Machine Learning: "parameterized" program, learning algorithm finds the set of parameters that best approximates desired function/behavior. Tranditional programming: Data + Program $\rightarrow$ Computer $\rightarrow$ Output. ML: Data + Output $\rightarrow$ Computer $\rightarrow$ Program. Components: representation (decision tree, hyperplane, etc), evaluation (accuracy, entropy, etc), optimization (gradient descent etc).
Inductive/Supervised Learning: training data includes desired outputs. Learn parameters on training set, tune *hyperparameters*, compute accuracy of test set. Overfitting = fitting training data very closely but not generalizing well. Inductive bias: set of assumptions that the learner uses to predict outputs of given inputs that it has not encountered.
Data: Training data = 70%, Held out/validation/development data = 10%, Test data = 20%.
Hyperparameter: non-learnable algorithm settings, controls aspects of the learning process, you set before training.
Parametric vs. Non-parametric: # of parameters fixed vs grows with data, model complexity fixed/predefined vs grows with data, linear regression, logistic regression, neural networks (usually) vs k-NN, decision trees, Gaussian processes, kernel methods.
Decision Trees: Every possible finite discrete-valued function can be represented by some decision tree & Given a training dataset, the target function is surely included. Internal node tests value of a feature. Leaf node specifies class. Feature vectors represent values of features. Want distribution of examples in each node to be pure. Depth # = # of used features? ID3: Look at current training set, determine best attribute, split according to values. If the new set if pure, it's a leaf. Otherwise, recursion. $Entropy(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$. Low entropy = high information. Attribute = x, set = S. $Entropy(S|x) = \sum_{v \in values(x)} \frac{S_v}{S} Entropy(S_v)$. $Gain(S, x) = Entropy(S) - Entropy(S|x)$. Given subset S, if you partition on x, subtract new entropy from old entropy to get entropy difference. Want a big difference/big entropy reduction. Large number of attr. values more likely to be pure so gain is biased, generalization decreases.
k-Nearest Neighbor: classify x by finding training example (x_i, y_i) nearest to x. Examine k-closest training data points to x, assign to most frequently occurring class. Optionally give closer points larger weights/more distant points smaller weights. Increasing k reduces variance, increases bias; trade-off between overfitting (small k) and underfitting (large k). If discrete-valued target function: given query x_q, take vote among its k nearest neighbors. If real-valued function: take mean of f values given k nearest neighbors. $\hat{f}(x_q) \leftarrow \frac{1}{k}\sum_{i=1}^{k} f(x_i)$. Good when instances map to points in R^n, < 20 attributes per instance, lots of training data.
Advantages: very fast, complex target functions, no losing info.
Distadvantages: slow at query time, large data sets slow for determining nearest neighbor, easily fooled by *irrelevant attributes*, for high-d space the nearest neighbor may not be very close (curse of dimensionality) i.e. in high-d spaces, points drawn from a probability distribution so hypercube side length in d dimensions $l^d \approx \frac{k}{n}$...at high d the k nearest neighbors are not much closer than any other points and the probability for a random uniformly distributed point to stay in the interior is $(1 - 2\epsilon)^d$ where ϵ is l difference between two hypercubes, decreases as d increases, but high d ok in practice bc data not distributed uniformly in general, might still work if low intrinsic d in the data (i.e. data occupies a square in 3D space).
Decision boundaries: Each line segment is equidistant between two points of opposite classes. The more examples that are stored, the more complex the decision boundaries can become.
Distance functions: most common Euclidean $d_E(x^i, x^j) = \text{sqrt}\sum_{k=1}^{p}(x_k^i - x_k^j)^2$ each is variable measured in the same units. If not, standardize by dividing by sample SD to make them all equally important $\hat{\sigma}_k = \text{sqrt}\frac{1}{n}\sum_{i=1}^{n}(x_k^i - \bar{x}_k)^2$ where sample mean $\bar{x}_k = \frac{1}{n}\sum_{i=1}^{n} x_k^i$. Weighted Euclidean distance $d_{WE}(i, j) = \text{sqrt}\sum_{k=1}^{p}\left(w_k(x_k^i - x_k^j)\right)^2$. Could weight each feature by its mutual information with the class. Other distance metrics: Minkowski or L_λ $d(i, j) = $ lambdath rt $\sum_{k=1}^{p}\left(w_k(x_k(i) - x_k(j))\right)^\lambda$. Manhattan, city block, or L_1: $d(i, j) = \sum_{k=1}^{p}|x_k(i) - x_k(j)|$. L_∞: $d(i, j) = \max_k |x_k(i) - x_k(j)|$.
Irrelevent attributes: Get rid of any irrelevant features (dimensions), i.e. take a 2D plot to a 1D line.
Loss Function: quantifies difference between y and $\hat{y}$. For regression, squared loss$=L(\hat{y}, y) = (\hat{y} - y)^2$, absolute loss$=L(\hat{y}, y) = |\hat{y} - y|$. For binary classification, zero/one loss=0 if $\hat{y} = y$ and 1 otherwise.
Logistic Regression: $P(Y|X) = P(Y)$ given X. Take $wx + b \rightarrow$ *probability*, so it's not a confident binary answer. $\sigma(z) = 1/(1 + e^z)$, $\sigma(wx + b) =$
Perceptron: linear model, cannot represent non-linear data, convergence theorem: model will find a hyperplane to separate data but it might not be good. Want larger margin with training data (i.e. line is further away from all data points). Parameters $w = (w_1, w_1, ...w_D), b$. Activation: $a = \left[\sum_{d=1}^{D} w_d x_d\right] + b$. Predict +1 if $a > 0$, -1 otherwise. w is hyperplane where one side corresponds to $Y = +1$ and the other $Y = -1$. Can represent $g(x_1, x_2) = AND(x_1, x_2)$. Not all functions linearly separable so need networks. There's $d = 1, ..., D$ one for each feature. If $ya \leq 0$ meaning the sign agrees, then $w_d \leftarrow w_d + yx_d$ for all d and $b \leftarrow b + y$. If same example twice, should be better second time. Many epochs overfit, little epochs underfit. Shuffle in case there's a pattern. Training accuracy should get better over epochs but test data probably bottoms out and then increases again. Converges if it can correctly classify every training example. Multiclass: Have a weight vector for each class w_y (i.e. poisonous & tasty), calc activation for each class, highest wins.
Linear Regression: Supervised learning steps: 1. Decide what the input-output pairs are, 2. Decide how to encode inputs and outputs thus deciding the input space and output space, 3. Chose a class of hypotheses/mappings, 4. Choose an error function (cost function) to define the best, hypothesis, 5. Choose an algorithm to search through the space of hypotheses efficiently. You will find a closed-form solution. $h_w(x) = \sum_{i=0}^{n} w_i x_i = w^T x$. issues: 1. If matrix is too large, commutation will be expensive, 2. or inverse matrix may not exist
Cost Function: measures difference between y and $\hat{y}$ (least mean square regression). $J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(h_w(x_i) - y_i\right)^2$, choose w to minimize mean squared error $J(w)$. To minimize, compute gradient $\partial/\partial w_j J(w)$, set to 0. $\nabla f(u_1, u_2, ..., u_n)$ computes a vector of partial derivatives. Solution: $\nabla_w J = X^T X w - X^T y$, $w = (X^T X)^{-1} X^T y$ but remember inverse only exists if columns of X are linearly independent.
Irrelevent attributes: Get rid of any irrelevant features, i.e. take a 2D plot to a 1D line. Remaining indepent features is dimensions.
Logistic Regression: classification is prob. not certain yes/no. No closed-form solution. Special case of generalized linear model. Predict class probs $P(Y|X)$ ($P(Y)$ given X) w/ linear model. Turn $wx + b$ into prob. Logistic function: $\sigma(z) = \frac{1}{1+\exp(-z)}$ takes $(-\infty, +\infty)$ to $(0, 1)$ (inverse of logit function). For 2 classes: $P(Y = 1|X = \langle X|_1, ..., X_n|)\rangle = \frac{1}{1+\exp(w_0 + \sum_i w_i X_i)}$ then $P(Y = 0|X = \langle X|_1, ..., X_n|)\rangle = \frac{\exp(w_0 + \sum_i w_i X_i)}{1+\exp(w_0 + \sum_i w_i X_i)}$ so $\frac{P(Y=0|X)}{P(Y=1|X)} = \exp(w_0 + \sum_i w_i X_i)$. Take log, $Y = 0$ if RHS¿0 (linear classification rule). Choose model where reality is probable. $\prod_{(x,y)\in D} \hat{P}(y|x) = \prod_{(x,y)\in D} \frac{1}{1+\exp(-y(w^T x + b))}$. Take log then left is conditional log-likelihood and right is negative logistic loss, convex function of w (easy to optimimize).

Logistic Loss: Maximize log-likelihood, minimize logistic loss. 0-1 loss is non-smooth bc small change in parameter could lead to big change in acc. Examples of loss functions: Zero/one: $1[y\hat{y} \leq 0]$ (there's loss if y and y aren't either both 1 or both 0). Hinge: $\max(0, 1 - y\hat{y})$. Logistic: $\frac{1}{\log 2}\log(1+\exp[-y\hat{y}])$. Exponential: $\exp[-y\hat{y}]$. Squared: $(y-\hat{y})^2$. Optimization techniques: find min/max of multivariate functions. Compute gradient over all examples before updating rather than perceptron which updates model immediately after each misprediction.
Gradient Descent: for all k, do $g(k) \leftarrow \nabla_z F|_{z^{(k-1)}}$ and $z(k) \leftarrow z^{(k-1)} - \eta(k)g(k)$ and return $z(k)$.$g(k)$ is gradient at current location and $\eta(k)$ *hyperparameter* is the learning rate at current location. Too small = slow learning/get stuck in local minimum/underfitting, too large = overshoot minimum/loss fluctuates or diverges. For logistic regression $F = -\log\prod_{(x,y)\in D}\hat{P}(y|x) = \sum_{(x,y)\in D}\log\left(1+\exp(-y(w^Tx+b))\right)$. Take gradient wrt w_i and b.
Mini Batching: A way to approximate the gradient computation over the whole training data. Cons: not guaranteed to give steepest direction. Pros: Efficient, less variance than stochastic gradient descent (batch=1, updates parameters using one datapoint at a time), can help jump out of shallow minima.
Overfitting: Maximum likelihood solution prefers higher weights, so larger feature influence, can = overfit. Small weights = simpler solution, smaller variance, less overfit (bias and variance tradeoff). Weight regularization penalizes high weights. Adjust bias with regularization function, i.e. sum of loss functions for each data point $+\lambda R(w,b)$. High weight = high variance. Large λ = paying attention to small weights & underfitting. Small λ = minimizing error, not paying attention to generalization (not sure if this is the right λ). λ is a *hyperparameter.*
Regularizers: P-Norms (norm of weight vector) L_2 (circle): square root of sum of squared weights, L_1 (diamond): sum of absolute weights. $||w||_p = \left(\sum_d |w_d|^p\right)^{1/p}$. If const, isosurface. L_p. L_2 = ridge regression. Irrelevant features have small but non-zero weights. L_1 = lasso. Can make some weights exactly zero. Computationally more expensive, biased toward sparse solutions. Choice of regularizer is *hyperp.*
Convex Functions: If f and g are convex functions, then $\alpha f + \beta g$ also convex for any real numbers α and β. First order: f is convex $\longleftrightarrow$ for all a,b: $f(a) \geq f(b) + \nabla f(b)^T(a-b)$ (the function is globally above the tangent at b).
Linear Models: In linearly separable binary classification, there's infinite number of hyperplanes that work. Which is best and for which points should we be most confident? Margin is 2x Euclidean distance to nearest training point. Use validation on held data to tune hyperps. Cross-validation=split data into subsets, rotate through which set is used for validation then avg results.
Margins: Linear SVM (support vector machine) is perceptron for which w, b maximizes margin. If γ_i is distance from x_i to decision boundary: $\lambda_i = \frac{w}{||w||}\cdot x_i + \frac{b}{||w||}$. Margin of hyperplane is $2M$ where M is minimum γ_i. We want to max margin (i.e. max of this min). Treat γ_i as the contraints. $M \leq \gamma_i$ and $||w||M = 1$. Now we want min $||w||$ wrt w, b subject to $y_i(wx_i + b) \geq 1$. Now constraints are linear. What if we minimize $||w||^2$ instead of $||w||$ (everything else the same)? Quadratic programming. It's positive semidefinite/convex.
Quadratic Programming: n variables, m constraints. Given a real-valued n-dimensional vector c, an $n\times n$-dimensional real symmetric matrix Q, an $m \times n$-dimensional real matrix A,and an m-dimensional real vector b, find an n-dimensional vector x that minimizes $\frac{1}{2}x^TQx + c^Tx$ subject to $Ax \leq b$.
Primal & Dual Optimization: Primal: $\min_w f(w)$ such that $g_i(w) \leq 0$, $i = 1, ..., k$. Generalized Lagrangian: $L(w,\alpha) = f(w) + \sum_{i=1}^k \alpha_i g_i(w)$, where α_i are Lagrange multipliers. Dual: $P(w) = \max_{\alpha:\alpha_i\geq 0} L(w,\alpha)$, then $P(w) = f(\infty)$ if all constraints are satisfied or $+\infty$ otherwise so we need to compute $p^* = \min_w P(w) = \min_w \max_{\alpha:\alpha_i \geq 0} L(w,\alpha)$. Let $d^* = \max_{\alpha:\alpha_i\geq 0}\min_w L(w,\alpha)$ then $d^* \leq p^*$, $p^* = L(W^p, \alpha^p)$, $d^* = L(w^d, \alpha^d)$, $d^* = L(w^d,\alpha^d) \leq L(w^p, \alpha^d \leq L(w^p, \alpha^p) = p^*$. If f, g_i are convex and g_i can all be satisfied simultaneously for some w, then $d^* = p^* = L(w^*, \alpha^*)$. Take $\partial/\partial w_i L(w^*, \alpha^*))$, you get $\alpha_i^* \geq 0$. Lagrangian: $L(w,b,\alpha) = \frac{1}{2}||w||^2 + \sum_i \alpha_i(1 - y_i(wx_i + b))$. Primal: $\min_{w,b}\max_{\alpha:\alpha_i\geq 0} L(w,b,\alpha)$. Dual: switch max/min. Quadratic objective + linear constraints (both convex) = optimal solutions coincide.
Support Vector Machine: Find optimal αs, $\alpha_i > 0$ for $1 - y_i(wx_i + w_0) = 0$. Points lying on edge of margin = support vectors bc they define *decision boundary.* Calc output using dot prod of point w/ sup vec: sign $\sum_{i=1}^m \alpha_i y_i(x_i x) + w_0$. Non-linearly separable: replace x_i w/ $\phi(x_i)$. $w = \sum \alpha_i y_i \phi(x_i)$. Classify w/ sign of that $+w_0$. Slow if many sup vecs.
Kernel Functions: learning alg. w/ dot-prod = generalize to kernels. Kernel=similarity, any func $K(x_1, x_2) = \phi(x_1)\phi(x_2)$. Linear: $K(x_i,x_j) = x_i^T x_j$, Polynomial: $(1 + x_iTx_j)^p$, RBF/Gaussian: $\exp(-||x_i - x_j||^2/2\sigma^2)$, sigmoid: $\tanh(\beta_0 x_i^T x_j + \beta_1)$. Pos semidef.
Soft margin: allow misclassifications: $y_i(wx_i + w_0) \geq 1 - \zeta_i$. ζ_i in (0,1) point w/in margin, $zeta_i \geq 1$ point misclassified. Soft error $\sum_i \zeta_i$. Mult by const C control overfitting. Large C=decrease margin if it helps. Overfit: low margin, many sup vecs.
Multiclass: one-vs-all, n classifiers, choose class w largest margin or one-vs-one, $n(n-1)/2$ classifiers, choose class chosen by most classifiers.
Neural Networks:
Activation functions: sign(a), $\sigma(a)$, $\tanh(a)$, ReLU$(a) = \max(a, 0)$, SoftPlus$(a) = \log(1 + e^a)$, ELU$(a) = a$ for $a \geq 0$, $\alpha(e^a - 1)$ for $a < 0$.
Vanishing gradients: small gradient can block stochastic grad. desc. esp for large network. Bounded act func (sigmoid, tanh,) are prone.