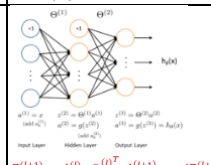
Couse Summary

Machine Learning Algorithm

ML Algorithm	Supervised Learning				Unsupervised Learning		
	Linear Regression	**Logistic Regression**	**Neural Network**	**Support-Vector Machines (SVMs with Kernels)**	**K-Means**	**Principle Component Analysis (PCA)**	**Anomaly Detection**
Motivation	Continuous output	Discrete output Classification problem (K-class)	Non-linear hypothesis w/o exhausting all non-linear terms	SVMs: Large margin classifier Kernel: non-linear decision boundary => linear	Most popular clustering algorithms	Dimensionality reduction for data compression/visualization	Very small # of positive examples
Input	$x_j^{(i)}$ - value j of i-th training example X – ***m x (n+1)*** matrix, row is each training example, and column is each feature (add bias feature 1 to first column)	X – ***m x (n+1)*** matrix (bias feature 1 added to first column)	X – ***m x (n+1)*** matrix, include bias unit Input layer (n units + 1 bias units), hidden layers (layer j has S_j units)	F – ***m x (m+1)*** matrix (Kernels of X refer to landmarks L, initialization: L = X) (preprocessing X – ***m x n*** matrix – to F)	X – ***m x n*** matrix M – ***K x n*** matrix (random pick K training examples as *cluster centroids*)	X – ***m x n*** matrix	X – ***m x n*** matrix
Output	y – ***m x 1*** vector, row is each training example	y – ***m x K*** matrix, $y^{(i)} = [0\ 0 \dots 1 \dots 0]$, $y_k^{(i)} = 1$ (one-hot notation) (or y – ***m x 1*** vector, $y^{(i)} = k$ belongs to class k)	*Assume a classification problem:* y – ***m x K*** matrix, $y^{(i)} = [0\ 0 \dots 1 \dots 0]$ output layer (K units)	y – ***m x K*** matrix, $y^{(i)} = [0\ 0 \dots 1 \dots 0]$, $y_k^{(i)} = 1$ (class k)	-- unsupervised -- Algo predicts the output as a ***m x 1*** vector, row is each training example, column is the class k (=1,2,…,K)	-- unsupervised -- Algo output: Z – ***m x K*** matrix (K <= n, and K=2 or 3 for easy plot)	y – ***m x 1*** vector (y_i = 1 anomalous; y_i = 0 normal)
Parameters	θ - ***(n+1) x 1*** vector, θ_0 is for bias feature	θ - ***(n+1) x K*** matrix, column correspond to K output columns of y, row is parameter	*Mapping function between layer L and layer L+1:* $\Theta^{(l)}$ - ***S_{L+1} x S_L + 1*** matrix	θ - ***(m+1) x K*** matrix, column correspond to K output columns of y, row is parameter	Null	$U_{Reduced}$ – ***m x n*** matrix	Null
Hypothesis	$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ x_0 = 1 (bias feature); x_j can be x^2, x^3, … or $x_a x_b$ – a polynomial or nonlinear problem $h_\theta(x^{(i)}) = \theta^T x^{(i)}$, $h_\theta(X) = X \cdot \theta$	$h_\theta^{(k)}(x) = P(y = k \mid x; \theta)$ k = 1,2,3 …, K $h_\theta(x) = g(\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n)$ $g(z) = \frac{1}{1+e^{-z}}$ – sigmoid/logistic function $h_\theta(x^{(i)}) = \frac{1}{1+e^{-\theta^T x^{(i)}}}$, $h_\theta(X) = \frac{1}{1+e^{-X \cdot \theta}}$	 $Z^{(l+1)} = A^{(l)} \cdot \Theta^{(l)^T}$, $A^{(l+1)} = g(Z^{(l+1)})$ $A^{(l+1)}$ *then add mx1 of "1" as bias units*	Compute the kernels from given X: $l^{(i)} = x^{(i)}$; $f_m^{(i)} = similarity(x^{(i)}, x^{(m)}) = exp\left(-\frac{\sum_{j=1}^{n}(x_j^{(i)} - x_j^{(m)})^2}{2\sigma^2}\right)$, $f_0^{(i)} = 1$ (bias term) $h_\theta(f^{(i)}) = 1, if\ \theta^T f^{(i)} \geq 0; else\ h_\theta(f^{(i)}) = 0$ **Note**: perform feature scaling before using the Gaussian Kernel. Other type of kernels: polynomial, string kernel, chi-square, histogram intersection, …	1. Random initialization: K cluster centroids 2. Cluster assignment $c^{(i)} := index\ (1 \sim K)$ closest centroid to $x^{(i)}$ 3. Move centroids $\mu^{(k)} :=$ average of points assigned to cluster k 4. compute cost function	1. Data processing: mean normalization 2. Compute "Covariance matrix": $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \cdot (x^{(i)})^T$ – ***n x n*** matrix 3. compute the eigenvectors of Σ: [U, S, V] = svd(Sigma). *Octave Code* svd (singular value decomposition); Sigma -- Σ 4. take the first K columns of $U_{n \times n}$: $z^{(i)} = U_{reduced}^T \cdot x^{(i)}$	Gaussian (Normal) distribution: *Mean μ, variance σ^2, $x \sim N(\mu, \sigma^2)$* $P(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}*\sigma} * \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$ Parameter Estimation: $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$, $\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$ Multi-variant Gaussian distribution: $P(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} * |\Sigma|^{\frac{1}{2}}} * \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right)$
Optimization Objective (cost function)	Mean Squared Error: $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ -- cost function of linear regression is always convex (converge) -- for sufficient small α, J(θ) should decrease on every iteration (plot J(θ) v.s. integration for debug)	Logistic cost function: $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)} log h_\theta(x^{(i)})_k + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})_k)]$ Sum over all K classes and all m training sets.	Feedforward propagation: $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)} log h_\theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k)]$ Sum over all K classes and all m training sets.	Cost function: $J(\theta) = C\sum_{i=1}^{n}[y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_2(\theta^T f^{(i)})] + \frac{1}{2}\sum_{i=1}^{m}\theta_j^2$ -- C=1/λ, controls the penalty for misclassified training examples Using kernels with logistic regression can be very slow	Cost function: $J(c, \mu) = \frac{1}{m}\sum_{i=1}^{m}\|\|x^{(i)} - \mu_{c^{(i)}}\|\|^2$ -- repeat the entire steps (random initialization 100 times) to get rid of the local minimum; pick clustering that gave lowest cost function	1. Reconstruct with approximation $x_{approx}^{(i)} = U_{reduced} \cdot z^{(i)}$ 2. Choose the smallest K that can retain p (i.e. 99%) of variance: $\frac{\frac{1}{m}\sum_{i=1}^{m}\|\|x^{(i)} - x_{approx}^{(i)}\|\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|\|x^{(i)}\|\|^2} \leq 1 - p$	Given new example x, compute p(x): $p(x) = \prod_{j=1}^{n}\frac{1}{\sqrt{2\pi}*\sigma_j} * \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$ Then predict: $y = \begin{cases} 1 & if\ p(x) < \epsilon\ (anomaly) \\ 0 & if\ p(x) \geq \epsilon\ (normal) \end{cases}$ *Use CV to choose parameter ϵ* -- data is very skew, we need to use precision/recall, F1-score.
Gradient Descent	$\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}$ $\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ α -- learning rate	$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$	Backpropagation: $g'(z) = \frac{d}{dz}g(z) = g(z) \cdot (1 - g(z))$ $\delta^{(l)} = \left(\delta^{(l+1)} \cdot \Theta^{(l)}(:, 2{:}end)\right) \odot g'(z^{(l)})$ $\frac{\partial J(\theta)}{\partial \theta^{(l)}} = \frac{1}{m}(\delta^{(l+1)})^T \cdot A^{(l)}$	-- SVM model computes the gradient for you	Null	Null	Null
Vectorization (regularized)	$J(\theta) = \frac{1}{2m}(X \cdot \theta - y)^T(X \cdot \theta - y) + \frac{\lambda}{2m}\sum[\theta_{row1=0}{}^2]$ * here is sum over all matrix element squares $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}X^T(X \cdot \theta - y) + \frac{\lambda}{m}\theta_{row1=0}$	$J(\theta) = -\frac{1}{m}\sum_{along\ row}\left[y \odot \log\left(\frac{1}{1+e^{-X \cdot \theta}}\right) + (1 - y) \odot \log\left(1 - \frac{1}{1+e^{-X \cdot \theta}}\right)\right] + \frac{\lambda}{2m}\sum\sum[\theta_{row1=0}{}^2]$, ***1 x K*** vector, ⊙ is the element-wise product of two matrix $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m}X^T(\frac{1}{1+e^{-X \cdot \theta}} - y) + \frac{\lambda}{m}\theta_{row1=0}$	$J(\Theta) = -\frac{1}{m}\sum\sum[y \odot \log(h(\Theta)) + (1 - y) \odot \log(1 - h(\Theta))] + \frac{\lambda}{2m}\sum_{layer}\sum\sum[\Theta_{row1=0}{}^2]$ Regression part: sum over all matrix elements; regularization part: sum over all matrix elements and all layer Θ matrix $\frac{\partial J(\Theta)}{\partial \Theta^{(l)}} = \frac{1}{m}(\delta^{(l+1)})^T \cdot A^{(l)} + \frac{\lambda}{m}\Theta_{col\_1=0}$ "unroll" each Θ and append together	$J(\theta) = C\sum_{along\ row}[y \odot cost_1(\theta^T f^{(i)}) + (1 - y) \odot cost_2(\theta^T f^{(i)})] + \frac{1}{2}\sum\sum[\theta_{row1=0}{}^2]$ -- large C: lower bias, high variance -- large σ^2: features $f^{(i)}$ vary more smoothly, high bias and low variance	Null	$Z = X \cdot U_{reduced}$ Compute the % of variance from diagonal matrix S: $S = \begin{bmatrix} S_{11} & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & S_{nn} \end{bmatrix}$, $\frac{\sum_{j=1}^{k} S_{jj}}{\sum_{j=1}^{n} S_{jj}} \geq p$	Transform non-Gaussian features to make them look more Gaussian: E.g.1: $x_j = \log(x_j + c)$ E.g.2: $x_j = \sqrt{x_j + c}$ Multi-variant Gaussian Distribution parameter: $\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$; $\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$
Decision boundary	Null	$\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n = 0$ or $\theta^T x^{(i)} = 0$ $(\theta^T x^{(i)} > 0, y = 1; else\ \theta^T x^{(i)} < 0, y = 0)$	Null	$\theta^T f^{(i)} = 0$, but with the choice of θ that: $\theta^T f^{(i)} \geq 1\ if\ y^{(i)} = 1$; $\theta^T f^{(i)} \leq -1\ if\ y^{(i)} = 0$ effectively more margin	Null	Null	Null
Notes /Limitations	-- Pre-process data by **mean normalization** of each feature: $x_j = \frac{x_j - u_j}{s_j}$, where u_j is mean and S_j is range/ standard deviation (save computation time) -- analytical solution of θ when dimension (# of training set is small): $\boldsymbol{\theta} = (X^T X + \lambda\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \dots \end{pmatrix})^{-1} X^T y$	-- $h_\theta(x)$ compute the probability of each class; for the new test set, output is the class k that maximize the $h_\theta(x)$ -- binary classification/ multi-class classification	-- randomly initialize $\Theta_{ij}^{(l)}$ to a random value in [-ε, +ε] (symmetry breaking to avoid redundant hidden units) -- we can choose $\varepsilon_{initial} = \frac{\sqrt{6}}{\sqrt{S_L + S_{L+1}}}$ -- usually more hidden layer is better, choose same # of units in every hidden layer	-- Idea of kernels is to set each training example as a landmark $l^{(i)} = x^{(i)}$, then transfer each $x^{(i)} \to f^{(i)}$, wherein each element $f_m^{(i)} = similarity(x^{(i)}, x^{(m)})$ -- if n is small and m is intermediate (n=1~1000, m=10~1000), we can use SVM with kernel; otherwise, use logistic regression or SVM without kernel -- for linear classification problem, use logistic regression or SVM without kernel	-- suitable for K not too large (2~10) -- choose cluster number K: (1) elbow method (i.e. J v.s. K plot); (2) based on later/downstream purpose (i.e. T-shirt size)	-- Reduce from n-dimension to k-dimension: find k vectors $u^{(1)}$, $u^{(2)}$, …, $u^{(k)}$ onto which to project the data, so as to minimize the projection error -- This mapping optimization should be defined by running PCA only on training set, and then applied to CV or test set. -- **misuse** includes: (1) use PCA to reduce features to prevent overfitting; (2) use PCA before even try with original data set	-- Differ from supervised learning, anomaly detection has very small # of positive examples, hard for any algo to learn the positive examples, and new anomalies may look nothing like trained examples. -- Choose feature that might take on unusually large/small values (exaggerate the anomaly signal) -- If features form clear correlation, use multivariant GD in case of miss some anomaly (Σ has off-diagonal elements)
Applications	# linear regression problem # recommender system	# Multi-class Classification: One-vs-All	# Binary operations (i.e. XNOR) # Multi-class Classification: One-vs-All	# non-linear Multi-class Classification # One-vs-All (e.g. spam-email classification, weather prediction)	# Market segmentation; # social network analysis; # organize computing clusters; # Astronomical data analysis	# image compression # data visualization # reduce dimension to speed up learning algorithm	# Fraud detection # Manufacturing outliers #Monitor

Summarizer: Renjie (Ryan) Chen **ren.j.chen@gmail.com**

Build a Machine Learning System

ML System	Algorithm Computation				ML Diagnostic		
	Optimization Algorithm	**Regularization**	**Gradient Checking**	**Evaluate a Hypothesis**	**Bias/Variance & Learning Curve**	**Error Analysis**	**Ceiling Analysis**
Explanations	*Basic:* -- Gradient descent *Advanced:* -- Conjugate gradient; BFGS; L-BFGS *Note:* -- advanced algos don't need to pick the learning rate, often faster than gradient descent	*Motivation:* Address overfitting issue by reduce the magnitudes of the parameters θ_j while keep the # of features unchanged	*Motivation:* Numerically estimate the gradient in order to validate the gradient decent terms. Disable it before running the advanced optimization algorithm	*Evaluate a Hypothesis:* 1. shuffle data set and divide into 3 groups: 60% training; 20% cross-validation (CV); 20% test 2. *Optimizing the parameter based on training error -> determine the degree of model based on CV error -> estimate the generalization error with test set*	*Under fitting (high bias):* Both J_{train} and J_{CV} are high; λ could be too large; Plot learning curve (error v.s. training set size m), J_{train} and J_{CV} converge but both very high *Over fitting (high variance):* J_{train} is low and J_{CV} is high; λ could be too small; Plot learning curve, J_{train} and J_{CV} converge with a gap; increase training set size can help	*Motivation:* Rare class/skew class Training example belongs to one class out-numbers heavily the other classes. Actual Class / Predict class 1: True Positive; False Positive 0: False Negative; True Negative	*Motivation:* Prioritize the high-error modules in a ML pipeline Machine learning pipeline: Image → Text detection → Character segmentation → Character recognition
Applications	*Octave Code* Step1: write a function *[J, grad] = costFunction(theta, X, y)* Step2: compute the optimal theta *Options = optimset ('GradObj','on','MaxIter','100');* *initialTheta = zeros(2,1);* *[optTheta, functionVal, exitFlag] = fminuc(@costFunction, initialTheta, options);*	*Cost function:* $J(\theta) = J(\theta) + \frac{\lambda}{2m}\sum_{j=1}^{n}\sum_{k=1}^{K}\theta_{j,k}^2$ * Remember θ is (n+1) x k, the first row of θ_0^k are excluded *Gradient:* $\frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J(\theta)}{\partial \theta} + \frac{\lambda}{m}\theta_{first\_row=0}$	*Concept:* $\frac{\partial J(\theta)}{\partial \theta_j} \approx \frac{J(\theta_j+\epsilon,\theta_{1,2,\ldots}) - J(\theta_j-\epsilon,\theta_{1,2,\ldots})}{2\epsilon}$ *Octave Code* *For i=1:n,* *thetaPlus = theta; thetaMinus = theta;* *thetaPlus += epsilon; thetaMinus -= epsilon;* *gradApprox(i) = (J(thetaPlus) – J(thetaMinus))/2/epsilon;* *end;* check that gradApprox ≈ DVec from backpropagation (neuralNetwork)	*Test set error:* Regression problem – mean square error (same as cost function) Classification problem: $err(h_\theta(x), y) = 1$ *if* $h_\theta(x) \geq 0.5$ *and y=0 or if* $h_\theta(x) < 0.5$ *and y=1; elsewise,* $err(h_\theta(x), y) = 0$ *Note:* When optimizing the parameters θ by training set, use regularization. When plotting training/CV error, do not include regularization.	-- "small" neural network (fewer hidden layers/units) is more prone to underfitting (high bias). -- high bias and high variant can both happen, as regional over/under fit *Build ML system:* 1. start with a simple algorithm 2. plot learning curve 3.Error analysis on CV set (determine if high bias or variance)	*Equations:* $precision = \frac{tp}{tp+fp}$; $recall = \frac{tp}{tp+fn}$ $F1\ score = \frac{2PR}{P+R}$ *Threshold:* For logic regression: $0 \leq h_\theta(x) \leq 1$ If $h_\theta(x) \geq threshold$, predict 1. *Note:* -- increase threshold can improve the precision but trade off the recall	**E.g.** manually feed perfect "text detection" system accuracy can be improved to 89%. Improve "text detection" can bring 17% up Improve "Character Segm" only bring 1% up Component / Accuracy Overall system: 72% → Text detection: 89% Character segmentation: 90% Character recognition: 100%

Special Applications

	Concept/Model	Computation	Notes	Applications
Recommender Systems	Problem Statement: Given output, while input and parameters are not defined Model: Linear regression model Output: Y – ***m x u*** matrix, y_{ij} – user j's rating on movie i (row is move, column is user) R – ***m x u*** matrix, r_{ij} = 1 – user j has rated movie i (otherwise r_{ij} = 0) Parameters: X – ***m x (n+1)*** matrix, x_{ik} – movie i has feature k (i.e. romance, action, etc.) Θ – ***(n+1) x u*** matrix, θ_{kj} – user j's parameter for feature k	Algorithm: Collaborative Filtering Algorithm Cost function: $\min_{\theta,x}\frac{1}{2}\sum\sum_{r(i,j)=1}\left(\left(\theta^{(j)}\right)^T * x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{u}\sum_{k=1}^{n}\left(\theta_k^{(j)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$ Vectorization: $J(\Theta, X) = \frac{1}{2}\sum\sum(X\Theta\odot R - Y\odot R)^2 + \frac{\lambda}{2}\sum\sum\Theta^2 + \frac{\lambda}{2}\sum\sum X^2$ Gradient: $\frac{\partial J(\Theta,X)}{\partial \Theta} = X^T(X\Theta\odot R - Y\odot R) + \lambda\Theta_{column1=0}$ $\frac{\partial J(\Theta,X)}{\partial X} = X^T(X\Theta\odot R - Y\odot R) + \lambda X_{row1=0}$ Find related movies: $\|x(i) - x(j)\| < \varepsilon$	-- In this problem, both parameters x and θ are unknow. Putting both θ and x optimization objective together and combine into a single cost function, instead of optimizing θ and x back-forth in serial (θ -> x -> θ -> x -> …) -- For users who have not rated any movies, use the mean normalization of the training examples (existing users)	# recommend movies/ads for users
Large Scale Machine Learning	*Before deploying expensive ML algorithm:* -- Use relatively small data set -> plot learning curve -> confirm it's high variant -- Understand what's cost to simply gather more data *Map Reduce and data parallelism:* -- Reduce the training set into several subset -- Compute the training set separately (i.e. in different computers or CPU cores) and combine them up	"Batch" gradient descent: $\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ Stochastic gradient descent: *randomly shuffle data set, then* $\frac{\partial J(\theta)}{\partial \theta_j} = (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ "Mini-batch" gradient descent: $\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{b}\sum_{i=1}^{b}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$	-- For stochastic gradient, the learning rate α is typically held constant, we can slowly decrease α over time if we want θ to converge (*E.g.* $\alpha = \frac{const1}{iternationNumber+const2}$)	# online learning (continuous flood of data, compute gradient only on new data, can adapt to user behavior changes)