CSC311 Notes

When to use machine learning vs. other computer tools?

- It's hard to manually code a solution to an issue (eg. recognizing vision, speech)
- Program has to adapt to changing environments (eg. detect spam)
- Program should perform better than humans
- Privacy/fairness (eg. ranking search results)

Machine learning vs statistics

- Both find patterns in data, draw on similar math bases, same core algorithms
- Stats about helping scientists/policymakers draw conclusions; ML about algorithms, coding the theoretical
- Stats about interpreting results, math & rigor; ML about predictive performance scalability, autonomous agents

Programs **learn** from experience E with respect to tasks T, performance measures P, when E grows $\Rightarrow$ P improves.

Supervised Learning: Teaching by giving labelled examples of correct behavior.
Reinforcement Learning: Teaching by letting program interact with simulation, maximize some value
Unsupervised Learning: No labelled examples, finds "interesting" data patterns.

Supervised Learning

Parameter: A variable that affects the learning model's output. "Learning" refers to optimizing the parameters.
Parametric Models: Models that "learn" parameters from data and at test-time, refer back to the "learned" data

- eg. linear regression, which learns optimal weights for a linear equation like $y(x) = \alpha x + \beta$
- eg. decision trees, which build a tree based on parameter values for predicting stuff

Non-Parametric Models: Models without parameters. There's no "learning phase"; all work is done at run-time.

- eg. nearest neighbours

Training Set ($\mathcal{D}$): A set of data fed into the program for learning.

- Data consists of examples of correct behavior to emulate: inputs and labels (i.e. correct outputs)
- Labels can also be called targets/responses/outcomes/outputs/classes
- Usually, we collapse any input into a vector of form $x = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$ because it makes computation fast.
  - The vector components are also called features/covariates
  - e.g. Images can be thought of huge matrices of RGB values. We concatenate each row into a super-long vector because we don't need the matrix's extra "spatial" information.
- Regression Algorithm: An algorithm $\mathbb{R}^d \to \mathbb{R}$ that predicts continuous values
- Classification Algorithm: An algorithm $\mathbb{R}^d \to \{\text{discrete outputs}\}$ that predicts a class
- Formally, $\mathcal{D} = \{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_n, t_n)\}$, where $\mathbf{x}_i \in \mathbb{R}^d$ is an input and $t_i \in \mathbb{R}$ is the corresponding label.

Validation Set: A set of data where for any $(\mathbf{x}_i, t_i)$ pair, each t_i is compared to $y(\mathbf{x}_i)$ using the same model on different hyperparameter values. Used to find the optimal hyperparameter values.

- Hyperparameter: A parameter we set that affects learning, in contrast to parameters an algorithm derives.
- Grid Search: Exhaustively searching for optimal value through a manually-chosen subset of hyperparameters
- Random Search: Testing random hyperparameter configurations using a probability distribution

Test Set: A set of data where for any $(\mathbf{x}_i, t_i)$ pair, t_i is compared to $y(\mathbf{x}_i)$, the model's prediction

- Generalization Error: How often the program correctly predicts data it hasn't seen before (ie. **generalizes**).
- Test sets are used on the "final" model, unlike the validation set, which tests variations of the same model

Nearest Neighbours

Argmin ($\underset{\boldsymbol{x}\in\boldsymbol{\mathcal{D}}}{\textbf{argmin}}\, \boldsymbol{f(x)}$): The argument of $\min_{x\in\mathcal{D}} f(x)$; $f\left(\underset{x\in\mathcal{D}}{\operatorname{argmin}} f(x)\right) = \min_{x\in\mathcal{D}} f(x)$. Same logic with argmax.

Indicator/Identity Function: The function $\mathbb{I}(P) = \begin{cases}1 & P \text{ is true}\\ 0 & P \text{ is false}\end{cases}$.

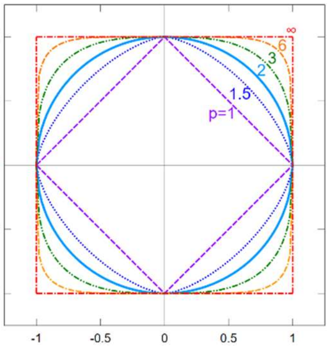
Metric Space: A set of points with some notion of distance between them defined by a distance function. A generalization of the idea of distance (MAT327).

***L^p* Space:** In functional analysis, a metric space defined by distance function

$$\|\mathbf{x}\|_p = \sqrt[p]{\sum_{i=1}^{d} |x_i|^p} = \sqrt[p]{|x_1|^p + |x_2|^p + \cdots + |x_n|^p}$$

where $\mathbf{x} \in \mathbb{R}^d$. This is also called the ***p*-norm**.

- On the right is the unit circle in $\mathbb{R}^2$ under different p-values.
- The **2-norm** is the Euclidean norm, how we've understood distance so far: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$



Nearest Neighbours (NN): A non-parametric algorithm $y(\mathbf{x})$ that, given a query point $\mathbf{x}$, returns the label of the training set input "nearest" to $\mathbf{x}$.

- Since we assume inputs are vectors, use Euclidean distance: $\|\mathbf{u} - \mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2}$

  Not always the *best* distance measure. Higher/lower p-values make larger/smaller errors more significant.

Algorithm

1) Recall the training set $\mathcal{D} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_n, t_n)\}$. Find the nearest $(\mathbf{x}^*, t^*)$ to $\mathbf{x}$ with the formula

$$\mathbf{x}^* = \underset{\mathbf{x}_i \in \{\mathbf{x}_1,\ldots,\mathbf{x}_n\}}{\operatorname{argmin}} (\text{distance}(\mathbf{x}_i, \mathbf{x})) = \underset{\mathbf{x}_i \in \{\mathbf{x}_1,\ldots,\mathbf{x}_n\}}{\operatorname{argmin}} \|\mathbf{x}_i - \mathbf{x}\|$$

   In other words, $\|\mathbf{x}^* - \mathbf{x}\| = \min_{\mathbf{x}_i \in \{\mathbf{x}_1,\ldots,\mathbf{x}_n\}} \|\mathbf{x}_i - \mathbf{x}\|$. If there are multiple minimums, pick any one of them.

2) Output $y(\mathbf{x}^*) = t^*$

Practically, compute $\|\mathbf{x}_i - \mathbf{x}\|^2 = \sum_{i=1}^{d} (u_i - v_i)^2$ since it saves computing $\sqrt{\ldots}$ and squaring preserves argmins.

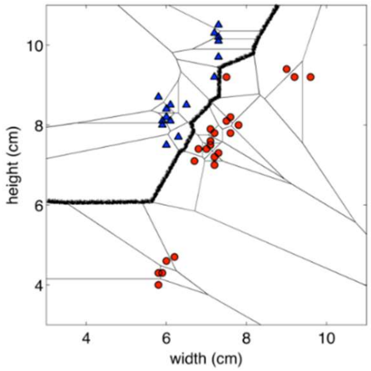
- Algorithm is the same between regression/classification; only difference is t^* is a scalar/discrete value
- Despite its simplicity, NN is a competitive learning algorithm
- NN is very effective if we know *what* we should measure distances of, and *how* we measure distances

Input Space: Set of all possible inputs (vectors) that the learning model accepts; the domain of the learning model's algorithm, in $\mathbb{R}^d$

Decision Boundaries: The dividing lines in input space between inputs that output different categories.

Voronoi Diagram: Given points $\{p_1, \ldots, p_n\}$ in $\mathbb{R}^d$, a partition of input space into n tiles where each tile contains the closest points to some p_i.

- Helps visualize how NN assigns parts of input space to categories
- Plot $\mathbf{x}$, see the tile $\mathbf{x}$ falls in, find the other point $\mathbf{x}^*$ in the tile, return t^*



K-Nearest Neighbours (KNN): An extension of NN using odd hyperparameter $k \in \mathbb{N}$. Finds the k nearest neighbours to X and returns a normal/weighted average (regression) or majority classification (classification).

Algorithm

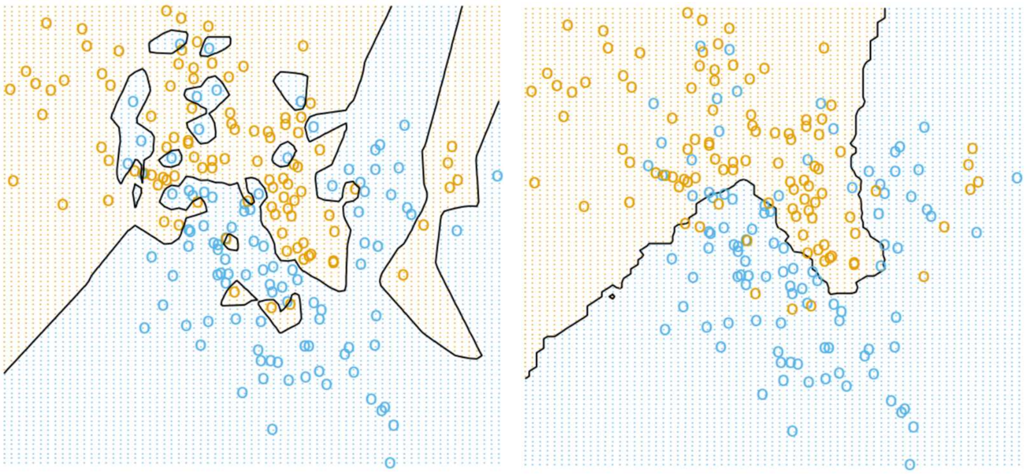
1) Find k examples of $(\mathbf{x}^*, t^*)$ with the closest $\mathbf{x}^*$ values to $\mathbf{x}$ using the NN algorithm.
2) Output the mean/mode of $\{t_1^*, \dots, t_k^*\}$ with the formulae

Regression (Mean)

$$y(\mathbf{x}) = \frac{1}{k}\sum_{i=1}^{k} t_i^*$$

Classification (Mode)

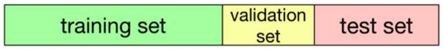
$$y(\vec{\mathbf{x}}) = \underset{t_i^* \in \{t_1^*, \dots, t_k^*\}}{\operatorname{argmax}} \sum_{i'=1}^{k} \mathbb{I}(t_i^* = t_{i'}^*)$$

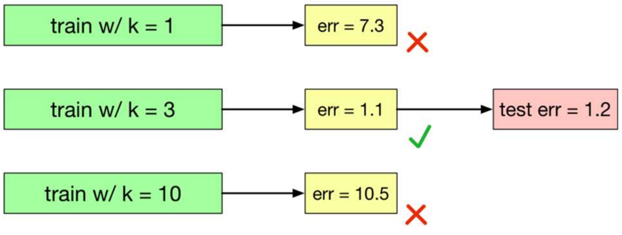


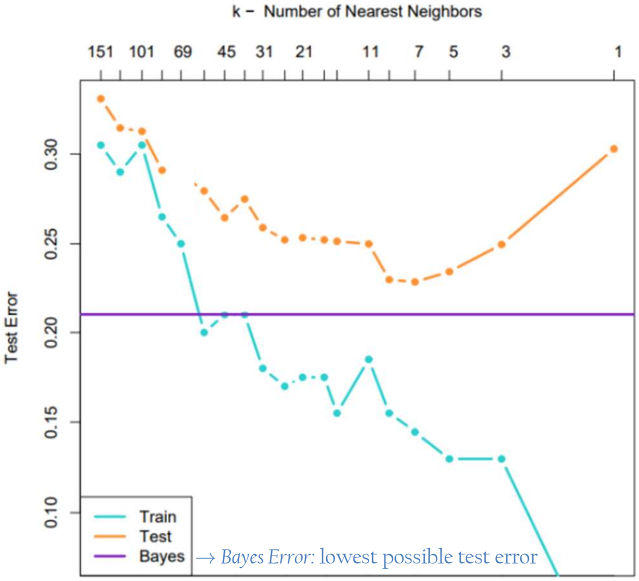
Small k	*Balanced k*	*Large k*
• Overfitting: Too sensitive to noise or mislabelled data, complex (“class noise”) • Captures fine-grained patterns • If $k = 1$, training set error is 0	• Nice theoretical properties if $\lim_{n\to\infty} k = \infty$ and $\lim_{n\to\infty} \frac{k}{n} = 0$ with $k = f(n)$ for some function f • Generally, choose $f(n) < \sqrt{n}$ • Can be chosen by validation sets	• Underfitting: Simple, can’t show important regularities • Stable predictions by averaging examples • If $k = n$, KNN gives the same output for any input

Validation, Testing, and KNN

- Use a validation set to tune for k, then a testing set at the end to test everything
- As k grows, training error increases, but this is fine (don’t expect 100% accuracy)
- What matters is performing just as well on training and testing sets; generalizing







→ *Bayes Error:* lowest possible test error

Pitfalls of Nearest Neighbours

Curse of Dimensionality: Various phenomena with data in higher dimensions, occurring usually because *volume of space increases very fast* with increasing dimensions

- e.g. Sample points from n-dimensional sphere. As n increases, distance between points get **farther** and **more uniform** *(provable using expected value and covariance)*
- Thus, common distances become less meaningful; using Euclidean norm may become problematic.
- Luckily, some data has low **intrinsic dimension:** it lies on or near a low-dimensional manifold, so the volume issues are less pronounced
    - **Manifold:** Informally, a lower-dimensional surface/object in a higher dimension

Normalization: Issues regarding units – higher ranges of absolute numbers in a vector contributes more to distance than smaller ranges of numbers.

- e.g. In a dataset where $\mathbf{x} = \begin{bmatrix} \text{age} \\ \text{income} \end{bmatrix}$, age ranges from 0 to 100, income ranges from 0 to 500,000. Income will influence distance much more than age.
- **Normalize** each vector by setting the boundaries to 0 and 1 by applying this on each value:
$$x_i \leftarrow \frac{x_i - \mu}{\sigma}$$
where $\mu =$ mean of all x_i, $\sigma =$ standard deviation of all x_i.
- However, there are also cases where we don't want to normalize!

Computational Cost: In machine learning, which requires large amounts of data, there are very high standards for efficient running-times of algorithms/memory storage

- Naïve implementations of NN have running-time $\mathcal{O}(nd)$, with n data points, d dimensions, which then must be sorted, which is usually $\mathcal{O}(n \log n)$. This algorithm is run for each query, too, which is expensive.
- A lot of work has been done to make NN more efficient.

Decision Trees

Entropy: In information theory, a number describing information/surprise/uncertainty in a random variable (RV).

- Symbol is H, capital letter of *eta* (η)
- Unit is a **bit**, because it's commonly used in computers storing/compressing/encrypting information
- *High Entropy:* Variable distribution is uniform; less predictable; more "info"
    - e.g. A fair coin flip has entropy of 1 bit
- *Low Entropy:* Variable distribution is concentrated, more predictable; less "info"; 0 means 100% predictability
    - e.g. A biased coin flip of 8/9 heads, 1/9 tails has an entropy of ~½ bits
- Most commonly applies to discrete RVs, but can be extended to continuous RVs in *differential entropy*

$$H(X) = -\sum_{x\in X} p_X(x)\log_2 p_X(x) \qquad H[Y|X=x] = -\sum_{y\in Y} p_{Y|X}(y|x)\log_2 p_{Y|X}(y|x)$$

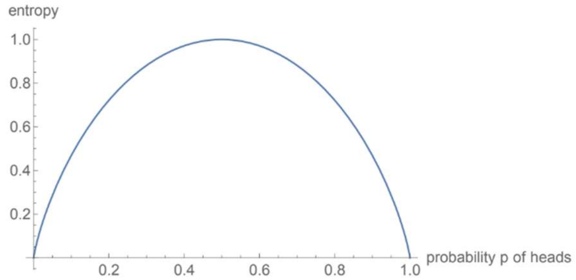
$$H(X,Y) = -\sum_{x\in X}\sum_{y\in Y} p_{X,Y}(x,y)\log_2 p_{X,Y}(x,y) \qquad \mathbb{E}[H[Y|X]] = -\sum_{x\in X}\sum_{y\in Y} p_{X,Y}(x,y)\log_2 p_{Y|X}(y|x)$$

$H(X) \geq 0$	$H(X\|X) = 0$	$H(X\|Y) = H(X)$ for independent X, Y	$H(X,Y) = H(X\|Y) + H(Y)$	$H(X\|Y) \leq H(X)$

Information Gain ($IG(Y|X)$): How much more certain we are about Y's result from knowing X.

$$IG(Y|X) = H(Y) - H(Y|X)$$

- AKA how informative X is to Y, or the loss of entropy/uncertainty in Y due to knowing X
- *X is completely informative:* $IG(Y|X) = H(Y)$
- *X is completely uninformative:* $IG(Y|X) = 0$



Greedy Heuristic: A type of approach/strategy that involves always choosing the most locally-optimal choice (ie. what looks best in the moment)

Loss ($\mathcal{L}$): Refers to a loss function, that returns a non-negative scalar quantifying "error levels". Should be minimized

- Objective Function: In optimization, a function that needs to be minimized.
- Cost Function: Usually averaged loss over $\mathcal{D}$ plus the regularizer function (see later). No strict definition, often interchangeable with loss. (sometimes, omit $\frac{1}{n}$ as it's a scalar multiple)

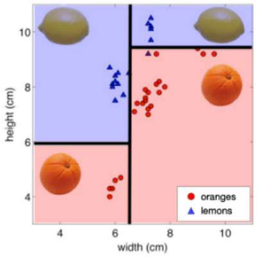
$$\mathcal{J}(\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y(\mathbf{x}_i), t_i)$$

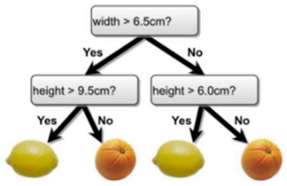
- Mean Squared Error: Average squared difference between a model output $y(\mathbf{x})$ and training target t.

$$\mathcal{J}(\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(y(\mathbf{x}_i) - t_i)^2\,, \qquad \mathcal{L}(y(\mathbf{x}), t) = (y(\mathbf{x}) - t)^2$$

Decision Tree: A parametric model based on a tree structure where features are split based on their value.

- *Decision boundaries* are axis-aligned lines
- *Internal nodes* are booleans about feature values
- *Leaf nodes* are outputs (ie. predictions)
- *Hyperparameters* are tree height, branches per node, number of features, nodes, information gain threshold, etc.





Algorithm – Learning

1) Start with no nodes.
2) Use a **greedy heuristic** to pick features and a split that minimizes **loss**.
   a. *Regression Tree:* Minimize **squared error** in the divided regions per split
   b. *Classification Tree:* Maximize **information gain** per split

   For splits with the same loss, the program picks splits that maximize the decision boundary's **margins** – i.e. distance to points on both sides
3) Stop subdividing if all points in regions have same classification, no points left, or hyperparameters say so

Algorithm – Predicting

4) Let R be a region of input space corresponding to an output. Find the R associated with new point $\mathbf{x}$. Let $\{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_k, t_k)\}$ be the training points where $x_i \in R$
   a. *Regression Tree:* Since the output is continuous, we output the mean of all t_i
   b. *Classification Tree:* Since the output is discrete, we output the mode of all t_i

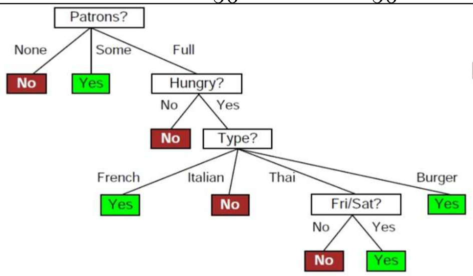
eg. A decision tree classifies $*$ and $\star$. What is the information gain of the split as shown on the right?

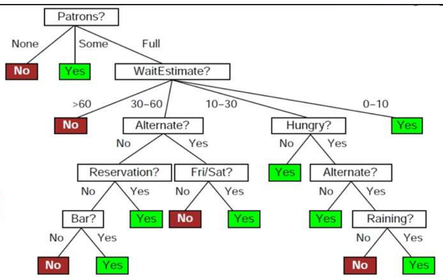
Original:	$13*, 56\star$	Split:	*Left:* $3*,\ 22\star$ *Right:* $10*,\ 21\star$

$$\begin{aligned} H(Y) &= -\sum_{y\in Y} p_Y(y)\log_2 p_Y(y) \\ &= -p_Y(*)\log_2 p_Y(*) - p_Y(\star)\log_2 p_Y(\star) \\ &= -\frac{13}{56}\log_2\frac{13}{56} - \frac{43}{56}\log_2\frac{43}{56} \\ &\approx \boxed{0.7817} \end{aligned}$$

$$\begin{aligned} H(Y|X=\text{left}) &= -\sum_{y\in Y} p_{Y|X}(y|\text{left})\log_2 p_{Y|X}(y|\text{left}) \\ &= -\frac{3}{25}\log_2\frac{3}{25} - \frac{22}{25}\log_2\frac{22}{25} \approx \boxed{0.5294} \\ H(Y|X=\text{right}) &= -\sum_{y\in Y} p_{Y|X}(y|\text{right})\log_2 p_{Y|X}(y|\text{right}) \\ &= -\frac{10}{31}\log_2\frac{10}{31} - \frac{21}{31}\log_2\frac{21}{31} \approx \boxed{0.9072} \end{aligned}$$

$$\begin{aligned} \therefore IG(Y|X) &= H(Y) - H(Y|X) \\ &= H(Y) - P(X=\text{left})H(Y|X=\text{left}) - P(X=\text{right})H(Y|X=\text{right}) \\ &\approx 0.7817 - \frac{25}{56}(0.5294) - \frac{31}{56}(0.9072) \approx \boxed{0.0432} \end{aligned}$$





Small Tree	*Balanced Tree*	*Big Tree*
• Underfitting: Simple, can't handle important distinctions in data	• "Occam's Razor": Find the simplest hypothesis that fits observations • Ideally, smaller trees with informative nodes near the root	• Overfitting: Complex, incorporates meaningless noise into decisions • Difficult for humans to interpret • Computationally inefficient (redundant, nonsense attributes)

Pitfalls of Decision Trees

- As you descend paths in a decision tree, training set data is further subdivided and exponentially decreases
- Greedy algorithms do not necessarily yield the global optimum

KNN	*Decision Trees*
➢ One hyperparameter ➢ Can choose what to measure distance of, how to measure distance	➢ Faster at test-time ➢ Easier to interpret for a non-professional ➢ Easier for discrete features, missing values, poorly-scaled data

Bias-Variance Decomposition

Generalization: Ability of models to predict from unseen examples (in the same distribution as training data)

- Models can overfit and underfit simultaneously. Simple models underfit more, complex models overfit more.

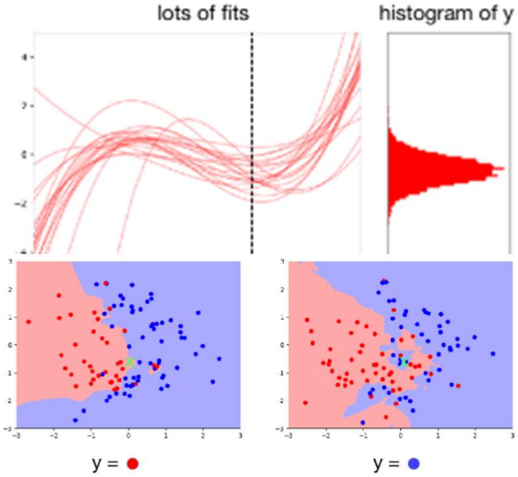
Bias-Variance Decomposition (BVD): A way to quantify underfitting and overfitting

- **Data Generating Distribution ($p_{X,T}(\mathbf{x}, t)$):** A hypothetical probability distribution function from which we assume all training/test data $\mathcal{D}$ originates. We assume data is **independent and identically distributed (i.i.d.)**.

Computation

1) Create a training set $\mathcal{D}$ using samples from $p_{X,T}(\mathbf{x}, t)$
2) Pick a fixed query point $\mathbf{x}$ in $\mathcal{D}$, find output $Y = y(\mathbf{x})$
3) Sample a T value from $p_{T|X}(t|\mathbf{x})$, the hypothetical probability distribution of outputs, given an input
4) Compute loss using $\mathcal{L}(Y, T)$
5) By repeating steps 2-5 enough times, we find the expected loss, $\mathbb{E}[\mathcal{L}(Y,T)|X]$. We want to minimize this value to reduce bias and variance in our model.

Y and T are both random variables (RV), but when we find $\mathbb{E}$, we treat one of Y and T as a RV at a time and the other as a constant.



BVD of Mean Squared Error

Recall, MSE's loss function is $\mathcal{L}(Y, T) = (Y - T)^2$

Treating T as RV, Y as arbitrary output

$$\begin{aligned}\mathbb{E}[\mathcal{L}(Y,T)|X] &= \mathbb{E}[(Y-T)^2|X] \\ &= \mathbb{E}[Y^2 - 2YT + T^2|X] \\ &= \mathbb{E}[Y^2] - 2Y\mathbb{E}[T|X] + \mathbb{E}[T^2|X] \\ &= \mathbb{E}[Y^2] - 2Y\mathbb{E}[T|X] + \mathbb{E}[T|X]^2 + \mathrm{Var}[T|X] \\ &= \mathbb{E}[(Y - \mathbb{E}[T|X])^2] + \mathrm{Var}[T|X]\end{aligned}$$

To minimize $(Y - \mathbb{E}[T|X])^2$, set $Y = \mathbb{E}[T|X] = Y^*$; predict the expected output of a sample, the Bayes optimal decision.

The second term is Bayes Error, the inherent unpredictability/ noise of T. This is the best error algorithms can hope to get.

Treating Y as RV, T as arbitrary output

$$\begin{aligned}&\mathbb{E}[\mathcal{L}(Y,T)|X] \\ &= \mathbb{E}[(Y - \mathbb{E}[T|X])^2] + \mathrm{Var}[T|X] \ \text{(from result on the left)} \\ &= \mathbb{E}[(Y - Y^*)^2|X] + \mathrm{Var}[T|X] \\ &= Y^{*2} - 2Y^*\mathbb{E}[Y|X] + \mathbb{E}[Y^2|X] + \mathrm{Var}[T|X] \\ &= Y^{*2} - 2Y^*\mathbb{E}[Y|X] + \mathbb{E}[Y|X]^2 + \mathrm{Var}[Y|X] + \mathrm{Var}[T|X] \\ &= \underbrace{(Y^* - E[Y|X])^2}_{\text{bias}} + \underbrace{\mathrm{Var}[Y|X]}_{\text{variance}} + \underbrace{\mathrm{Var}[T|X]}_{\text{Bayes error}}\end{aligned}$$

Bayes Optimal: An algorithm that makes Bayes optimal decisions, ones that minimize expected loss; any error in this algorithm is due to Bayes error and it cannot be further improved.

Bias-Variance Trade-off: Idea that by tuning parameters/hyperparameters, one can increase bias but at the cost of decreasing variance and vice versa.

Bias: Model accuracy, related to underfitting	Model can't learn all patterns in data Model has enough data for stable estimates	High bias Low variance
Variance: Model variability, related to overfitting	Model learns patterns in data Model incorporates noise/quirks of sampled data	Low bias High variance

Bagging/Bootstrap Aggregation

Ensemble Method: Using the majority prediction of multiple learning algorithms to get a prediction

- Small ensembles can be better than a single good model

Bagging/Bootstrap Aggregation: A type of ensemble method where n training sets are "sampled" from $p_{X,T}(\mathbf{x}, t)$, a model is trained on each set, and each prediction $y_i(\mathbf{x})$ (for the model on the i-th set) is averaged/compared.

Regression	*Classification (Binary)*
$y(\mathrm{x}) = \frac{1}{n}\sum_{i=1}^{n} y_i(\mathbf{x})$	$y(\mathrm{x}) = \mathbb{I}\left(\frac{1}{n}\sum_{i=1}^{n} y_i(\mathrm{x}) > 0.5\right)$

- In reality, $p_{X,T}(\mathbf{x}, t)$ is often expensive/impossible to sample from, and we only have one training set $\mathcal{D}$.
- Thus, assume data in $\mathcal{D}$ is independent, use $p_{\mathcal{D}}(\mathbf{x}, t)$ as a proxy for $p_{X,T}(\mathbf{x}, t)$, create n datasets using resamples/bootstrap samples drawn with replacement from $p_{\mathcal{D}}(\mathbf{x}, t)$.
- As $|\mathcal{D}| \to \infty, p_{\mathcal{D}}(\mathbf{x}, t) \to p_{X,T}(\mathbf{x}, t)$

Bias is the same (as predictions are same)	Variance is reduced (as we average independent predictions)
$\mathbb{E}[y(\mathbf{x})] = \mathbb{E}[y_i(\mathbf{x})]$	$\mathrm{Var}[y(\mathbf{x})] = \mathrm{Var}\left[\frac{1}{n}\sum_{i=1}^{n} y_i(\mathbf{x})\right] = \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}[y_i(\mathbf{x})] = \frac{1}{n}\mathrm{Var}[y_i(\mathbf{x})]$

Pitfalls of Bagging:

- Resampled data might not be independent! We thus try reducing correlation between resampled data sets by introducing variability
  - eg. Average over multiple algorithms, multiple configurations of an algorithm
  - eg. In **random forests** (ie. ensemble of decision trees), choose a subset of features for each tree, have that tree only split according to those features
- Does not reduce bias (at least in the case of squared error)
- Equal weights to ensemble members may not be ideal – weighted ensembling is often better for radically-different members

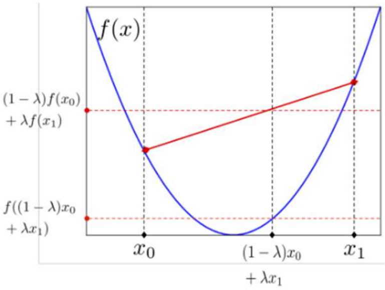
Linear Regression

Partial Derivative: Of function $f\colon \mathbb{R}^n \to \mathbb{R}$, the derivative with respect to (w.r.t.) a variable

- For instance, in $\frac{\partial f}{\partial x} = \frac{\partial}{\partial x} f$, differentiate f w.r.t. x, treating all other variables as constants.
- As a notational shorthand, $\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial (x_1, \ldots, x_n)} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_n} \end{bmatrix}$
- eg. $\frac{\partial}{\partial x}(x^2 y + x y^2 + y) = 2yx + y^2$

Gradient (∇): Of function $f\colon \mathbb{R}^n \to \mathbb{R}$, vector $\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right) \in \mathbb{R}^n$.

- eg. $\nabla(x^2 y + x y^2 + y) = (2yx + y^2, x^2 + 2xy + 1)$
- If $\frac{\partial}{\partial x_i} f(x) = \alpha_i$, then $\nabla f = (\alpha_1, \ldots, \alpha_n) = \boldsymbol{\alpha}$
- If A is a matrix and $\mathbf{x}$, b are vectors, then $\frac{\partial}{\partial \mathbf{x}}(A\mathbf{x} + b) = A$



Convex Function: Functions where $f'' > 0$ or $f'' < 0$. AKA the line segment between any x_0, x_1 is above/below $f(x)$,

$$f\big((1-\lambda)x_0 + \lambda x_1\big) \leq (1-\lambda) f(x_0) + \lambda f(x_1)$$

- Recall a critical point at $f' = 0$ is either a local maximum or local minimum
- The local maximum/minimum of a convex function is the global maximum/minimum

Vectorization: The practice of turning loops in code to vector/matrix operations.

- *Simpler & readable* code – no dummy variables/indices
- *Computationally faster* – linear algebra libraries are highly-optimized, matrix multiplication is very fast, parallelizable and good for GPU, cut down on slowness of Python interpreter overhead
  - Why use Python vs. C/C++ if performance matters? Because Python's quicker to write code in, it's better for prototyping, quickly experimenting, & communicating algorithms to others. Most code is done in libraries compiled in other languages too. C/C++ is good for optimizing algorithms.
- Most "math" in of a machine learning algorithm is just matrices/vectors anyways

Linear Regression: A parametric regression model that finds the best linear relationship between two variables

- Involves many *modular components* very typical of machine learning:

Model	describes variable relationships
Loss function	quantifies how bad a fit to data is
Optimization Algorithm	minimizes the loss function
Regularizer	quantifies our preference for candidate models *(optional)*

- Goal is to learn a function y such that $\forall(\mathbf{x}, t) \in \mathcal{D}, t \approx y(\mathbf{x})$

Model: Let $\mathbf{x} \in \mathbb{R}^D$ be input, $y(\mathbf{x})$ be output, $\mathbf{w} \in \mathbb{R}^D$ be weight, $b \in \mathbb{R}$ be bias/intercept. Parameters are b and $\mathbf{w}$.

$$y(\mathbf{x}) = b + \sum_{d=1}^{D}(w_d x_d) = b + \mathbf{w} \cdot \mathbf{x}$$

Vectorization in Linear Regression

For N training set vectors of dimension D.

Store all training set vectors into a **design matrix** of size $\mathbb{R}^{N \times D}$	$X = \begin{bmatrix} \overrightarrow{x_1}^{\mathrm{T}} \\ \vdots \\ \overrightarrow{x_N}^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} (x_1)_1 & \cdots & (x_1)_D \\ \vdots & \ddots & \vdots \\ (x_N)_1 & \cdots & (x_N)_D \end{bmatrix}$	$X\mathbf{w} + \mathbf{b} = \begin{bmatrix} (x_1)_1 & \cdots & (x_1)_D \\ \vdots & \ddots & \vdots \\ (x_N)_1 & \cdots & (x_N)_D \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_D \end{bmatrix} + \begin{bmatrix} b \\ \vdots \\ b \end{bmatrix}$
Store all outputs into a **target vector** of size $\mathbb{R}^N$	$\mathbf{t} = (t_1, \ldots, t_N)$	$= \begin{bmatrix} \mathbf{w} \cdot \mathbf{x}_1 + b \\ \vdots \\ \mathbf{w} \cdot \mathbf{x}_N + b \end{bmatrix}$
Store **weights** for each x_d as a vector of size $\mathbb{R}^D$	$\mathbf{w} = (w_1, \ldots, w_D)$	$= \begin{bmatrix} y(\mathbf{x}_1) \\ \vdots \\ y(\mathbf{x}_N) \end{bmatrix} \approx \mathbf{t} = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix}$
Store **bias** as vector in $\mathbb{R}^N$	$\mathbf{b} = (b, \ldots, b)$	

We can simplify $X\mathbf{w}+\mathbf{b}$ into $X\mathbf{w}$ by cramming value b into X and $\mathbf{w}$. We could also just label b as w_0.

$$X=\begin{bmatrix}1 & \mathbf{x}_1^{\mathrm{T}}\\ \vdots & \vdots\\ 1 & \mathbf{x}_N^{\mathrm{T}}\end{bmatrix}=\begin{bmatrix}1 & (x_1)_1 & \cdots & (x_1)_D\\ \vdots & \vdots & \ddots & \vdots\\ 1 & (x_N)_1 & \cdots & (x_N)_D\end{bmatrix}, \mathbf{w}=\begin{bmatrix}b\\ w_1\\ \vdots\\ w_D\end{bmatrix}, \text{then } X\mathbf{w}=\begin{bmatrix}1 & (x_1)_1 & \cdots & (x_1)_D\\ \vdots & \vdots & \ddots & \vdots\\ 1 & (x_N)_1 & \cdots & (x_N)_D\end{bmatrix}\begin{bmatrix}b\\ w_1\\ \vdots\\ w_D\end{bmatrix}=\begin{bmatrix}\mathbf{w}\cdot\mathbf{x}_1+b\\ \vdots\\ \mathbf{w}\cdot\mathbf{x}_N+b\end{bmatrix}$$

<u>Loss/Cost Function:</u> Use a modified squared error loss function with $\frac{1}{2}$ to simplify derivative calculations *(see below)*.

$$\begin{aligned}\mathcal{L}(y(\mathbf{x}),t)&=\frac{1}{2}(y(\mathbf{x})-t)^2\\ \mathcal{J}(y(\mathbf{x}),t)&=\frac{1}{N}\sum_{n=1}^{N}\mathcal{L}(y(\mathbf{x}_n),t_n)\\ &=\frac{1}{2N}\sum_{n=1}^{N}(y(\mathbf{x}_n)-t_n)^2\\ &=\frac{1}{2N}\left[(y(\mathbf{x}_1)-t_1)^2+\cdots+(y(\mathbf{x}_N)-t_N)^2\right]\end{aligned}$$

$$\begin{aligned}&=\frac{1}{2N}\left\|\begin{bmatrix}y(\mathbf{x}_1)\\ \vdots\\ y(\mathbf{x}_N)\end{bmatrix}-\begin{bmatrix}t_1\\ \vdots\\ t_N\end{bmatrix}\right\|^2\\ &=\frac{1}{2N}\left\|\begin{bmatrix}\mathbf{w}\cdot\mathbf{x}_1+b\\ \vdots\\ \mathbf{w}\cdot\mathbf{x}_N+b\end{bmatrix}-\mathbf{t}\right\|^2\\ &=\frac{1}{2N}\|X\mathbf{w}-\mathbf{t}\|^2\end{aligned}$$

We then write $\mathcal{J}$ as a function of its weights and bias, $\mathcal{J}(\mathbf{w})$, to make it suitable for optimizing our parameter, $\mathbf{w}$.

<u>Optimization Algorithm:</u> Find $\operatorname{argmin}_{\mathbf{w}\in\mathbb{R}^D}\mathcal{J}(\mathbf{w})$, the $\mathbf{w}$ that minimizes a non-negative $\mathcal{J}(\mathbf{w})$.

- *Direct Solution:* Find critical point! Set $\nabla\mathcal{J}(\mathbf{w})=0$, solve for $\mathbf{w}$ *(possible in linear regression, rarely possible elsewhere)*

$$\nabla\mathcal{J}(\mathbf{w})=\frac{1}{N}(X^{\mathrm{T}}X\mathbf{w}-X^{\mathrm{T}}\mathbf{t})=0$$
$$\therefore \mathbf{w}=(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{t}$$

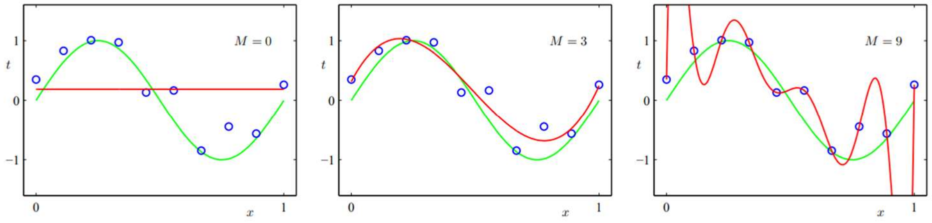
- *Iterative Solution:* Repeatedly apply an update rule that gradually takes us closer to the solution
  - **Iterative Algorithm:** An algorithm that is repeatedly updates a variable until it meets criteria
  - **(Batch) Gradient Descent:** Iterative algorithm that adjusts weights in direction of steepest descent
    - **Direction of Steepest Ascent:** The direction of ∇f
    - **Direction of Steepest Descent:** The direction of $-\nabla f$

$$\begin{aligned}\mathbf{w}&\leftarrow\mathbf{w}-\alpha\nabla\mathcal{J}(\mathbf{w})\\ &=\mathbf{w}-\frac{\alpha}{N}\sum_{n=1}^{N}(y(\mathbf{x}_n)-\mathbf{t}_n)\,\mathbf{x}_n\end{aligned}$$

      where $\alpha=$ step size/learning rate, a hyperparameter.
      - Values of α are usually very small (eg. 0.01)
      - Need smaller α for total loss (N times as small) vs. average loss.
    - Use gradient descent as it's versatile, often easier than direct solving, computationally efficient
      - eg. In linear regression, for $\mathbf{w},\mathbf{x}\in\mathbb{R}^D$ and N samples of $\mathbf{x}$, the direct solution of $\mathbf{w}=(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{t}$ requires matrix inversion, which is $\mathcal{O}(D^3)$
      - eg. Gradient descent cost is $\mathcal{O}(ND)$ and can be even less.

Feature Mapping: Using linear regression on a non-linear relationship by transforming input space using feature mapping $\psi:\mathbb{R}^{d_1}\to\mathbb{R}^{d_2}$ into something linear, comparing transformed inputs with outputs.

- **Polynomial Feature Mapping:** Fitting data with $y(x)=\sum_{n=0}^{N}w_nx^n=w_0+w_1x+w_2x^2+\cdots+w_Nx^N$
  - Here, we have $\psi:\mathbb{R}\to\mathbb{R}^N$ by $\psi(x)=(x^0,\dots,x^N)$, so $y(\mathbf{x})=\mathbf{w}\cdot\psi(\mathbf{x})$.
  - N is a hyperparameter. $N=1$ is just regular linear regression.



Optimization – Derivation of direct solution, linear regression

Since $\nabla \mathcal{J}(\mathbf{w}) = \left(\frac{\partial}{\partial w_1}\mathcal{J}(\mathbf{w}), \dots, \frac{\partial}{\partial w_D}\mathcal{J}(\mathbf{w})\right)$, we will consider the derivative w.r.t. a component.

$$\begin{aligned}
\frac{\partial}{\partial w_d}\mathcal{J}(\mathbf{w}) &= \frac{\partial}{\partial w_d}\left(\frac{1}{2N}\|X\mathbf{w}-\mathbf{t}\|^2\right) \\
&= \frac{1}{2N}\frac{\partial}{\partial w_d}\left(\left\|\begin{bmatrix}\mathbf{w}\cdot\mathbf{x}_1+b \\ \vdots \\ \mathbf{w}\cdot\mathbf{x}_N+b\end{bmatrix} - \begin{bmatrix} t_1 \\ \vdots \\ t_N\end{bmatrix}\right\|^2\right) \\
&= \frac{1}{2N}\frac{\partial}{\partial w_d}\left(\sqrt{\sum_{n=1}^{N}[(\mathbf{w}\cdot\mathbf{x}_n+b)-t_n]^2}\right)^2 \\
&= \frac{1}{2N}\frac{\partial}{\partial w_d}\sum_{n=1}^{N}[(\mathbf{w}\cdot\mathbf{x}_n+b)-t_n]^2 \\
&= \frac{1}{2N}\sum_{n=1}^{N}2[(\mathbf{w}\cdot\mathbf{x}_n+b)-t_n]\frac{\partial}{\partial w_d}[\mathbf{w}\cdot\mathbf{x}_n+b-t_n] \quad \text{(chain rule works the same)} \\
&= \frac{1}{N}\sum_{n=1}^{N}[(\mathbf{w}\cdot\mathbf{x}_n+b)-t_n]\frac{\partial}{\partial w_d}[(w_1(x_n)_1+\cdots+w_d(x_n)_d+\cdots+w_D(x_n)_D)+b-t_n] \\
&= \frac{1}{N}\sum_{n=1}^{N}[(\mathbf{w}\cdot\mathbf{x}_n+b)-t_n](x_n)_d \quad (= \text{iterative solution}) \\
&= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)(x_n)_d - \frac{1}{N}\sum_{n=1}^{N}t_n(x_n)_d
\end{aligned}$$

$$\therefore \nabla\mathcal{J}(\mathbf{w}) = \begin{bmatrix}\frac{\partial}{\partial w_1}\mathcal{J}(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial w_D}\mathcal{J}(\mathbf{w})\end{bmatrix} = \begin{bmatrix}\frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)(x_n)_1 \\ \vdots \\ \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)(x_n)_D\end{bmatrix} - \begin{bmatrix}\frac{1}{N}\sum_{n=1}^{N}t_n(x_n)_1 \\ \vdots \\ \frac{1}{N}\sum_{n=1}^{N}t_n(x_n)_D\end{bmatrix} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)\overrightarrow{x_n} - \frac{1}{N}\sum_{n=1}^{N}t_n\overrightarrow{x_n}$$

We then vectorize by turning all summations into matrices. Make sure to check dimensions for matrix multiplication!

$$\begin{aligned}
\frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)\overrightarrow{x_n} &= \frac{1}{N}\begin{bmatrix}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)(x_n)_1 \\ \vdots \\ \sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n+b)(x_n)_D\end{bmatrix} \quad \left(\text{expand outer vector } \mathbf{x}_n = \begin{bmatrix}(x_n)_1 \\ \vdots \\ (x_n)_D\end{bmatrix}\right) \\
&= \frac{1}{N}\begin{bmatrix}(\mathbf{w}\cdot\mathbf{x}_1+b)(x_1)_1+\cdots+(\mathbf{w}\cdot\mathbf{x}_N+b)(x_N)_1 \\ \vdots \\ (\mathbf{w}\cdot\mathbf{x}_1+b)(x_1)_D+\cdots+(\mathbf{w}\cdot\mathbf{x}_N+b)(x_N)_D\end{bmatrix} \quad \text{(expand summation)} \\
&= \frac{1}{N}\begin{bmatrix}(x_1)_1 & \cdots & (x_N)_1 \\ \vdots & \ddots & \vdots \\ (x_1)_D & \cdots & (x_N)_D\end{bmatrix}\begin{bmatrix}\mathbf{w}\cdot\mathbf{x}_1+b \\ \vdots \\ \mathbf{w}\cdot\mathbf{x}_N+b\end{bmatrix} \quad \text{(factor into matrices)} \\
&= \frac{1}{N}[\mathbf{x}_1 \quad \cdots \quad \mathbf{x}_N]X\mathbf{w} \\
&= \frac{1}{N}X^{\mathrm{T}}X\mathbf{w}
\end{aligned}$$

$$\frac{1}{N}\sum_{n=1}^{N}t_n\mathbf{x}_n = \frac{1}{N}\begin{bmatrix}\sum_{n=1}^{N}t_n(x_n)_1 \\ \vdots \\ \sum_{n=1}^{N}t_n(x_n)_D\end{bmatrix} = \frac{1}{N}\begin{bmatrix}t_1(x_1)_1+\cdots+t_N(x_N)_1 \\ \vdots \\ t_1(x_1)_D+\cdots+t_N(x_N)_D\end{bmatrix} = \frac{1}{N}\begin{bmatrix}(x_1)_1 & \cdots & (x_N)_D \\ \vdots & \ddots & \vdots \\ (x_1)_D & \cdots & (x_N)_D\end{bmatrix}\begin{bmatrix}t_1 \\ \vdots \\ t_N\end{bmatrix} = \frac{1}{N}X^{\mathrm{T}}\mathbf{t}$$

From which you can conclude

$$\begin{aligned}
\therefore \nabla \mathcal{J}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n + b)\mathbf{x}_n - \frac{1}{N}\sum_{n=1}^{N} t_n \mathbf{x}_n \\
&= \frac{1}{N}(X^{\mathrm{T}}X\mathbf{w} - X^{\mathrm{T}}\mathbf{t}) = 0 \\
X^{\mathrm{T}}X\mathbf{w} &= X^{\mathrm{T}}\mathbf{t} \\
\therefore \mathbf{w} &= (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{t}
\end{aligned}$$

For future reference, here're useful formulas, which you can find more of here and here:

Squared norm to vector dot product	$\lVert\mathbf{x}\rVert^2 = \mathbf{x}^{\mathrm{T}}\cdot\mathbf{x} = \mathbf{x}\cdot\mathbf{x}^T$
Derivative product of functions of x	$\frac{\partial}{\partial \mathbf{x}} f(\mathbf{x})^{\mathrm{T}} g(\mathbf{x}) = f(\mathbf{x})^{\mathrm{T}}\frac{\partial g}{\partial \mathbf{x}} + g(\mathbf{x})^{\mathrm{T}}\frac{\partial f}{\partial \mathbf{x}}$ for $f, g\colon \mathbb{R}^n \to \mathbb{R}^n$
Quadratic form	$\mathbf{x}^{\mathrm{T}}A\mathbf{x}$ where A is a symmetric matrix (ie. $A_{i,j} = A_{j,i}$). $\frac{\partial}{\partial \mathbf{x}}\mathbf{x}^{\mathrm{T}}A\mathbf{x} = 2\mathbf{x}^{\mathrm{T}}A$
Matrix product to summation	$(XY)_{i,k} = \sum_{j=1}^{n} X_{i,j}Y_{j,k}$

Here's another direct solution. Note some differing notation, and that I left out b because leaving it in is annoying.

$$\begin{aligned}
\mathcal{J}(\mathbf{w}) &= \frac{1}{2N}\sum_{n=1}^{N}(\mathbf{w}\cdot\mathbf{x}_n - t_n)^2 \\
&= \frac{1}{2N}\sum_{n=1}^{N}\left(\sum_{d=1}^{D}(w_d(x_n)_d) - t_n\right)^2 \\
\frac{\partial}{\partial w_{d^*}}\mathcal{J}(\mathbf{w}) &= \frac{1}{2N}\sum_{n=1}^{N}\left[2\left(\sum_{d=1}^{D}(w_d(x_n)_d) - t_n\right)\frac{\partial}{\partial w_{d^*}}\left(\sum_{d=1}^{D}(w_d(x_n)_d) - t_n\right)\right] \quad (\text{where } d^* \in \{1, \dots, D\}) \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{d=1}^{D}(w_d(x_n)_d) - t_n\right)(x_n)_{d^*} \quad \left(= \frac{1}{N}\sum_{n=1}^{N}(\vec{w}\cdot\vec{x_n} - t_n)\,(x_n)_{d^*}, \text{iterative solution}\right) \\
&= \frac{1}{N}\sum_{n=1}^{N}\left(\sum_{d=1}^{D} w_d(x_n)_d\right)(x_n)_{d^*} - \frac{1}{N}\sum_{n=1}^{N} t_n\,(x_n)_{d^*} \\
&= \frac{1}{N}\sum_{d=1}^{D}\left(\sum_{n=1}^{N}(x_{d^*}^{\mathrm{T}})_n(x_n)_d\right)w_d - \frac{1}{N}\sum_{n=1}^{N}(x_{d^*}^{\mathrm{T}})_n t_n \\
&= \sum_{d=1}^{D} A_{d^*,d}w_d - \mathbf{c}_{d^*}
\end{aligned}$$

$$\left.\begin{aligned}
A_{d^*,d} &= \frac{1}{N}\sum_{n=1}^{N}(x_{d^*}^{\mathrm{T}})_n(x_n)_d = \frac{1}{N}\sum_{n=1}^{N}X_{d^*,n}^{\mathrm{T}}X_{n,d} \\
\mathbf{c}_{d^*} &= \frac{1}{N}\sum_{n=1}^{N}(x_{d^*}^{\mathrm{T}})_n t_n \qquad = \frac{1}{N}\sum_{n=1}^{N}X_{d^*,n}^{\mathrm{T}}t_n
\end{aligned}\right. \quad \Rightarrow \quad \begin{aligned}
A &= \frac{1}{N}X^{\mathrm{T}}X \\
\mathbf{c} &= \frac{1}{N}X^{\mathrm{T}}\vec{t}
\end{aligned}$$

From which you can conclude

$$\begin{aligned}
\therefore \nabla\mathcal{J}(\mathbf{w}) &= \begin{bmatrix}\sum_{d=1}^{D} A_{1,d}w_d \\ \vdots \\ \sum_{d=1}^{D} A_{N,d}w_d\end{bmatrix} - \begin{bmatrix} c_1 \\ \vdots \\ c_D\end{bmatrix} \\
&= \begin{bmatrix} A_{1,1}w_1 + \dots + A_{1,D}w_D \\ \vdots \\ A_{N,1}w_1 + \dots + A_{N,D}w_D\end{bmatrix} - \mathbf{c}
\end{aligned}$$

$$\begin{aligned}
&= \begin{bmatrix} A_{1,1} & \cdots & A_{1,D} \\ \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,D}\end{bmatrix}\begin{bmatrix} w_1 \\ \vdots \\ w_D\end{bmatrix} - \mathbf{c} \\
&= A\mathbf{w} - \mathbf{c} = 0 \\
\therefore \mathbf{w} &= A^{-1}\mathbf{c} \\
&= \frac{1}{N}(X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{t}
\end{aligned}$$

You can also obtain the solution by working directly with matrix derivatives if you're comfortable.

$$\begin{aligned}
\mathcal{J}(\mathbf{w}) &= \frac{1}{2N}\|X\mathbf{w}-\mathbf{t}\|^2 \\
&= \frac{1}{2N}(X\mathbf{w}-\mathbf{t})^{\mathrm{T}}(X\mathbf{w}-\mathbf{t}) && \text{By } \|\mathbf{x}\|^2 = \mathbf{x}^{\mathrm{T}}\cdot\mathbf{x} = \mathbf{x}\cdot\mathbf{x}^T \\
\frac{\partial}{\partial\mathbf{w}}\mathcal{J}(\mathbf{w}) &= \frac{1}{2N}\left[2(X\mathbf{w}-\mathbf{t})^{\mathrm{T}}\frac{\partial}{\partial\mathbf{w}}(X\mathbf{w}-\mathbf{t})\right] && \text{By } \tfrac{\partial}{\partial\mathbf{x}}f(\mathbf{x})^{\mathrm{T}}g(\mathbf{x}) = f(\mathbf{x})^{\mathrm{T}}\tfrac{\partial g}{\partial\mathbf{x}} + g(\mathbf{x})^{\mathrm{T}}\tfrac{\partial f}{\partial\mathbf{x}} \text{ (where } f(\mathbf{w}) = g(\mathbf{w}) = X\mathbf{w}-\mathbf{t}\text{)} \\
&= \frac{1}{N}(X\mathbf{w}-\mathbf{t})^{\mathrm{T}}X && \text{By } \tfrac{\partial}{\partial\mathbf{x}}(A\mathbf{x}+\mathbf{b}) = A \\
&= \frac{1}{N}[(X\mathbf{w})^{\mathrm{T}}-\mathbf{t}^{\mathrm{T}}]X && \text{Since } X\mathbf{w} \text{ and } \mathbf{t} \text{ are } n\times 1 \text{ vectors, } (X\mathbf{w}-\mathbf{t})^{\mathrm{T}} = (X\mathbf{w})^{\mathrm{T}} - \vec{t}^{\mathrm{T}} \\
&= \frac{1}{N}(\mathbf{w}^{\mathrm{T}}X^{\mathrm{T}}X - \mathbf{t}^{\mathrm{T}}X) = 0 \\
\mathbf{w}^{\mathrm{T}}X^{\mathrm{T}}X &= \mathbf{t}^{\mathrm{T}}X \\
\mathbf{w}^{\mathrm{T}} &= \mathbf{t}^{\mathrm{T}}X(X^TX)^{-1} \\
\therefore\ \mathbf{w} &= (X^TX)^{-1}X^T\mathbf{t}
\end{aligned}$$

Regularization: A function $\mathcal{R}$ artificially added to a loss function to penalize unwanted parameter values.

- ***L^2/l_2*-Regularization/Penalty:** Penalizes high magnitudes of $\mathbf{w}$. The $\frac{1}{2}$ simplifies derivative calculations.

$$\mathcal{R}(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2$$

  - **Ridge Regression:** Least squares error combined with L^2 regularization, which punishes both bad fits ($\mathcal{J}$) and high weights ($\mathcal{R}$).

$$\begin{aligned}
\mathcal{J}_{\mathrm{reg}}(\mathbf{w}) &= \mathcal{J}(\mathbf{w}) + \lambda\mathcal{R}(\mathbf{w}) \\
&= \frac{1}{2}\|X\mathbf{w}-\mathbf{t}\|^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2
\end{aligned}$$

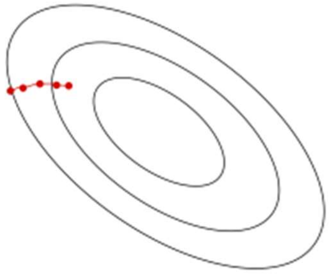
    - λ is a hyperparameter that controls the penalization of large weights.
    - We ignore $\frac{1}{N}$ in the cost function for simplicity.

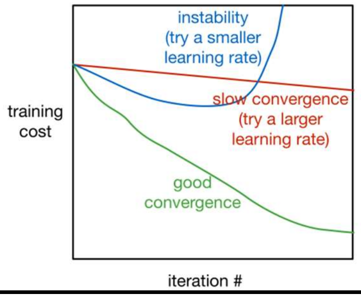
Direct Solution:

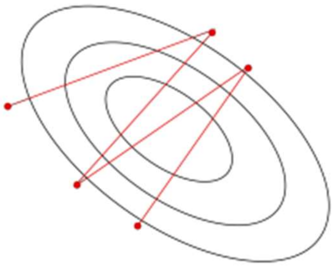
$$\begin{aligned}
\nabla\mathcal{J}_{\mathrm{reg}}(\mathbf{w}) &= \nabla\mathcal{J}(\mathbf{w}) + \nabla\lambda\mathcal{R}(\mathbf{w}) \\
&= X^{\mathrm{T}}X\mathbf{w} - X^{\mathrm{T}}\mathbf{t} + \lambda\mathbf{w} = 0 \\
\therefore\ \mathbf{w} &= (X^{\mathrm{T}}X + \lambda I)^{-1}X^{\mathrm{T}}\mathbf{t}
\end{aligned}$$

Iterative Solution:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \alpha\nabla\mathcal{J}_{\mathrm{reg}}(\mathbf{w}) \\
&= \mathbf{w} - \alpha(X^{\mathrm{T}}X\mathbf{w} - X^{\mathrm{T}}\mathbf{t} + \lambda\mathbf{w}) \\
&= \mathbf{w}(1-\alpha\lambda) - \alpha(X^{\mathrm{T}}X\mathbf{w} - X^{\mathrm{T}}\mathbf{t})
\end{aligned}$$







Small α	*Balanced α*	*High α*
➢ Slow progress, convergence takes a long time	➢ Optimal values usually 0.001 to 0.1 ➢ **Training curve** (above) can diagnose issues	➢ Instability, oscillations, misses optimum

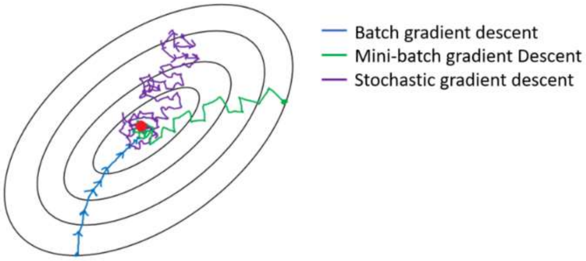
Stochastic Gradient Descent: Gradient descent that chooses a random $\mathbf{x}$ of $\mathcal{D}$ to calculate loss, instead of all of $\mathcal{D}$.

- Instead of $\mathcal{J}(\mathbf{w}) = \frac{1}{2N}\|X\mathbf{w}-\mathbf{t}\|^2$, we have $\mathcal{J}(\mathbf{w},\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\cdot\mathbf{w}-t\|^2$
- Running-time of each update is independent of size of training data (ie. N). Faster!
- Mathematically, it is unbiased and equivalent to gradient descent, given sampling is random

$$\mathbb{E}\left[\frac{\partial\mathcal{J}_{\mathrm{stochastic}}(\mathbf{w},\mathbf{x})}{\partial\mathbf{w}}\right] = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial\mathcal{J}_{\mathrm{stochastic}}(\mathbf{w},\mathbf{x}_n)}{\partial\mathbf{w}} = \frac{\partial\mathcal{J}_{\mathrm{batch}}(\mathbf{w})}{\partial\mathbf{w}}$$

Mini-Batch Gradient Descent: Similar to stochastic gradient descent, but chooses a subset $\{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ of $\mathcal{D}$.

- Batch/subset size is a hyperparameter.
- In theory, get i.i.d. examples from $\mathcal{D}$. In practice, we permute training set, go through it sequentially. Each pass over all data is called an **epoch**.



Small Batch Size	*Balanced Batch Size*	*High Batch Size*
➢ Faster updates ➢ High variance in $\mathcal{J}(\mathbf{w}, \mathbf{x})$, noisier movement ➢ Cannot exploit speed of vectorization	➢ Batch size is hyperparameter ➢ A reasonable size might be 100	➢ Slower updates ➢ Less variance in $\mathcal{J}(\mathbf{w}, \mathbf{x})$, more accurate movement ➢ Exploits speed of vectorization

**Learning rate affects how batch sizes affect noise, so when testing, it's best to start with high learning rates and move down*

Linear Classification

Binary Linear Classification: A parametric classification model that, given a $\mathbf{x} \in \mathbb{R}^d$, predicts a binary target t

- Depending on what's convenient, use $t \in \{0,1\}$ or $t \in \{-1,1\}$ for negative/positive examples
- The parameter threshold r splits input space into two

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b, \qquad y(\vec{x}) = \begin{cases} 1 & \text{if } f(\mathbf{x}) \geq r \\ 0 & \text{if } f(\mathbf{x}) < r \end{cases}$$

In practice, we can remove b and r by introducing $w_0 = b - r,\ x_0 = 1$, and squishing it into $\mathbf{w}$ and $\mathbf{x}$ like this:

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} = \begin{bmatrix} b - r \\ w_1 \\ \vdots \\ w_d \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}, \quad f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = (w_1 x_1 + \cdots + w_d x_d) + b - r, \quad y(\mathbf{x}) = \begin{cases} 1 & \text{if } f(\mathbf{x}) \geq 0 \\ 0 & \text{if } f(\mathbf{x}) < 0 \end{cases}$$

$|f(\mathbf{x})|$ is proportional to the algorithm's **confidence** in the prediction.

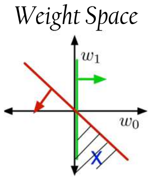
Input/Data Space: A subset of $\mathbb{R}^d$; all possible of $\mathbf{x}$.
Weight Space: A subset of $\mathbb{R}^d$; all possible values of $\mathbf{w}$.
Half-Space: A half of $\mathbb{R}^n$ divided by a hyperplane (a plane in $\mathbb{R}^{n-1}$). Can be written as $w_0 x_0 + \cdots + w_d x_d \geq r$

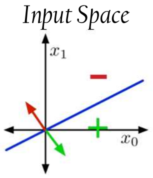
NOT Operator

$\mathbf{x}^\mathrm{T}$		t
x_0	x_1	
1	0	1
1	1	0

As $f(\mathbf{x}) = w_0 x_0 + w_1 x_1$, then for $y(\mathbf{x}) = t$ to hold, we have

$$f(\mathbf{x}_1) = w_0(1) + w_1(0) = w_0 \qquad \geq 0$$
$$f(\mathbf{x}_2) = w_0(1) + w_1(1) = w_0 + w_1 < 0$$



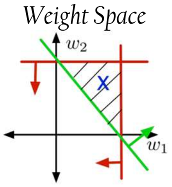


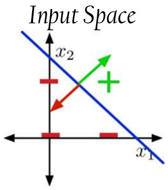
AND Operator

$\mathbf{x}^\mathrm{T}$			t
x_0	x_1	x_2	
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

As $f(\mathbf{x}) = w_0 x_0 + w_1 x_1 + w_2 x_2$, then for $y(\mathbf{x}) = t$ to hold, we must have

$$f(\mathbf{x}_1) = w_0(1) + w_1(0) + w_2(0) = w_0 \qquad < 0$$
$$f(\mathbf{x}_2) = w_0(1) + w_1(0) + w_2(1) = w_0 \qquad + w_2 < 0$$
$$f(\mathbf{x}_3) = w_0(1) + w_1(1) + w_2(0) = w_0 + w_1 \qquad < 0$$
$$f(\mathbf{x}_4) = w_0(1) + w_1(1) + w_2(1) = w_0 + w_1 + w_2 \geq 0$$





Feasible Region: A region of weight space that correctly classifies all $\mathbf{x}_i$

- Each $\mathbf{x}_i$ adds more constraints on w_i, AKA creates more half-spaces whose intersections w_i must lie in

Feasible: The classification problem, if the feasible region is non-empty

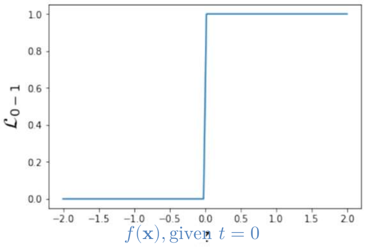
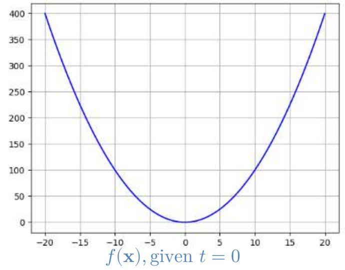
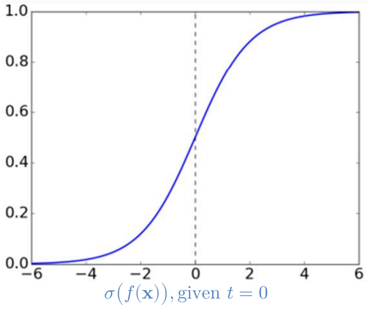
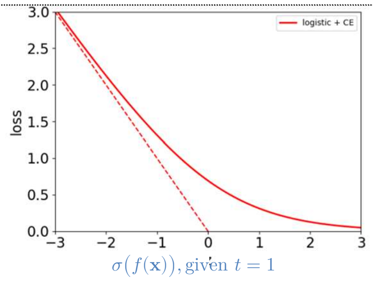
Convex Set: A set S such that $\forall x_0, x_1 \in S$, the line segment connecting x_0, x_1 is in S. That is,

$$\lambda x_0 + (1 - \lambda) x_2 \quad (\text{where } 0 \leq \lambda \leq 1)$$

- Weighted averages/convex combinations are in S: $\quad \lambda_1 x_1 + \cdots + \lambda_n x_n \quad (\text{where } \lambda_i > 0, \sum_{i=1}^{n} \lambda_i = 1)$
- **Both +/– regions of input space** are half-spaces, which are convex
- **Feasible regions of weight space** are intersections of convex half-spaces, which are convex

Linearly Separable: Training data, if linear decision rules can perfectly separate it. Almost never the case.

- If linearly separable, learn weights with linear programming/optimization (APM236), perceptron algorithm
- If not linearly separable, find weights that minimize average loss!
- Proving something is not linearly separable: Suppose it is linearly separable, then the half-spaces containing positive/negative points are convex, and so are the line segments connecting any two points in a half-space. Show a line between two points intersects a point in a different half-space, creating a contradiction

Strategy	Description	Plot
0-1 Loss	$\mathcal{L}(y(\mathbf{x}),t) = \mathbb{I}[y(\mathbf{x}) \neq t]$ $\mathcal{J}(y(\mathbf{x}),t) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}[y(\mathbf{x}_i) \neq t_i]$ Also called the miscalculation rate. *Problem:* Derivative is 0 almost everywhere and undefined at $f(\mathbf{x}) = 0$, so we can't minimize loss with gradient descent	
Squared Loss	$\mathcal{L}(y(\mathbf{x}),t) = \frac{1}{2}(f(\mathbf{x}) - t)^2$ Unlike 0-1 Loss, we're working with $f(\mathbf{x})$, whose codomain is $\mathbb{R}$ so r matters more. Set $r = \frac{1}{2}$ as the in-between of 0 and 1. *Problem:* High-confidence predictions (ie. $\lvert f(\mathbf{x})\rvert$ is high) incur large loss, even if they're correct predictions.	
Logistic Function, Squared Loss	$\sigma(x) = \frac{1}{1+e^{-x}} \in [0,1]$ $y(\mathbf{x}) = \sigma(f(\mathbf{x})) = \frac{1}{1+e^{-f(\mathbf{x})}}$ $\mathcal{L}(y(\mathbf{x}),t) = \frac{1}{2}(y(\mathbf{x}) - t)^2$ σ is the logistic function, a non-linear sigmoid (s-shaped). *Problem:* Confidently wrong answers have low gradients, resulting in slow updates (ie. the gradient signal is weak)	
Logistic Function, Cross-Entropy / Logistic Loss	$y(\mathbf{x}) = \sigma\big(f(\mathbf{x})\big) = \frac{1}{1+e^{-f(\mathbf{x})}}$ $\mathcal{L}_{\text{CE}}(y(\mathbf{x}),t) = \begin{cases} -\ln y(\mathbf{x}) & \text{if } t = 1 \\ -\ln\big(1 - y(\mathbf{x})\big) & \text{if } t = 0 \end{cases}$ $= -t\ln y(\mathbf{x}) - (1-t)\ln\big(1 - y(\mathbf{x})\big)$ If $f(\mathbf{x}) \ll 0$, $y(\mathbf{x}) \approx 0$ and is **numerically 0**, too small to store, causing computational instability (CSC336). So, expand $y(\mathbf{x})$: $\mathcal{L}(y(\mathbf{x}),t) = t\ln(1+e^{-f(\mathbf{x})}) + (1-t)\ln(1+e^{-f(\mathbf{x})})$	

Logistic Regression: Linear classification using logistic function and cross-entropy loss

- *Why is it called "regression" if we're doing classification? It's a product of history where there were no naming conventions.*

Logistic function + cross-entropy loss is optimal for linear binary classification

- Smooth, continuous, differentiable
- Small loss for correct predictions, penalizes extreme misclassifications

Direct Solution:	*Iterative Solution:*	$\vec{w} \leftarrow \vec{w} - \alpha\nabla\mathcal{L}(y(\vec{x}),t)$
DNE, since $\sigma(x)$ is non-linear	Possible, cross-entropy loss is convex in $\vec{w}$	$= \vec{w} - \alpha(y(\vec{x}) - t)\vec{x}$

Linear/logistic regression are generalized linear models with the same gradient update rules.

Linear Regression	*Logistic Regression*
$\mathbf{w} \leftarrow \mathbf{w} - \nabla\mathcal{J}(\mathbf{w})$ $= \mathbf{w} - \frac{\alpha}{N}\sum_{n=1}^{N}(y(\mathbf{x}_n) - t_n)\mathbf{x}_n$ $\downarrow$ $y(\mathbf{x}) = \mathbf{w}\cdot\mathbf{x}$	$\vec{w} \leftarrow \vec{w} - \alpha\nabla\mathcal{L}(y(\mathbf{x}),t)$ $= \vec{w} - \alpha(y(\mathbf{x}) - t)\vec{x}$ $\downarrow$ $y(\mathbf{x}) = \sigma(\mathbf{w}\cdot\mathbf{x})$
Note: This cost function incorporates all $\mathbf{x}_i$.	*Note:* This loss function uses a single $\mathbf{x}_i$.

Optimization – Derivation of iterative solution, logistic regression

$$\begin{aligned}\nabla\mathcal{L} &= \frac{\partial\mathcal{L}_{\text{CE}}}{\partial w_i}\\ &= \frac{\partial\mathcal{L}_{\text{CE}}}{\partial y}\cdot\frac{\partial y}{\partial f}\cdot\frac{\partial f}{\partial\vec{w}}\\ &= \frac{y(\mathbf{x})-t}{y(\mathbf{x})\big(1-y(\mathbf{x})\big)}\cdot\big(1-y(\mathbf{x})\big)y(\mathbf{x})\cdot\mathbf{x}\\ &= \big(y(\mathbf{x})-t\big)\mathbf{x}\end{aligned}$$

$$\frac{\partial f}{\partial \mathbf{w}} = \frac{\partial}{\partial\mathbf{w}}[\mathbf{w}\cdot\mathbf{x}] = \mathbf{x}$$

$$\begin{aligned}\frac{\partial\mathcal{L}_{\text{CE}}}{\partial y} &= \frac{\partial}{\partial y}\Big[-t\ln y(\mathbf{x})-(1-t)\ln\big(1-y(\mathbf{x})\big)\Big]\\ &= -\frac{t}{y(\mathbf{x})}-(1-t)\left(-\frac{1}{1-y(\mathbf{x})}\right)\\ &= -\frac{t}{y(\mathbf{x})}+\frac{1-t}{1-y(\mathbf{x})}\\ &= \frac{-t\big(1-y(\mathbf{x})\big)+(1-t)y(\mathbf{x})}{y(\mathbf{x})\big(1-y(\mathbf{x})\big)}\\ &= \frac{y(\mathbf{x})-t}{y(\mathbf{x})\big(1-y(\mathbf{x})\big)}\end{aligned}$$

$$\frac{\partial y}{\partial f} = \frac{\partial}{\partial f}\left[\frac{1}{1+e^{-f(\mathbf{x})}}\right] = \frac{e^{-f(\mathbf{x})}}{(1+e^{-f(\mathbf{x})})^2} = \frac{1+e^{-f(\mathbf{x})}-1}{1+e^{-f(\mathbf{x})}}\cdot\frac{1}{1+e^{-f(\mathbf{x})}} = \left(1-\frac{1}{1+e^{-f(\mathbf{x})}}\right)y(\mathbf{x}) = \big(1-y(\mathbf{x})\big)y(\mathbf{x})$$

Multi-Class Linear Classification: Given a $\mathbf{x}\in\mathbb{R}^D$, predicts a target $t\in\{1,\dots,K\}$

- **Integer Encoding:** Encoding a target as a number
  - eg. Apple is 1, orange is 2, banana is 3, $t\in\{1{,}2{,}3\}\in\mathbb{R}$
- **One-hot Vector/One-of-*k* Encoding:** Encoding a target as a unit vector in its own dimension
  - eg. Apple is $(1{,}0{,}0)$, orange is $(0{,}1{,}0)$, banana is $(0{,}0{,}1)$, and $\mathbf{t}\in\mathbb{R}^3$
  - As calculations usually imply "in-between values" and use "distances" between targets, we use one-hot vectors such that these "distances" between all targets are the same.

$$f(\mathbf{x}) = W\mathbf{x}+\mathbf{b} = \begin{bmatrix}(w_1)_1 & \cdots & (w_1)_D\\ \vdots & \ddots & \vdots\\ (w_K)_1 & \cdots & (w_K)_D\end{bmatrix}\begin{bmatrix}x_1\\ \vdots\\ x_D\end{bmatrix}+\begin{bmatrix}b\\ \vdots\\ b\end{bmatrix},\qquad y(\mathbf{x}) = \operatorname{softmax}\big(f(\mathbf{x})\big),\qquad \mathbf{t} = \begin{bmatrix}t_1\\ \vdots\\ t_K\end{bmatrix}$$

Softmax Function: A multivariate generalization of the logistic function, $\mathbb{R}^D\to(0{,}1)^D$.

- Inputs are called **logits**.
- Outputs are thought of as the probability/certainty of an option. All components add up to one.

$$\operatorname{softmax}(\mathbf{x}) = \left(\frac{e^{x_1}}{\sum_{d=1}^D e^{x_d}},\dots,\frac{e^{x_D}}{\sum_{d=1}^D e^{x_d}}\right)$$

- Equivalent to $\sigma(x)$ when $D=2$:

$$\operatorname{softmax}_1\big(f(x)\big) = \frac{e^{f_1(x)}}{e^{f_1(x)}+e^{f_2(x)}} = \frac{1}{1+e^{f_2(x)-f_1(x)}} = \frac{1}{1+e^{-f^\star(x)}} = \sigma\big(f^\star(x)\big)$$

Softmax function + cross-entropy loss is optimal for linear multiclass classification

Iterative Solution:

$$\begin{aligned}\mathcal{L}(y(\mathbf{x}),\mathbf{t}) &= -\sum_{k=1}^K t_k\ln y_k(\mathbf{x})\\ &= -\mathbf{t}\\ &\quad\cdot\big(\ln y_1(\mathbf{x}),\dots,\ln y_K(\mathbf{x})\big)\end{aligned}$$

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w}-\alpha\nabla\mathcal{L}(y(\mathbf{x}),\mathbf{t})\\ &= \mathbf{w}-\alpha\underset{\substack{\downarrow\\ y(\mathbf{x})=\operatorname{softmax}(\mathbf{w}\cdot\mathbf{x})}}{\big(y(\mathbf{x})-\vec{t}\big)}\mathbf{x}\end{aligned}$$

Recall $\mathcal{L}_{\text{CE}}(y(\mathbf{x}),t) = \begin{cases}-\ln y(\mathbf{x}) & \text{if } t=1\\ -\ln\big(1-y(\mathbf{x})\big) & \text{if } t=0\end{cases}$

This is cross-entropy re-written specifically for the binary case For multi-class, cross-entropy is $-\ln y(\mathbf{x})$, the component where $t_k=1$.

Why is the loss $-\sum_{k=1}^K t_k\ln y_k(\mathbf{x})$? Recall $\mathbf{t}$ is a one-hot vector.

When $t_k=0$, then $-t_k\ln y_k(\mathbf{x})=0$

When $t_k=1$, then $-t_k\ln y_k(\mathbf{x})=-\ln y_k(\mathbf{x})$

So the summation only computes CE at the component where $t_k=1$

Optimization – Derivation of Iterative Solution, Logistic Regression for Multiclass Linear Classification

$$\begin{aligned}\frac{\partial \mathcal{L}(y(\vec{x}),\vec{t})}{\partial \overrightarrow{w_k}} &= \frac{\partial \mathcal{L}(y(\vec{x}),\vec{t})}{\partial f(\vec{x})} \cdot \frac{\partial f(\vec{x})}{\partial \overrightarrow{w_k}} \\ &= \frac{\partial}{\partial f(\vec{x})}\left(-\vec{t} \cdot \begin{bmatrix}\ln y_1(\vec{x}) \\ \vdots \\ \ln y_K(\vec{x})\end{bmatrix}\right) \cdot \frac{\partial}{\partial \overrightarrow{w_k}}(W\vec{x}+\vec{b}) \\ &= \end{aligned}$$

$$W\vec{x}+\vec{b} = \begin{bmatrix}(w_1)_1 & \cdots & (w_1)_D \\ \vdots & \ddots & \vdots \\ (w_K)_1 & \cdots & (w_K)_D\end{bmatrix}\begin{bmatrix}x_1 \\ \vdots \\ x_D\end{bmatrix} + \begin{bmatrix}b_1 \\ \vdots \\ b_K\end{bmatrix} = \begin{bmatrix}\overrightarrow{w_1}\cdot\vec{x}+b_1 \\ \vdots \\ \overrightarrow{w_K}\cdot\vec{x}+b_K\end{bmatrix}$$

$$\therefore \frac{\partial}{\partial \overrightarrow{w_k}}(W\vec{x}+\vec{b}) = \frac{\partial}{\partial \overrightarrow{w_k}}\left(\begin{bmatrix}\overrightarrow{w_1}\cdot\vec{x}+b_1 \\ \vdots \\ \overrightarrow{w_k}\cdot\vec{x}+b_k \\ \vdots \\ \overrightarrow{w_K}\cdot\vec{x}+b_K\end{bmatrix}\right) = \vec{x}$$

$$\begin{aligned}-\vec{t}\cdot\begin{bmatrix}\ln y_1(\vec{x}) \\ \vdots \\ \ln y_K(\vec{x})\end{bmatrix} &= -\begin{bmatrix}t_1 \\ \vdots \\ t_K\end{bmatrix}\cdot\begin{bmatrix}\ln y_1(\vec{x}) \\ \vdots \\ \ln y_K(\vec{x})\end{bmatrix} \\ &= -\begin{bmatrix}t_1 \\ \vdots \\ t_K\end{bmatrix}\cdot\begin{bmatrix}\ln\Big(\text{softmax}_1\big(f(\vec{x})\big)\Big) \\ \vdots \\ \ln\Big(\text{softmax}_K\big(f(\vec{x})\big)\Big)\end{bmatrix} \\ &= -t_1\ln\Big(\text{softmax}_1\big(f(\vec{x})\big)\Big) - \cdots - t_K\ln\Big(\text{softmax}_K\big(f(\vec{x})\big)\Big) \\ &= -t_1\ln\left(\frac{e^{f_1(\vec{x})}}{\sum_{k=1}^K e^{f_k(\vec{x})}}\right) - \cdots -= -t_K\ln\left(\frac{e^{f_K(\vec{x})}}{\sum_{k=1}^K e^{f_k(\vec{x})}}\right) \\ &= -t_1\big(f_1(\vec{x}) - f_1(\vec{x})\times\ldots\times f_K(\vec{x})\big) - t_K\big(f_K(\vec{x}) - f_1(\vec{x})\times\ldots\times f_K(\vec{x})\big) \\ &= \big(f_1(\vec{x})\times\ldots\times f_K(\vec{x})\big)\big(t_1f_1(\vec{x})+\cdots+t_Kf_K(\vec{x})\big) \\ &= \big(f_1(\vec{x})\times\ldots\times f_K(\vec{x})\big)\left(\vec{t}\cdot f(\vec{x})\right)\end{aligned}$$

$$\frac{\partial}{\partial f_k(\vec{x})}\left(-\vec{t}\cdot\begin{bmatrix}\ln y_1(\vec{x}) \\ \vdots \\ \ln y_K(\vec{x})\end{bmatrix}\right) = \frac{\partial}{\partial f_k(\vec{x})}\left[\big(f_1(\vec{x})\times\ldots\times f_K(\vec{x})\big)\left(\vec{t}\cdot f(\vec{x})\right)\right]$$

$$\begin{aligned}&= \frac{\partial}{\partial f_k(\vec{x})}\left[\big(f_k(\vec{x})\big)^2\big(f_1(\vec{x})\times\ldots\times f_{k-1}(\vec{x})\times f_{k+1}(\vec{x})\times\ldots\times f_K(\vec{x})\big)\right] \\ &= 2f_k(\vec{x})\big(f_1(\vec{x})\times\ldots\times f_{k-1}(\vec{x})\times f_{k+1}(\vec{x})\times\ldots\times f_K(\vec{x})\big) \\ &= 2f_1(\vec{x})\times\ldots\times f_K(\vec{x})\end{aligned}$$

$$\therefore \frac{\partial}{\partial f(\vec{x})}\left(-\vec{t}\cdot\begin{bmatrix}\ln y_1(\vec{x}) \\ \vdots \\ \ln y_K(\vec{x})\end{bmatrix}\right) = 2\begin{bmatrix}f_1(\vec{x})\times\ldots\times f_K(x) \\ \vdots \\ f_1(\vec{x})\times\ldots\times f_K(x)\end{bmatrix}$$

$$ANSWER = (y_k(\vec{x}) - t_k)\vec{x}$$

AAAAAAA UNFINISHED

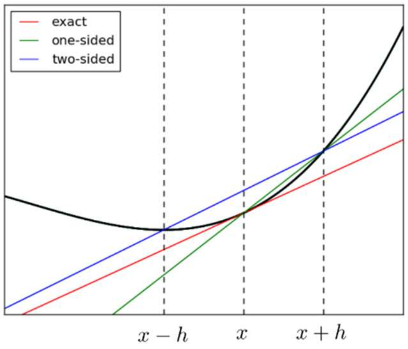
$$\frac{e^{x_1}}{\sum_{d=1}^D e^{x_d}}$$

$$\ln\left(\frac{e^{x_i}}{\sum_{d=1}^D e^{x_d}}\right) = \ln e^{x_i} - \ln(e^{x_1}+\cdots+e^{x_D}) = x_i - \ln(e^{x_1})\ldots\ln(e^{x_D}) = x_i - x_1\ldots x_D$$

$$w_i \leftarrow w_i - \frac{\alpha}{N}\sum_{n=1}^N((y_n)_i - (t_n)_i)x_i$$

Gradient Checking: Process of testing correctness of derivative implementation.

- **Finite Differences:** In numerical analysis, a way to approximate derivatives with something of form $f(x+b) - f(x+a)$. Use limit:
$$\frac{\partial}{\partial x_i} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i - h, \dots, x_n)}{2h}$$
    - Use **double precision floats** (preserve 15-17 significant digits)
    - Plug in a **tiny $\boldsymbol{h}$** (eg. 10^{-10}) for good approximations
- Let a = finite differences estimate, b = your implementation. Find **relative error** $= \frac{|a-b|}{|a|+|b|}$, which should be small (eg. 10^{-6})
- Algorithms often appear to work even when math is wrong.



KNN	*Linear Classifiers*
➢ High variance if *k* is small	➢ High bias, may underfit
➢ No parameters	➢ Parameters $\vec{w}, b$
➢ No training, slow at test-time	➢ Needs to learn a model, fast at test-time
➢ Bad in high-dimensions, scale-sensitive	➢ Good in high dimensions, not scale-sensitive

Classification Metrics

Accuracy: Fraction of examples correctly classified, equivalent to average 0-1 loss/error rate/misclassification rate

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

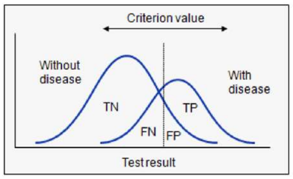
P	N	TP	TN	FP	FN
Positives	Negatives	True positives	True negatives	False positives, “type I” errors	False negatives, “type II” errors

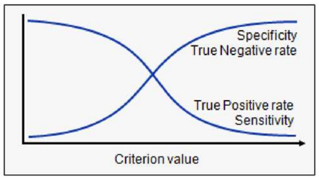
Sensitivity: True positive rate

$$\text{Sens.} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

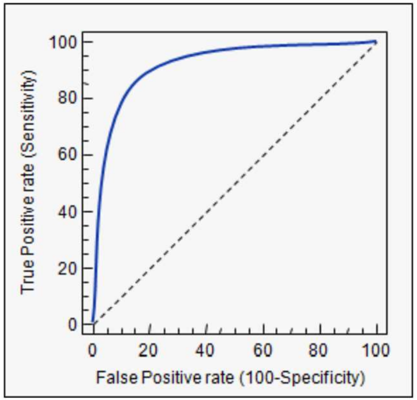
Specificity: True negative rate

$$\text{Spec.} = \frac{\text{TN}}{\text{N}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

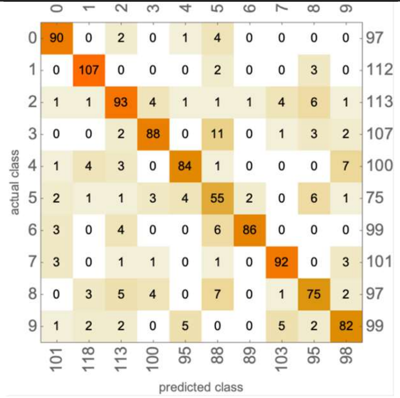




Receiver Operating Characteristic (ROC) Curve: Tracks a binary classifier’s specificity/sensitivity trade-off (top-left of curve is the “ideal”)



Confusion Matrix: For a multiclass classifier, a $K \times K$ matrix where rows are true targets, columns are predicted targets, entries are frequencies



actual class	0	1	2	3	4	5	6	7	8	9	
0	90	0	2	0	1	4	0	0	0	0	97
1	0	107	0	0	0	2	0	0	3	0	112
2	1	1	93	4	1	1	1	4	6	1	113
3	0	0	2	88	0	11	0	1	3	2	107
4	1	4	3	0	84	1	0	0	0	7	100
5	2	1	1	3	4	55	2	0	6	1	75
6	3	0	4	0	0	6	86	0	0	0	99
7	3	0	1	1	0	1	0	92	0	3	101
8	0	3	5	4	0	7	0	1	75	2	97
9	1	2	2	0	5	0	0	5	2	82	99
	101	118	113	100	95	88	89	103	95	98	

predicted class