified points are mapped to one side and the correctly classified points mapped to the other:



With the above understanding, it is necessary to implement that logic into the **SVM** function with the purpose of penalizing misclassified points more severely as they become farther away from the decision boundary they are misclassified with. The latter is achieved through the **SVM Hinge Loss**:



The intuition behind the above illustration of the **SVM Hinge Loss** is that when a point is correctly classified, there is no penalty; when a point is incorrectly classified, it is penalized in proportion to the distance away from the decision boundary. Therefore, the **more severe** the **misclassification** is, the heavier the respective penalty will be for the point. Note there is a slight overlap in the **Hinge Loss** function, indicating that a point can still suffer a slight amount of **loss** even it is correctly classified.

Applying the above **Loss Function** to the **Separable SVM** formulation:

$$\textbf{minimize } \|\beta\|_2^2 \text{ such that } y_i f(x_i) \geq 1$$

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}$$

Plugging in the **SVM Hinge Loss**:

$$\textbf{minimize } \|\beta\|_2^2 + C \sum_{i=1}^{n} [1 - y f(x)]_+$$

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}$$

The above formulation for **Nonseparable Cases of Support Vector Machines** is simply a special case of the standard **Machine Learning method minimizing a mixture of loss and regularization**:

$$f^* = \underset{\text{models } f}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell\big(y_i f(x_i)\big) + \boldsymbol{C} + \text{Regulatization}(f)$$

$$f(x_i) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \cdots + \beta_p x_{ip}$$

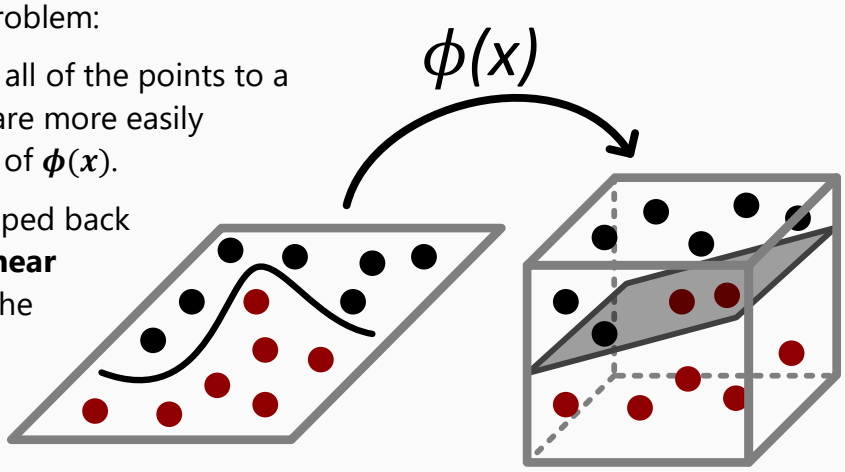
$$\text{Regularization}(f) = \beta_1^2 + \beta_2^2 + \beta_3^2 + \cdots + \beta_p^2 = \|\beta\|_2^2 \text{ (referred to as } \ell_2)$$

In practice, the optimization problem is solved using **Sequential Minimal Optimization** (**SMO**), where two variables are adjusted at a time. This is similar to **Coordinate Descent** utilized in **Boosting**. The difference is that a single variable cannot be adjusted at a time; this would violate constraints. Adjusting two variables at a time allows for flexibility to accommodate the latter constraints.

kernels for support vector machines (SVM)

Kernels allow for **Support Vector Machines** to convert from **linear models** to **nonlinear models** by utilizing a slightly different optimization problem:

The **Kernel "trick"** allows an **SVM** to map all of the points to a high dimensional space where the points are more easily separated; achieved through the mapping of $\boldsymbol{\phi(x)}$.



Then when the **decision boundary** is mapped back to the original dimensional space, a **nonlinear** boundary results that perfectly separates the data. The above illustration can be slightly misleading because the feature $\boldsymbol{\phi(x)}$ is almost never seen in the process.

An illustrative example of Kernels

Assuming a housing problem with the following features in the dataset:

$$x_i = [x_{i1}, x_{i2}, x_{i3}, x_{i4}, \dots] \rightarrow x_{i1} = \text{List Price}, x_{i2} = \text{Estimate}, x_{i3} = \text{Time on Market}, x_{i4} = \text{Avg Price}$$

For simplicity, the two features below will be of focus:

$$x_i = [x_{i1}, x_{i2}] \rightarrow x_{i1} = \text{House's List Price}, x_{i2} = \text{Zillow Estimate}$$

The **inner product** of features $[\boldsymbol{x_{i1}, x_{i2}}]$ is calculated as a metric to understand **distance in space** (2D)

$$\textbf{Inner Product}\ (x_i, x_k) = \boldsymbol{x_i} \times \boldsymbol{x_k} = x_{i1}x_{k1} + x_{i2}x_{k2}$$

In attempts to translate the above **2 dimensional** example into a **3 dimensional** space, the points are mapped through the function $\boldsymbol{\phi(x)}$ as follows:

$$x_i = [x_{i1}, x_{i2}]$$

$$\boldsymbol{\phi(x_i)} = \phi([x_{i1}, x_{i2}]) = [x_{i1}^2, x_{i2}^2, x_{i1}x_{i2}] \qquad (\textbf{2D} \rightarrow \textbf{3D for } \boldsymbol{i})$$

$$\boldsymbol{\phi(x_k)} = \phi([x_{k1}, x_{k2}]) = [x_{k1}^2, x_{k2}^2, x_{k1}x_{k2}] \qquad (\textbf{2D} \rightarrow \textbf{3D for } \boldsymbol{k})$$

$\boldsymbol{x}$ is above being mapped from a **2D** to a **3D** space and the **inner product** can be computed in **3D**:

$$\textbf{Inner Product}\ \big(\boldsymbol{\phi(x_i)}, \boldsymbol{\phi(x_k)}\big) = \boldsymbol{\phi(x_i)} \times\ \boldsymbol{\phi(x_k)} = x_{i1}^2, x_{i2}^2 + x_{k1}^2, x_{k2}^2 + x_{i1}x_{i2}x_{k1}x_{k2}$$

The above multiplies each dimension **component-wise** and computes the summation.

Therefore, one way to map to a high dimensional space with **SVM** is:

- Decide on the term $\boldsymbol{\phi(x)}$
- Replace all values of x_i in the optimization problem with $\boldsymbol{\phi(x_i)}$
- Solve for the new optimization problem

Alternate Kernel Application for SVM

There is a special property of the SVM optimization problem:

- The problem only depends on $\boldsymbol{x_i}$ through the inner products
- The only terms involving $\boldsymbol{x_i}$ are **Inner Product** $(\boldsymbol{x_i}, \boldsymbol{x_k})$
- $\boldsymbol{x_i}$ will never occur alone or in other ways, they will only occur in inner products

The prior optimization problem derived for **SVM** is referred to as the **Primal Problem**. The alternate method introduced above is referred to as the **SVM Dual Form** of this problem:

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_1 - \frac{1}{2} \sum_{i,\ k-1}^{m} \alpha_i \alpha_j y_i y_k \left(x_i^T x_k \right) \leftarrow \textbf{Inner Product}$$

$$\text{subject to } 0 \leq \alpha_j \leq C, \quad i = 1, \ldots, m \text{ and } \sum_{i=1}^{m} \alpha_i y_i = 0$$

In the objective above, it is noted that the only occurrence of $\boldsymbol{x}$ is inside the **inner product**. Therefore, rather than determining what $\boldsymbol{\phi}$ is for the purpose of mapping the data to a high dimensional space, the **inner product** in the original space can be replaced with an **inner product** in a different space.

Wherever the formulation states to compute the $\textbf{Inner Product}\ (\boldsymbol{x_i}, \boldsymbol{x_k})$, it will be replaced with the $\textbf{Inner Product}\ \left(\boldsymbol{\phi}(\boldsymbol{x_i}), \boldsymbol{\phi}(\boldsymbol{x_k})\right)$; There computation of $\boldsymbol{\phi}(\boldsymbol{x_i})$ is thus never required.

The question arises: How to compute the **inner products** in the new space without computing $\boldsymbol{\phi}$'s?

Using the previous example to illustrate the solution to the question:

$x_i = [x_{i1}, x_{i2}]$

$\textbf{Inner Product}\ (x_i, x_k) = \boldsymbol{x_i} \times \boldsymbol{x_k} = x_{i1}x_{k1} + x_{i2}x_{k2}$

$\textbf{Inner Product}\ \left(\boldsymbol{\phi}(\boldsymbol{x_i}), \boldsymbol{\phi}(\boldsymbol{x_k})\right) = \boldsymbol{\phi}(\boldsymbol{x_i}) \times\ \boldsymbol{\phi}(\boldsymbol{x_k}) = x_{i1}^2, x_{i2}^2 + x_{k1}^2, x_{k2}^2 + x_{i1}x_{i2}x_{k1}x_{k2}$

$\textbf{Inner Product}\ (x_i, x_k) \Longrightarrow \textbf{Inner Product}\ \left(\boldsymbol{\phi}(\boldsymbol{x_i}), \boldsymbol{\phi}(\boldsymbol{x_k})\right)$

It does not matter if $\boldsymbol{\phi}$ is infinite dimensional or impossible to calculate, as long as the **Kernel** can be calculated between two points, the optimization formula can be run.

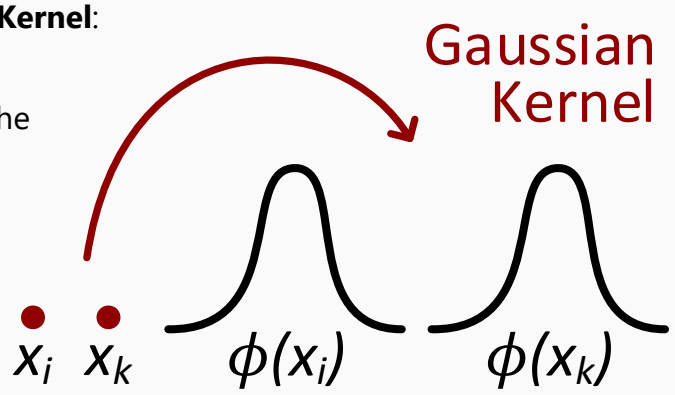
$$\textbf{Inner Product}\ (x_i, x_k) \Longrightarrow \textbf{Inner Product}\ \left(\boldsymbol{\phi}(\boldsymbol{x_i}), \boldsymbol{\phi}(\boldsymbol{x_k})\right) = \boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_k})$$

A particularly useful Kernel in practice is the **Gaussian Kernel**:

In this case, phi $\boldsymbol{\phi}$ cannot be effectively calculated on a computer due to its infinite dimensionality. However, the **inner product** can be calculated instead:

$$\boldsymbol{K}(\boldsymbol{x_i}, \boldsymbol{x_k}) = \exp\left(-\frac{\|x_i - x_k\|_2^2}{2\sigma^2}\right), \text{where } \boldsymbol{\sigma} \text{ is selected}$$

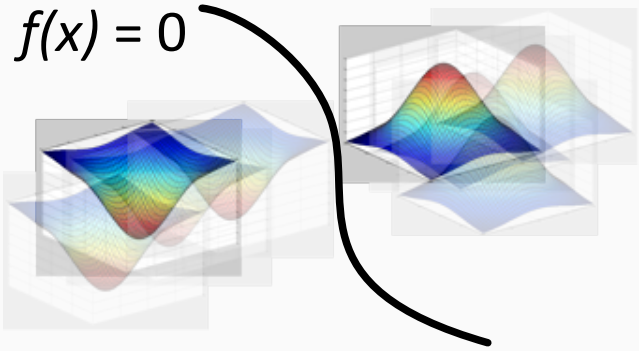
Therefore **Support Vector Machine** is now able to be performed in an **infinite dimensional space**.



When using a **Gaussian Kernel**, also referred to as a **Radial Basis Function (RBF) Kernel**, a **normalized**, **high-dimensional distribution** is essentially being applied to each datapoint. They are subsequently weights and summed. The distributions can be both **positive (+)** and **negative (-)** depending on the solution to the optimization problem. When a prediction is computed for a new observation in the dataset, a weighted summation of the normalized distributions is computed:

$$f(x) = \sum_{i=1}^{n} \alpha_i K(x, x_i) + b$$

→ the **weights α_i** are determined by the solution to the SVM optimization problem



The **Decision Boundary** is simply constructed where the function $\boldsymbol{f(x) = 0}$ as illustrated →

The examples above illustrated how **Support Vector Machines** are able to compute **complex nonlinear decision boundaries.**

It is important to note that if the **bandwidth parameter** is too small, the model can **overfit** data.

The **bandwidth parameter** is the **standard deviation** of the **Gaussian Distribution**.

Common Kernels for Support Vector Machines

- Linear Kernel (synonymous to no kernel and using the regular inner product)
- Polynomial (the degree of polynomial is chosen and will overfit the data if too large)
- Gaussian Kernel (the bandwidth is chosen and will overfit the data if too large)

Advantages of Support Vector Machines

- Easily handles nonlinearities (with Kernels)
- Based on quadratic programing with specialized solvers (like coordinate descent with dual-adjustments to variables at a time)
- Reproducible results regardless of user or runs (theoretically)
- Easily handles imbalanced data by reweighting the points

Disadvantages of Support Vector Machines

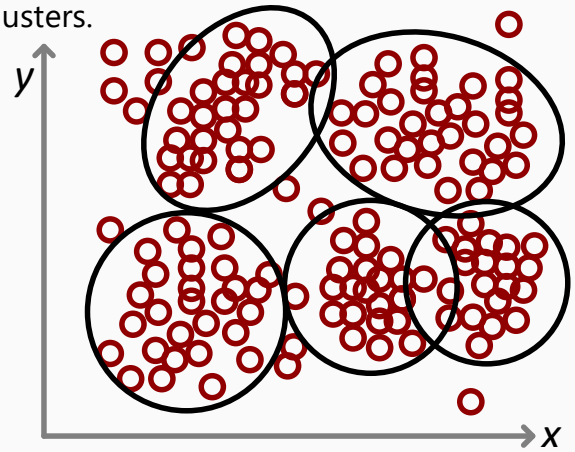
- The solvers can sometimes be slow
- Does not naturally scale to larger datasets
- The performance is not as reliable in practice (even after adjusting the kernel parameters)
- Support Vector Machine models are uninterpretable
- The choice of Kernel can be difficult

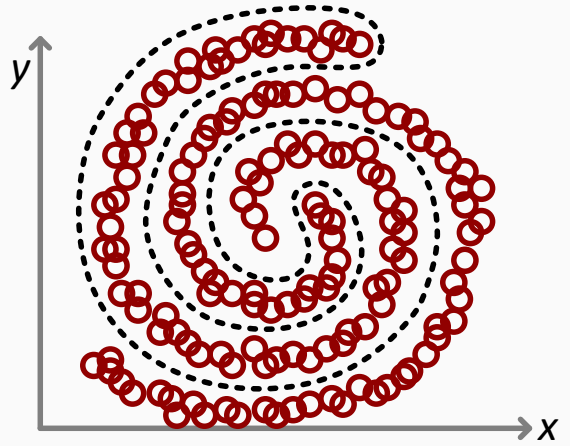
module6 · clustering and recommenders

unsupervised learning clustering

Unsupervised Learning implies the training data has no actual labels to learn from.
The goal in **Clustering** is to group the datapoint with unknown labels into something similar.

In the illustration to the right, the data exhibits five physical clusters. The algorithm applied needs to automatically be able to identify these clusters. This can become very difficult when the clusters are less obvious like in the alternate example. There are two clusters present in the alternate example and would be harder to identify with an **unsupervised clustering learning algorithm**. There is an algorithm that will handle each problem well, but each algorithm cannot handle the example it is not designed to be applied to.





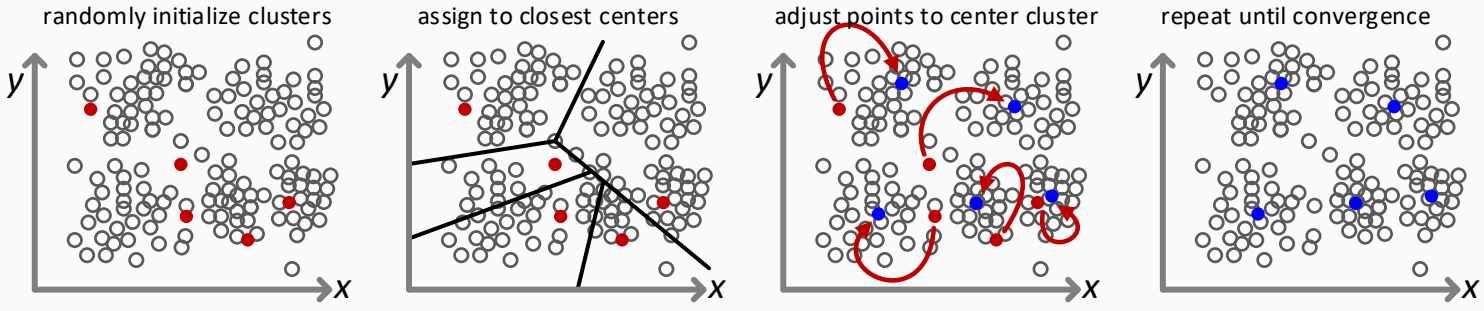
Clustering Applications

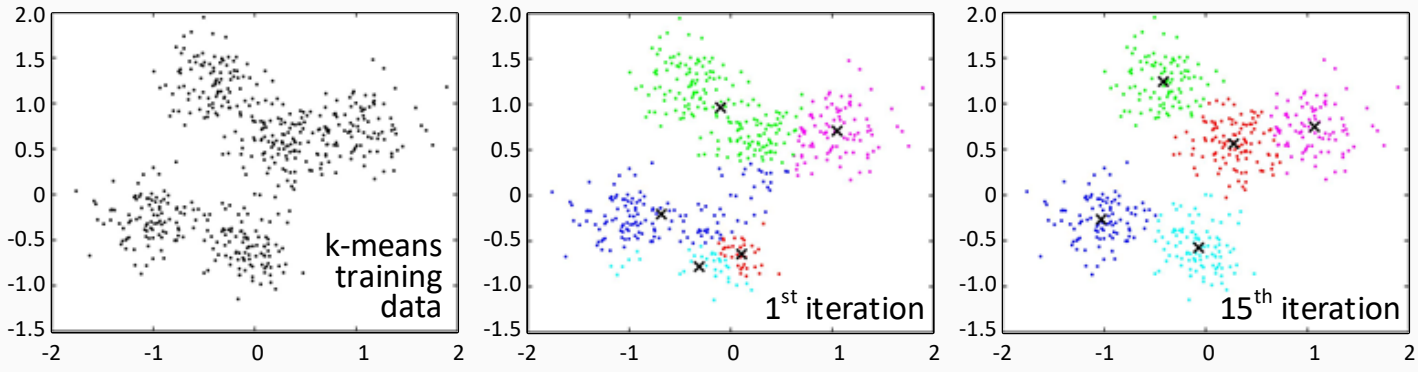
- Automatically group documents/webpages into topics
  - Grouping daily news stories into categories
- Clustering large numbers of products
  - Etsy products or Ebay listings
- Clustering customers into those with similar behavior

K-means Clustering

One of the most widely used clustering methods in practice.

- Input number of clusters k, randomly initialize centers
- Assign all points to the nearest cluster k_i center
- Change cluster k_i's centers to centralize datapoints
- Repeat until convergence





formal k-means clustering

Input: Dataset $x_1, \dots, x_n$, number of clusters K

Output: Cluster centers $c_1, \dots, c_k$

Goal: Minimize:

$$\text{cost}(c_1, \dots, c_k) = \sum_i \min_k \big(\text{dist}(x_i, c_k)\big)$$

The **objective** of **K-Means** Clustering is to **minimize** the **distance** between a point $\boldsymbol{x_i}$ and its nearest cluster center $\boldsymbol{c_k}$ summing all computed distances together. The **goal** to determine the cluster centers that minimize the **total distance** between all points $\boldsymbol{x_i}$ and the cluster centers (**Global Objective**).

Unlike Support Vector Machines and Logistic Regression, **K-Means** Clustering cannot optimize in a single step. **Global Minimization**: Try all possible assignments of m points to $\boldsymbol{K}$ clusters:

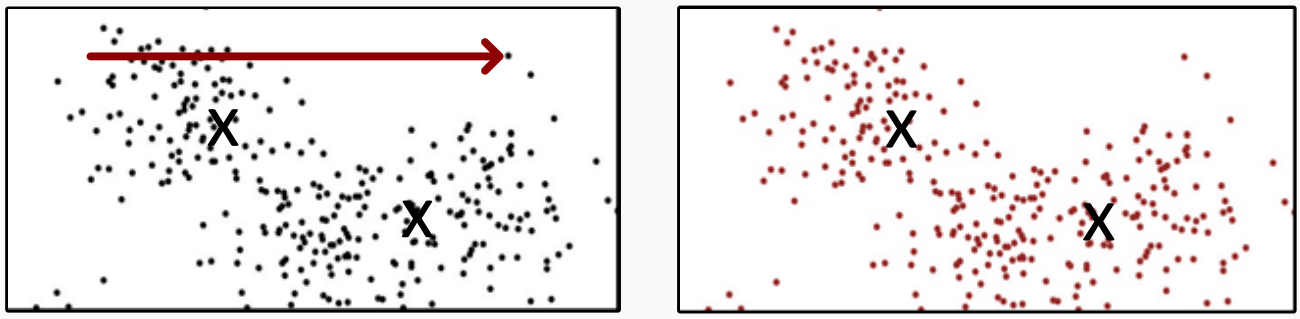
$$\text{Combinations}(m, K) = \frac{1}{K!} \sum_{k=1}^{K} (1)^{K-k} \binom{K}{k} k^n$$

$$\text{Combinations}(10, 4) = 34{,}000, \text{Combinations}(19, 4) = 10^{10}, \dots, \text{noncomputable}$$

Thus, **K-Means** Clustering is an approximation due the cluster assignments being indeterminable.

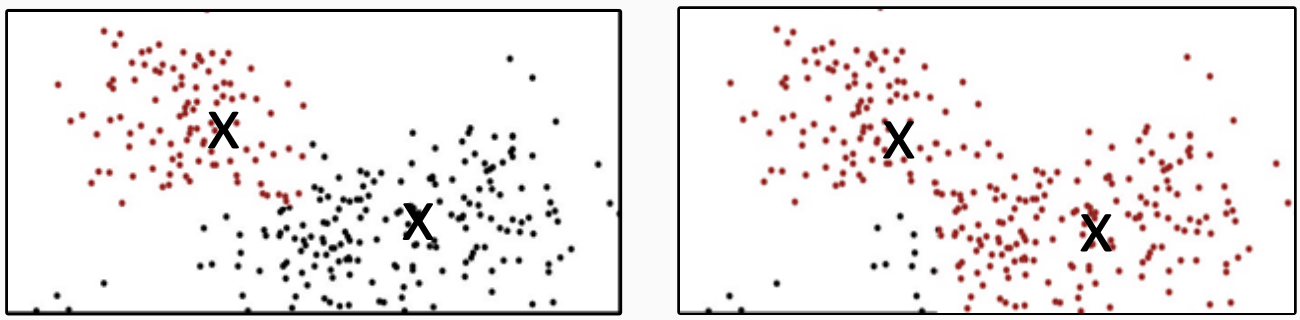
The **cost function** computes a sum over the points of distance to the nearest cluster center:

$$\text{cost}(c_1, \dots, c_k) = \sum_i \min_k \big(\text{dist}(x_i, c_k)\big)$$



The clusters can be summed at once from left to right as illustrated in the example above:

$$\text{cost}(c_1, \dots, c_k) = \sum_k \sum_{i: x_i \text{ is in cluster}_k} \text{dist}(x_i, c_k)$$



The clusters can equally be summed over by adding them up in order of the clusters illustrated above:

Illustrating a more general cost by focusing on the cluster assignments as well as the clusters $\boldsymbol{c_k}$:

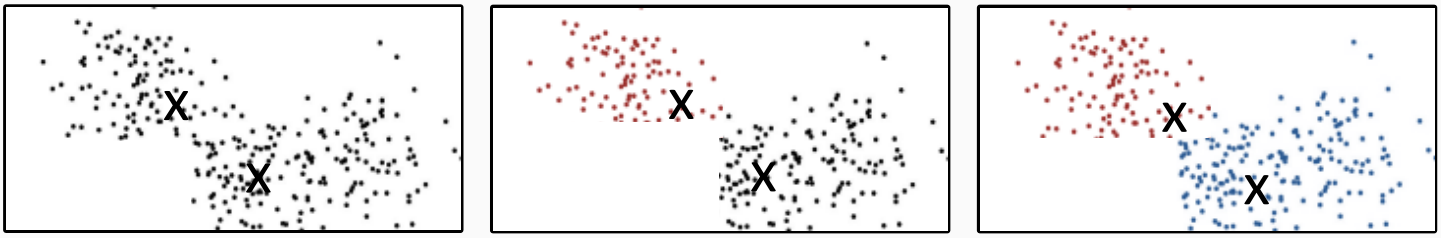
$$\text{cost}(\text{cluster}_1, \text{cluster}_2, \ldots, \text{cluster}_k, c_1, \ldots, c_K) = \sum_k \sum_{i:x_i \text{ is in cluster}_k} \text{dist}(x_i, c_k)$$

If the expression above was treated as just a functions of the cluster assignments $\boldsymbol{\text{cluster}_k}$ only:

$$\text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, c_1, \ldots, c_K) = \sum_k \sum_{i:x_i \text{ is in } \boldsymbol{\text{cluster}_k}} \text{dist}(x_i, c_k)$$

The best way to assign the $\boldsymbol{\text{clusters}_k}$ is to assign all of the points to the nearest cluster center $\boldsymbol{c_k}$:

$$\min_{\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}} \text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, c_1, \ldots, c_K)$$

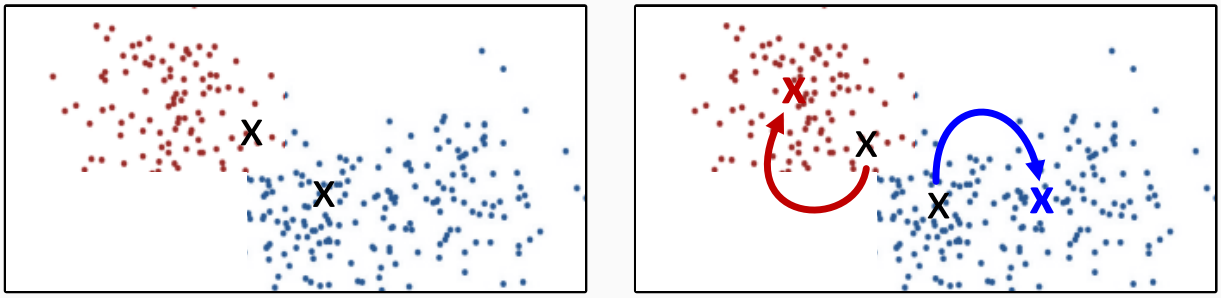


Alternatively, if the generalize cost function focused strictly on assigning the reference points $\boldsymbol{c_k}$ only:

$$\text{cost}(\text{cluster}_1, \text{cluster}_2, \ldots, \text{cluster}_k, \boldsymbol{c_1, \ldots, c_K}) = \sum_k \sum_{i:x_i \text{ is in } \boldsymbol{\text{cluster}_k}} dist(x_i, \boldsymbol{c_k})$$

The best way to minimize **cost** is to assign the centers to the middle of points assigned to that cluster:

$$\min_{\boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_K}} \text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, \boldsymbol{c_1, \ldots, c_K})$$



The above illustration assigns cluster centers by minimizing average distance to the reference points.

Thus the **generalize cost function** below is dependent on both the cluster assignments $\boldsymbol{\text{cluster}_k}$ and the cluster centers $\boldsymbol{c_K}$.

$$\text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, \boldsymbol{c_1, \ldots, c_K}) = \sum_k \sum_{i:x_i \text{ is in } \boldsymbol{\text{cluster}_k}} \text{dist}(x_i, \boldsymbol{c_k})$$

Input: number of clusters $\boldsymbol{K}$, randomly initialize centers $\boldsymbol{c_k}$

Until converged:

Assign all points to the closest cluster center

$$\min_{\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}} \text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, c_1, \ldots, c_K)$$

Change cluster centers to be in the middle of its points

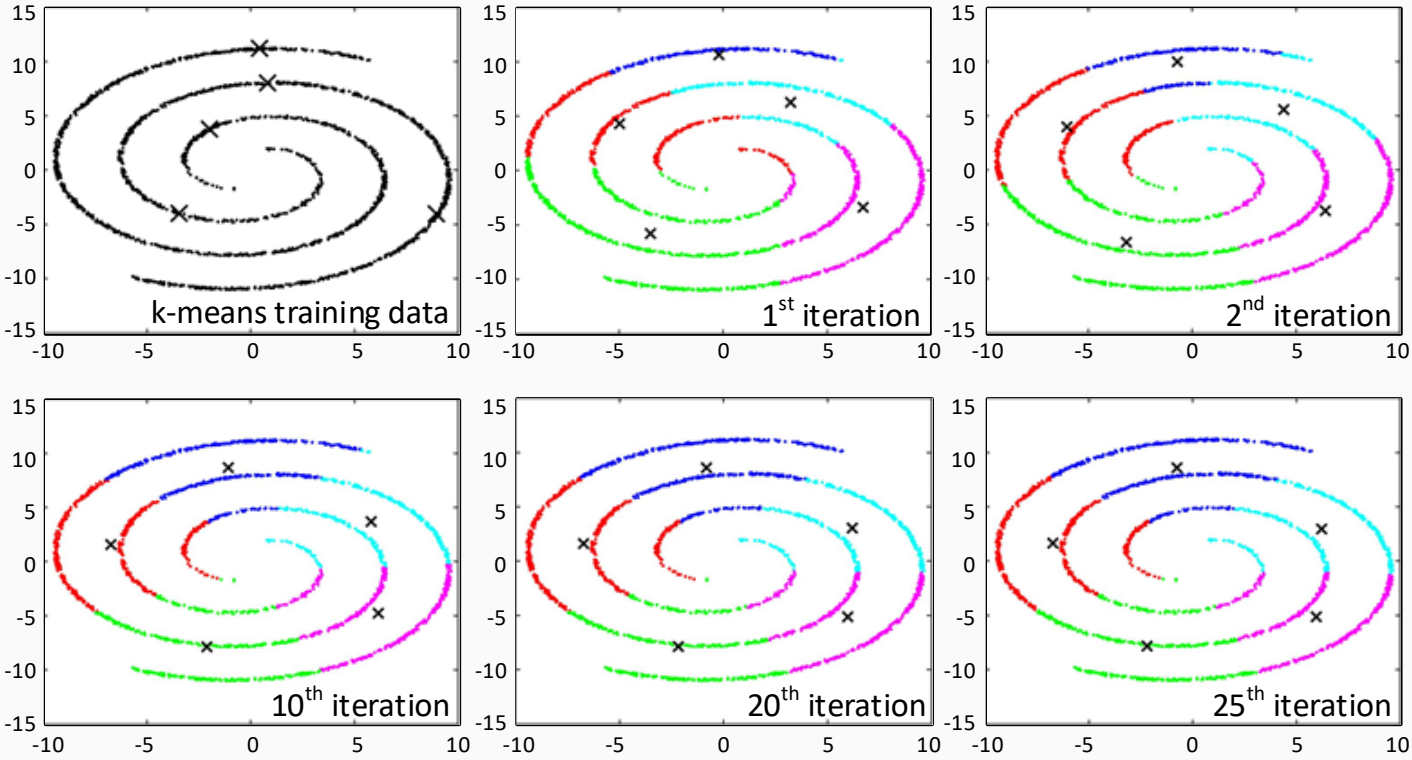
$$\min_{\boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_K}} \text{cost}(\boldsymbol{\text{cluster}_1}, \boldsymbol{\text{cluster}_2}, \ldots, \boldsymbol{\text{cluster}_k}, \boldsymbol{c_1, \ldots, c_K})$$

K-Means Clustering is functions in alternating iterations of assigning clusters and centering them (**alternation minimization**).

K-means does not always achieve its goal of minimizing the cost function (**Global Minimization**):

$$\text{cost}(\mathbf{cluster}_1, \mathbf{cluster}_2, \dots, \mathbf{cluster}_k, c_1, \dots, c_K)$$

K-Means often requires multiple replicates and is possible that K-means goal is not the correct one:

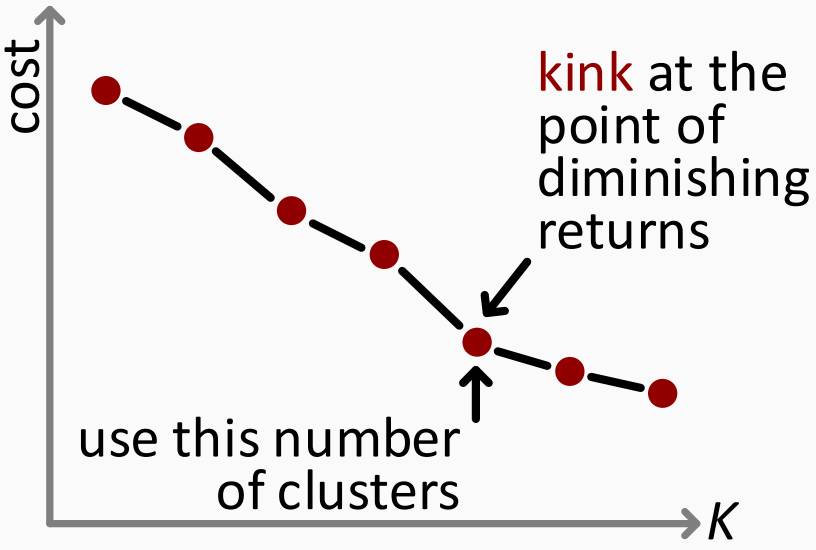


The above illustration simulates the K-Means optimization algorithm on a set of data. The first image illustrates the **random initialization** of **cluster centers**. After many iterations, the model has "**converged**". It is evident the 25 iterations in the above experiment did not yield meaningful results.

choosing k for k-means clusting

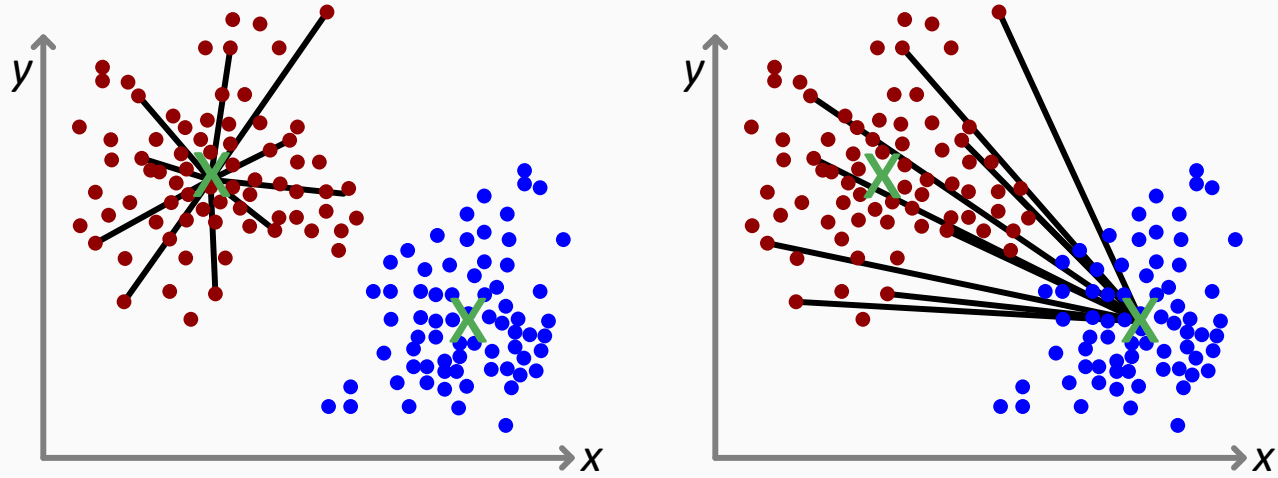
There are several acceptable options for choosing the optimal amount of clusters K.

One of the more widely used practical applications is to plot the number of clusters K against the **cost** and examine for the point of **diminishing returns**. As the number of clusters increases, the cost functions will decrease. After a certain threshold, the decrease in cost is insignificant to the additional of new clusters K.



Another option for optimal number of clusters K examines the ratio of **average distance of the assigned cluster center** to the **average distance of the other cluster centers**.

First compute the distance to the closest assigned cluster center;
Then compare the latter computation to the average of the distance to the other centers.



Ideally, the two separate computations above should differ from each other; implying that the clusters are not only tightly modeled, but far away and clearly separated from other clusters.

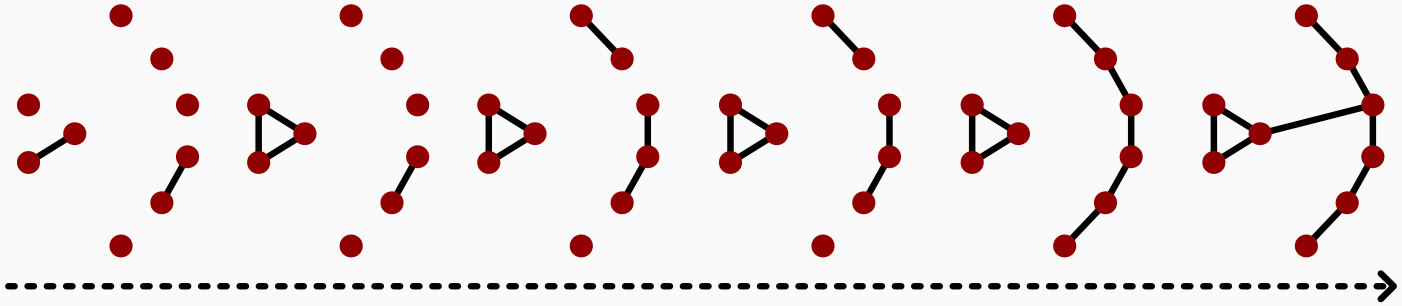
In application, the alternate metric to **choosing the number of clusters** K is not ideal for $K > 2$.

K-Means Clustering Summary

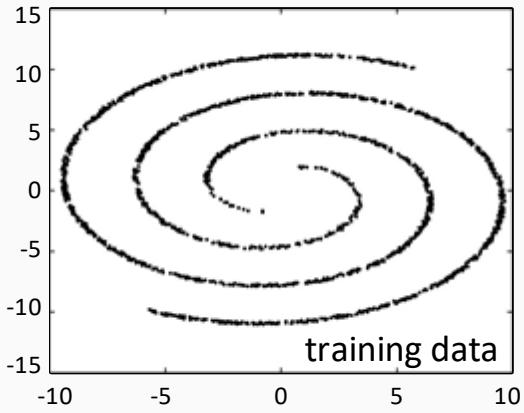
- Popular clustering algorithm, computationally efficient
- Performs alternating minimization on a cost function
- Does not always fully minimize the cost function
  (multiple replicates might be needed for a good solution)
- Can use the cost function to evaluate whether one replicate is better than another
- Can use cost function to help choose the number of clusters
- Does not work well for highly non-spherical clusters
- Euclidean distance is often used, but other distances can equally be used

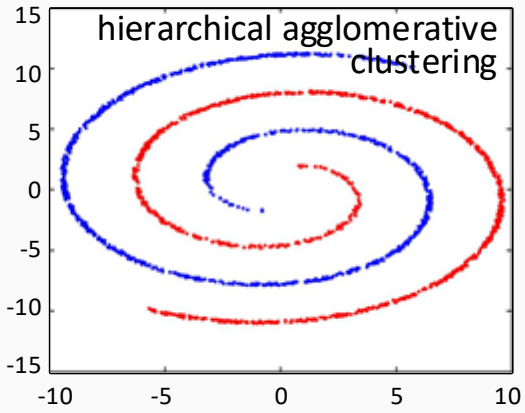
hierarchical agglomerative clustering

- Begins with each datapoint in its own cluster
- Repeatedly merges the clusters of the two closest points

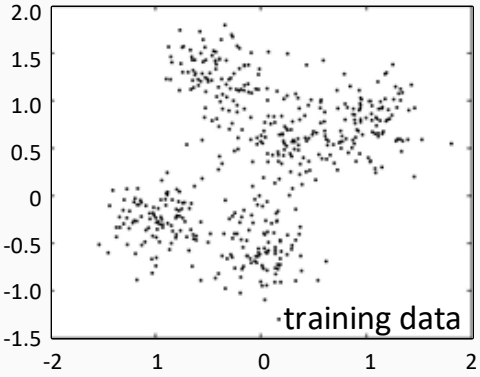


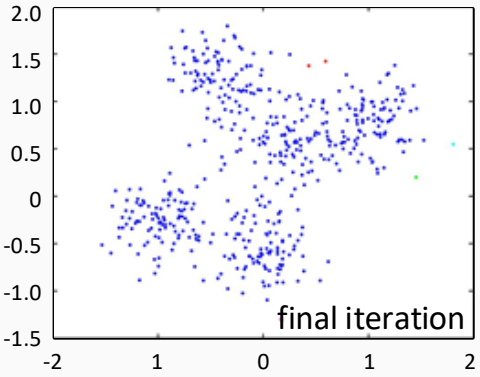
Hierarchical Agglomerative Clustering can handle data that **K-Means** fails to converge effectively:

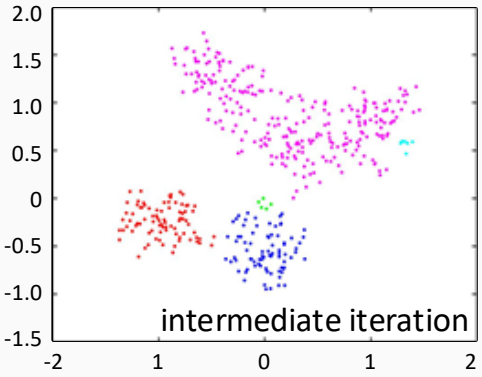




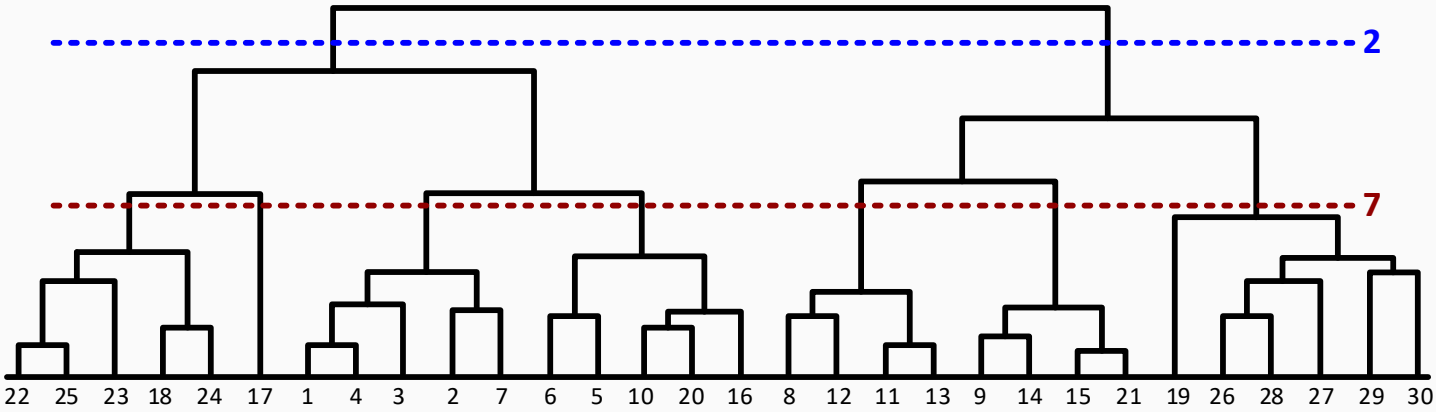
However, **Hierarchical Agglomerative Clustering** can also fail to converge on datasets the **K-Means** performs accurately on:







Dendrogram for Hierarchical Agglomerative Clustering

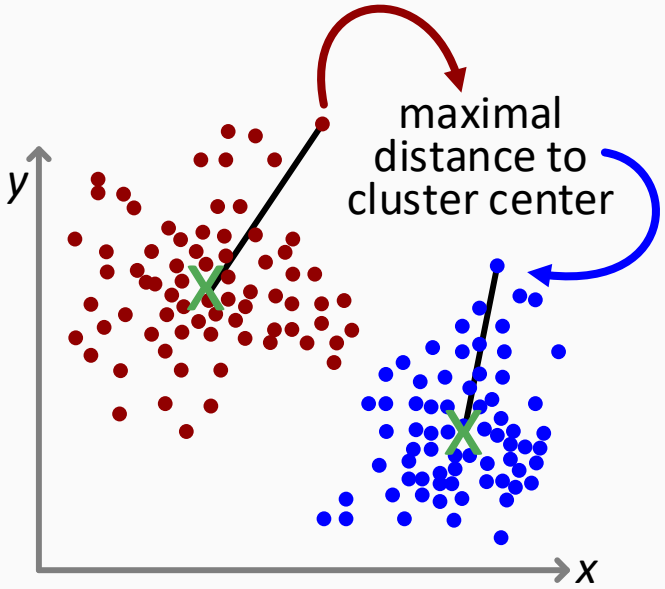


Hierarchical Agglomerative Clustering produces a **dendrogram** that map the nature of clusters.

The **dendrogram** can be pruned to control the number of clusters assigned to the data (seen above).

A problem with Hierarchical Agglomerative Clustering (HAC)

- The number of points in each cluster
  - If the number of points assigned to clusters **<** the total number data points available, some data could fail to be assigned to a cluster with the **Hierarchical Agglomerative Clustering** algorithm.
  - K-Means will not experience the aforementioned problem.
- Maximal distance to the cluster center
  - The maximal distance to cluster center is the largest distance of any point in a cluster
  - If the maximal distance is ≫ **much larger** ≫ then the average distance to the cluster center, the cluster may be overly dispersed.



K-Means Compared to Hierarchical Agglomerative Clustering

- Both are useful for different types of problems. **K-Means** works well with **spherical data**.
- **Hierarchical Agglomerative Clustering** is useful when clusters are **well-separated**. (Meaning data close together should be in the same cluster.)
- For **K-means**, one needs to choose the number of clusters.
  For **hierarchical clustering**, one chooses when to stop merging clusters.
- The **distance metric** is important and can have a large impact on the solution.

recommender systems

As a working example applying the various techniques of **Recommender Systems**, the Netflix Competition offering a $1 million prize to the team who could improve their existing **Recommender System** by **10%**, is used.

Assuming the following sample data:

	Alien	Bug's Life	Cars	Dark Night
Carmen	5	–	4	1
Joseph	5	4	–	–
Leonore	1	7	–	3
Esmerelda	2	8	1	–

The idea of a **Recommender System** in this context is to use the crowd's votes to complete entries.

There are two prominently used methods for Recommender Systems

- **User-Based Collaborative Filtering**
- **Item-Based Collaborative Filtering**

User-Based Collaborative Filtering

Predict Carmen's rating for Alien using **similar user ratings**

	Alien	Bug's Life	Cars	Dark Night
Carmen	?	4	4	1
Joseph	5	4	–	2
Leonore	1	1	–	3
Esmerelda	2	3	1	–

Item-Based Collaborative Filtering

Predict Joseph's rating for Dark Night using **similar items' ratings**

	Alien	Bug's Life	Cars	Dark Night
Carmen	2	–	1	1
Joseph	5	4	2	?
Leonore	4	–	3	3
Esmerelda	–	4	1	–

User-Based Collaborative Filtering and Correlation

$$\widehat{\boldsymbol{R}}(\mathbf{Carmen}, \mathbf{Alien}) = \bar{R}_{\text{Carmen}} + \frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times \left(R_{U,\mathbf{Alien}} - \bar{R}_U\right)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$$

Where $\boldsymbol{sim} =$

$$sim(\mathbf{Carmen}, U) = \frac{\sum_{\text{Movies } m} \left(R_{\text{Carmen},m} - \bar{R}_{\text{Carmen}}\right) - \left(R_{U,m} - \bar{R}_U\right)}{\sqrt{\sum_{\text{Movies } m} \left(R_{\text{Carmen},m} - \bar{R}_{\text{Carmen}}\right)^2} \sqrt{\sum_{\text{Movies } m} \left(R_{U,m} - \bar{R}_U\right)^2}}$$

$$\widehat{R}(\mathbf{Carmen, Alien}) = \bar{R}_{\text{Carmen}} + \frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$$

$\widehat{R}(\mathbf{Carmen, Alien})$ → The estimation of **Carmen's** rating of **Alien**, which she has not seen

$$\widehat{R}(\mathbf{Carmen, Alien}) = \bar{R}_{\text{Carmen}} + \frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$$

$\bar{R}_{\text{Carmen}}$ → **Carmen's** average rating → $\hat{R} > \bar{R}$ if it is expected that **Carmen** will enjoy **Alien**

$$\widehat{R}(\mathbf{Carmen, Alien}) = \bar{R}_{\text{Carmen}} + \frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$$

$\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)$ → All other **Users** are considered whether or not they rated **Alien** above or below the **User's** average rating. Each **User** is weighted by their similarity to **Carmen**; taking a weighted average of the **User's** votes.

$$\widehat{R}(\mathbf{Carmen, Alien}) = \bar{R}_{\text{Carmen}} + \frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$$

$\frac{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U) \times (R_{U,\mathbf{Alien}} - \bar{R}_U)}{\sum_{\text{Users } U} sim(\mathbf{Carmen}, U)}$ → The **Users** are weighted by how similar they are to **Carmen**. The **Users** who are more similar than others will be assigned larger weights; the denominator dictates all weights will sum to one, ultimately providing a prediction of **Carmen's** rating for **Alien**.

It is noted the sum over movies rated by both users term in the denominator:

$$\sqrt{\sum_{\text{Movies } m} \left(R_{\text{Carmen},m} - \bar{R}_{\text{Carmen}}\right)^2}$$

The above term is only summed over the movies that both users have rated. Therefore, the metric will favor users who have watched few movies that overlap with Carmen, but agree with Carmen on the ratings of movies they have both watched.

User-Based Collaborative Filtering and Correlation

- A simple approach to apply and interpret within a model
- However, if there are many users & movies with few ratings, similarities will be inaccurate and estimates will likely be poor
- Does not take into account how similar **items** are with each other

Item-Based Collaborative Filtering

The algorithm applied in **Item**-Based Collaborated Filtering is the same as **User**-Based. The correlations are used to weight the previous recommendations by how similar an **item** is to past items.

	Alien	Bug's Life	Cars	Dark Night
Carmen	2	–	1	1
Joseph	5	4	2	?
Leonore	4	–	3	3
Esmerelda	–	4	1	–

Item-Based Collaborative Filtering and Correlation

- Simple to use and is generally more robust in making predictions with **item-item** similarities opposed to **user-user** similarities
- Suffers from poor estimates when items are either not popular or do not contain much data
- Does not take into account how similar **users** are to the current one

matrix factorization

Matrix Factorization takes into account both the similarities between other user's ratings and items.

$$\textbf{Carmen}\begin{bmatrix}\textbf{Alien} & \textbf{Bug's Life} & \textbf{Cars} & \textbf{Dark Night}\\ 5 & - & 1 & 3\end{bmatrix}$$

$$\textbf{Carmen}^{\mathbf{1}}\begin{bmatrix}\textbf{Scary} & \textbf{Kiddy}\\ 5 & 1\end{bmatrix} \rightarrow \textbf{Dark Knight}\begin{bmatrix}\textbf{Scary} & \textbf{Kiddy}\\ \frac{1}{2} & \frac{1}{2}\end{bmatrix}$$

Prediction from **Carmen's** rating of **Dark Knight**: $\mathbf{5}\times\frac{\mathbf{1}}{\mathbf{2}}+\mathbf{1}\times\frac{\mathbf{1}}{\mathbf{2}}=\mathbf{3}\rightarrow$ **inner product of the vectors**

$$\textbf{Carmen}^{\mathbf{1}}\begin{pmatrix}\textbf{Scary} & \textbf{Kiddy}\\ 5 & 1\end{pmatrix}\begin{pmatrix}\textbf{Dark Knight}\\ 1/2\\ 1/2\end{pmatrix}\begin{matrix}\textbf{Scary}\\ \textbf{Kiddy}\end{matrix} = \textbf{Carmen}^{\mathbf{1}}\begin{pmatrix}\textbf{Dark Knight}\\ \mathbf{3}\end{pmatrix}$$

The above logic implies predictions can be made for **Carmen's** rating of many different **movies** and equally many different **users**:

$$\begin{matrix}\textbf{Carmen}\\ \vdots\end{matrix}\begin{bmatrix}\textbf{Alien} & \textbf{Bug's Life} & \textbf{Cars} & \textbf{Dark Night} & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots\\ 5 & - & 1 & 3 & 2 & - & 5 & 1 & 3 & 1 & 5 & -\\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\end{bmatrix}$$

Matrix Factorization determines the **latent factors**

Determining **latent space**

- How to represent each user in the space
- How to represent each movie in the space

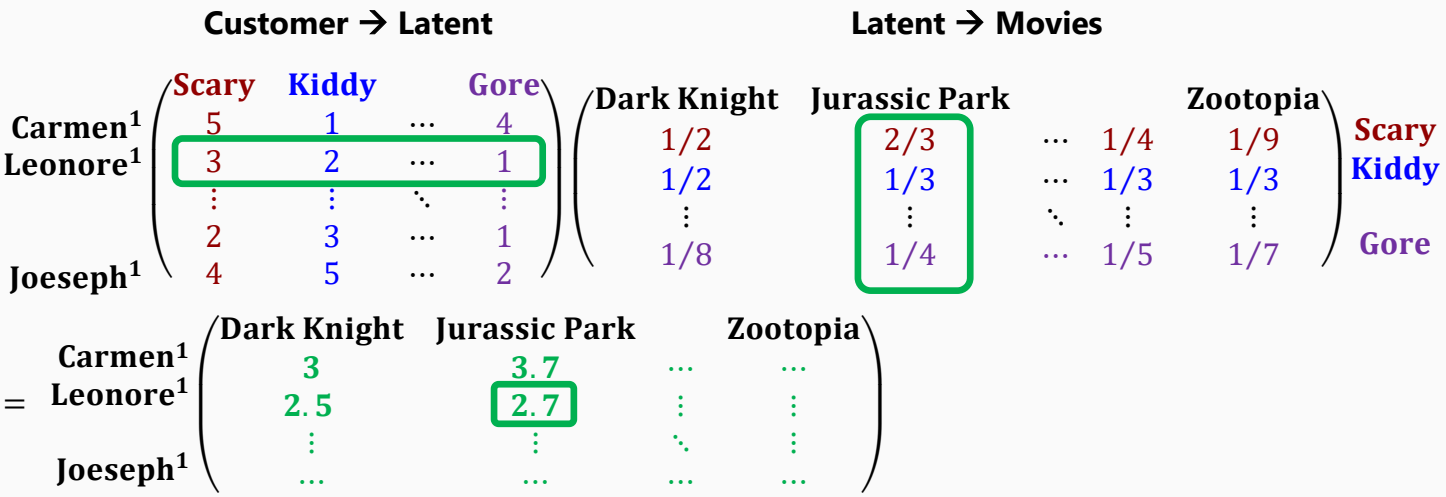
If the latent factors are ideal, few are required to recover ratings for accurate approximations of real ratings, regardless of having thousands of different movies to rate. If a new movie exists that a user has not seen prior, all that matters is the latent state of the movie and the latent state of the user; if these two attributes are known, predications can be accurately assigned:

$$\text{Rating for user } i \text{ on movie } j \approx \hat{R}_{ij} = \sum_{\text{latent factors } \ell} \text{user}_{u,\ell}\text{movie}_{\ell,m}$$

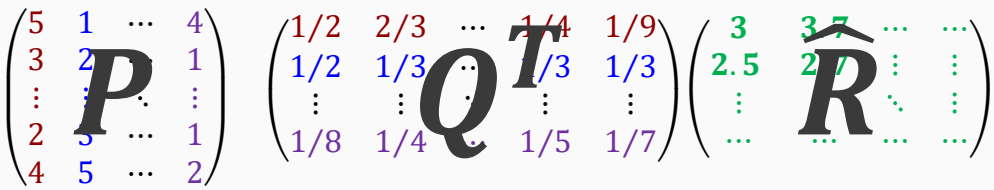
Prediction from **Leonore's** rating of **Dark Knight**: $\mathbf{3}\times\frac{\mathbf{1}}{\mathbf{2}}+2\times\frac{\mathbf{1}}{\mathbf{2}}=\mathbf{2.5}\rightarrow$ **vectors inner product**

$$\begin{matrix}\textbf{Carmen}^{\mathbf{1}}\\ \textbf{Leonore}^{\mathbf{1}}\end{matrix}\begin{pmatrix}\textbf{Scary} & \textbf{Kiddy}\\ 5 & 1\\ 3 & 2\end{pmatrix}\begin{pmatrix}\textbf{Dark Knight}\\ 1/2\\ 1/2\end{pmatrix}\begin{matrix}\textbf{Scary}\\ \textbf{Kiddy}\end{matrix} = \begin{matrix}\textbf{Carmen}^{\mathbf{1}}\\ \textbf{Leonore}^{\mathbf{1}}\end{matrix}\begin{pmatrix}\textbf{Dark Knight}\\ \mathbf{3}\\ \mathbf{2.5}\end{pmatrix}$$

Matrix Factorization applied to multiple movies and users:



The **first matrix** in the above illustration represents the **latent factors of users**. The **second matrix** represents the **latent factors of movies**. The **third matrix** represents the **inner product** of the latter:



The above model: $P \times Q^T = \hat{R}$ with the **goal** of choosing $\boldsymbol{P}$ and $\boldsymbol{Q^T}$ such that $\widehat{\boldsymbol{R}}$ is accurate.

$\widehat{\boldsymbol{R}}$ is accurate when: $\boldsymbol{P}$ and $\boldsymbol{Q^T}$ are chosen to **minimize**:

$$\sum_{\text{users } U,\ \text{movies},m} \left(R_{u,m} - \hat{R}_{u,m}\right)^2 = \sum_{\text{users } U,\ \text{movies},m} \left(\text{error}_{u,m}\right)^2$$

The estimated rating should ideally be as close to the true rating as possible. The squared distance between the true ratings and the estimated ratings are minimized (**Sum of Squares Error**).

The **optimization problem** above is solved through **gradient descent** on the **errors**. The method begins by minimizing $\boldsymbol{P}$, then minimizing $\boldsymbol{Q}$, then reiterating until convergence.

Alternate minimization scheme:

$$\text{error}_{u,m}^2 = \left(R_{u,m} - \hat{R}_{u,m}\right)^2 = \left(r_{ij} - \sum_{\text{latent } \ell=1}^{L} \boldsymbol{p}_{u,\ell}\boldsymbol{q}_{\ell,m}\right)^2$$

$$\frac{d}{d\boldsymbol{p}_{u,\ell}}\text{error}_{u,m}^2 = -2\left(R_{u,m} - \hat{R}_{u,m}\right)\boldsymbol{q}_{\ell,m} = -2\ \text{error}_{u,m}\boldsymbol{q}_{\ell,m}$$

$$\boldsymbol{p}_{u,\ell} \coloneqq \boldsymbol{p}_{u,\ell} - \alpha\frac{d\ \text{error}_{u,m}^2}{d\boldsymbol{p}_{u,\ell}} = \boldsymbol{p}_{u,\ell} + 2\alpha\ \text{error}_{u,m}\boldsymbol{q}_{\ell,m}$$

$$\frac{d}{d\boldsymbol{q}_{u,\ell}}\text{error}_{u,m}^2 = -2\left(R_{u,m} - \hat{R}_{u,m}\right)\boldsymbol{p}_{\ell,m} = -2\ \text{error}_{u,m}\boldsymbol{p}_{\ell,m}$$

$$\boldsymbol{q}_{u,\ell} \coloneqq \boldsymbol{q}_{u,\ell} - \alpha\frac{d\ \text{error}_{u,m}^2}{d\boldsymbol{q}_{u,\ell}} = \boldsymbol{q}_{u,\ell} + 2\alpha\ \text{error}_{u,m}\boldsymbol{p}_{\ell,m}$$

If the learning rate $\boldsymbol{\alpha}$ is set to be too **large**, gradient descent is likely to overshoot the minimum and oscillate back and forth. If the learning rate $\boldsymbol{\alpha}$ is set to be too **small**, gradient descent will be slow to coverage and might fail entirely. **Regularization** can also be added to the above illustration of **gradient descent**, adding an additional factor to the expression.

It is important to note that the above **minimization objective** does not necessarily provide the best possible salutation at a **global minimum**. The objective is not **convex** and thus, **local minimizers** may result when implementing but the technique works well in practice.

Questioning the above method:

Is **Matrix Factorization** illustrated above synonymous to setting the **unknowns** to **0** and performing least squares regression?

$$\begin{array}{c} \\ \textbf{Carmen}^{\mathbf{1}} \\ \textbf{Leonore}^{\mathbf{1}} \\ \\ \textbf{Joeseph}^{\mathbf{1}} \end{array} \begin{array}{c} \widehat{\boldsymbol{R}} \\ \begin{pmatrix} 3 & 3.7 & 2.7 & 2.9 \\ 2.5 & 2.7 & 4.1 & 1.3 \\ 2.1 & 0.9 & 0.5 & 2.8 \\ 4.6 & 2.6 & 4.2 & 0.1 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ \textbf{Carmen}^{\mathbf{1}} \\ \textbf{Leonore}^{\mathbf{1}} \\ \\ \textbf{Joeseph}^{\mathbf{1}} \end{array} \begin{array}{c} \boldsymbol{R} \\ \begin{pmatrix} 3 & 3.7 & ? & 2.9 \\ 2.5 & ? & 4.1 & ? \\ ? & 0.9 & 0.5 & 2.8 \\ ? & 2.6 & 4.2 & ? \end{pmatrix} \end{array}$$

$$\begin{array}{c} \\ \textbf{Carmen}^{\mathbf{1}} \\ \textbf{Leonore}^{\mathbf{1}} \\ \\ \textbf{Joeseph}^{\mathbf{1}} \end{array} \begin{array}{c} \widehat{\boldsymbol{R}} \\ \begin{pmatrix} 3 & 3.7 & 2.7 & 2.9 \\ 2.5 & 2.7 & 4.1 & 1.3 \\ 2.1 & 0.9 & 0.5 & 2.8 \\ 4.6 & 2.6 & 4.2 & 0.1 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ \textbf{Carmen}^{\mathbf{1}} \\ \textbf{Leonore}^{\mathbf{1}} \\ \\ \textbf{Joeseph}^{\mathbf{1}} \end{array} \begin{array}{c} \boldsymbol{R} \\ \begin{pmatrix} 3 & 3.7 & \mathbf{0} & 2.9 \\ 2.5 & \mathbf{0} & 4.1 & ? \\ \mathbf{0} & 0.9 & 0.5 & 2.8 \\ \mathbf{0} & 2.6 & 4.2 & \mathbf{0} \end{pmatrix} \end{array}$$

The answer is **no**; if the **unknowns** are replaced with **zero** in **matrix factorization**, any missing votes will suffer a large penalty which becomes problematic when many votes are missing. Only the errors that can be calculated are considered.