Linear Regression

Cost Function $J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^i) - y^i\right)^2$ (prediction model)

Gradient Descent repeat:

$w = w - \alpha\frac{\partial}{\partial w}J(w,b) \rightarrow \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^i)-y^i\right)x_j^i$

"bias"/y-intercept value: $b = b - \alpha\frac{\partial}{\partial b}J(w,b) \rightarrow \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^i)-y^i\right)$ (x_j: jth feature)

Gradient Descent(): Cost Function(Model Equation)

$\vec{X}^{(i)}$ = ith training entry, row vector of feature values

x_j = jth feature (column vector)

$x_j^{(i)}$ = jth feature in vector of ith training entry

$$\begin{bmatrix}1 & 3\\ 2 & 5\end{bmatrix}\begin{bmatrix}\ \end{bmatrix} = \begin{bmatrix}\ \end{bmatrix}$$

$f_{\vec{w},b}(\vec{x}) = \vec{x}\cdot\vec{w} + b = [w_1, w_2, w_3, \dots w_n]\cdot[x_1 x_2 x_3 \dots x_n] + b \leq w_1x_1 + w_2x_2 + w_3x_3 + \dots + b$

Vectorized Multiple Linear Regression

Gradient Descent:

Loop, adjust weights:

$w = w - ($ #Compute gradient()
loop each X training entry:
(vector product f(x) prediction − y training entry)
loop each feature:
model add * jth X-entry feature$)$ → gets vector of slopes size n $\left[\frac{\partial J}{\partial w_1}, \dots \frac{\partial J}{\partial w_n}\right]$

updates weights & bias

$b = b - \dots$

Feature Scaling – used to normalize the scaling of parameters to optimize gradient descent (make faster) converge

always use!! types: Mean Normalization: $x_j = \frac{x_j - \mu_j}{\max - \min}$ (μ ← mean) Z-Score normalization: $x_j = \frac{x_j - \mu_j}{\sigma_j}$ (μ ← mean(mu), σ ← standard deviation (lowercase sigma))

aim for $-1 \leq x_j \leq 1$

normalize inputs after creating $\vec{w}$ weights.

normalizing doesn't change cost b/c it's proportional (both X and $\vec{w}$ are scaled)

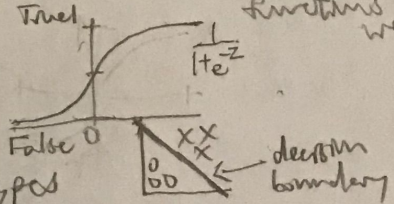
normalize X and W, NOT y training data - will be converted to correct scale.

Feature Engineering – combining features to make new ones

Polynomial Regression: $[x, x^2, x^3, \dots x^n]$ one parameter can be used for other degrees, to make complex functions w/ more weights.

Logistic Regression (classification)

Sigmoid function: $g(z) = \frac{1}{1+e^{-z}}$, $z = \vec{w}\cdot\vec{x} + b \rightarrow = \frac{1}{1+e^{(w\cdot x+b)}}$



Decision Boundary: Line of model between different classification types

Logistic Cost Function

$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{L\left(f_{\vec{w},b}(\vec{x}^i), y^i\right)}_{\text{loss function}} \rightarrow \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^i)\right) & \text{if } y^i = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^i)\right) & \text{if } y^i = 0 \end{cases}$

$J(\vec{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^i\log\left(f_{\vec{w},b}(\vec{x}^i)\right) + (1-y^i)\log\left(1 - f_{\vec{w},b}(\vec{x}^i)\right)\right)$ (same) $= -y^i\log\left(f_{\vec{w},b}(\vec{x}^i)\right) - (1-y^i)\log(1 - f_{\vec{w},b}(\vec{x}^i))$

Gradient Descent(): Loop this:

$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \rightarrow \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^i) - y^i\right)x_j^i$

$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b) \rightarrow \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^i) - y^i\right)$

$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{(\vec{w}\cdot\vec{x}+b)}}$

Cost function not used in gradient descent only for tracking progress

Regularization - for any model, you want it to generalize the accuracy of new input X's
changes values of $\vec{w}$ Problem is if it's too specific for data, it overfits. If too vague, it underfits
Regularization creates a balance

$$J(\vec{w},b)=\frac{1}{2m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^i)-y^i\right)^2+\frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2\right]$$

minimize; mean squared error; fit data; regularization term; keep w_j small; this term balances the trade off.

$f_{\vec{w},b}(\vec{x})=w_1x+w_2x^2+w_3x^3+b$
reduce w_j to reduce overfitting

Gradient Descent, Regularized Linear Regression, same for Logistic Regression, with $f_{\vec{w},b}(\vec{x}^i)=\frac{1}{1+e^{-z}}$

Repeat:

$$w_j=w_j-\alpha\frac{\partial}{\partial w_j}J(\vec{w},b)\rightarrow\left[\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^i)-y^i\right)x_j^i+\frac{\lambda}{m}w_j\right]$$

$$b=b-\alpha\frac{\partial}{\partial b}J(\vec{w},b)\rightarrow\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^i)-y^i\right)$$

Regularized Logistic Regression

$$J(\vec{w},b)=-\frac{1}{m}\sum_{i=1}^{m}\left[y^i\log\left(f_{\vec{w},b}(x^i)\right)+(1-y^i)\log\left(1-f_{\vec{w},b}(\vec{x}^i)\right)\right]+\frac{\lambda}{m}\sum_{j=1}^{n}w_j^2$$

To address overfitting, try to ① Add more training points, ② Apply regularization, or ③ Reduce # of features

Neural Networks

$\vec{x}$ input features → Layer 1 → Layer 2 (output classification) → $a^{[2]}$ ex. .84 Predict category based on %. for $\hat{y}$

Layer 1: $\vec{w}_1^{[1]},b_1^{[1]}$; $\vec{w}_2^{[1]},b_2^{[1]}$; $\vec{w}_3^{[1]},b_3^{[1]}$ → $a_1^{[1]}=g(\vec{w}_1^{[1]}\cdot\vec{x}+b_1^{[1]})$ $g(z)=\frac{1}{1+e^{-z}}$

Layer 2: $w_1^{[2]},b_1^{[2]}$ → $a_1^{[2]}=g(\vec{w}_1^{[2]}\cdot\vec{a}^{[1]}+b_1^{[2]})\rightarrow 0\le a_1\le 1$ Logistic Regression each neurons

$$a_j^{[l]}=g(\vec{w}_j^{[l]}\cdot\vec{a}^{[l-1]}+b_j^{[l]})$$

Activation, any Layer, unit

Forward Propagation - computations "forward", left to right

Dense() → create a layer; Sequential(): create neural net w/ layers and weights W and b

$$A=g(A^T\cdot W+B)$$

A^T [1,2] → 9 params: $\vec{w}_1^{[1]},b_1^{[1]}$ [1,2],3; $\vec{w}_2^{[1]},b_2^{[1]}$ [4,5],6; $\vec{w}_3^{[1]},b_3^{[1]}$ [7,8],9

$$A^{[1]}=\left([1,2]\cdot\begin{bmatrix}1&4&7\\2&5&8\end{bmatrix}+[3\ 6\ 9]\right)_{\text{sigmoid}}$$

sigmoid $(A^T\cdot W+B)$

$1\times2\cdot2\times3+1\times3=1\times3$ $y=z$ $=[8\ 20\ 32]$ → sigmoid

4 params: $w_1^{[2]},b_1^{[2]}$ → pred

[1,2,3],4
13 parameters

Activation Function Types:

Binary Classification: $y=\frac{1}{1+e^{-z}}$ Sigmoid

Regression $y=+/-$ Linear

Regression $y=0$ or $+1$ $\text{ReLU}=\max(0,z)$ rectified linear unit. Use for hidden layers

Softmax Regression - multiclass classification

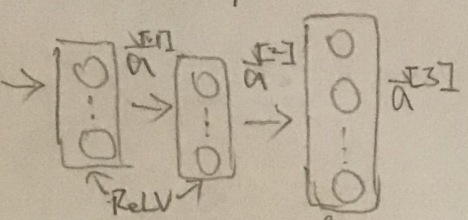
for j, 1-N outputs,

$$z_j=\vec{w}_j\cdot\vec{x}+b_j$$

$$a_j=\frac{e^{z_j}}{\sum_{k=1}^{N}e^{z_k}}\rightarrow P(y=j|\vec{x})=\frac{e^{z_j}}{e^{z_1}+e^{z_2}+\dots+e^{z_N}}$$

$$\text{loss}(a_1,\dots,a_N,y)=\begin{cases}-\log a_1 & \text{if } y=1\\ -\log a_2 & \text{if } y=2\\ \vdots\\ -\log a_N & \text{if } y=N\end{cases}$$

Softmax Output



$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]}$ $\quad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \dots + e^{z_{10}^{[3]}}} = P(y=1|\vec{x})$

$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]}$ $\quad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \dots + e^{z_{10}^{[3]}}} = P(y=10|\vec{x})$

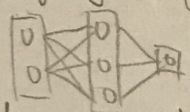
Need to get $\sum_{k=1}^{10} z_k^{[3]}$ first to use for each $a_k^{[3]}$

More numerically accurate version: $L(\vec{a}, y) = \begin{cases} -\log \frac{e^{z_1}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y=1 \\ \vdots \\ -\log \frac{e^{z_{10}}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y=10 \end{cases}$

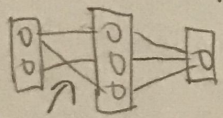
Put on one line, put a value in softmax into loss function $-\log z$, will be more accurate

Multilabel classification - not the same as multiclass. Inputs can have multiple outputs ex. model predicts image with car AND tree

Optimization Algorithm - algorithm advanced version of gradient descent where each parameter w_1 to w_n & b has a different α (learning rate) and keeps changing to get weights to converge faster.

Dense layer:  each activation unit takes an input every unit from previous layer.

Convolutional layer: activation units takes in only select activation units from previous layer - leads to faster computation and reduces overfitting.

ex. 

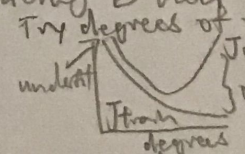
Back Propagation: the use of derivatives to find the weights of the neurons in a neural network, first feeding forward, then getting the partial derivative of each term in the cost function in a computation graph with respect to the intermediate value, propagating left to right to find out how much to nudge the weights for each neuron and layer.

- Train not all examples, keep ~10-20% to test your model, after training model, test the cost by (how well it is) getting cost of unused test examples and compare with
Get cost, split data into 3 parts w/ cross-validation set, then choose best training cost value cost and use untouched test set to test the generalization error.

ML Advice

Train ~60-70% of data, Cross-validation set test which model to use, then use remaining test set to determine how good the model is (3 sets).

1. Split data into training set, cross validation set, and test set.
2. Try many different NN architectures (#units/layers) and train on training set (scale it)
3. Use each model and find cost of cross validation set
4. The model with the lowest cv cost is the most appropriate model that keeps the cost low while generalizing to new examples.
5. Test model on test set to get accuracy.



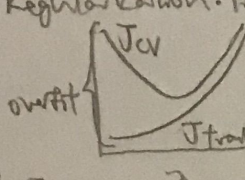
Bias and Variance (Over & Underfitting)

High bias (underfit)
Jtrain high
Jcv high

High variance (overfit)
Jtrain low
Jcv high

Just right:
Jtrain low
Jcv low

Regularization: try values of Lambda



To better the model, try getting data, changing # of features, changing λ, changing model architecture

Compare J_{train} and J_{cv} to baseline level, that is, how well you expect the algorithm to perform (could be how well humans can do task or other algos). Based on the difference of J_{train} and J_{test} to the baseline, then determine if it is overfitting (high variance) or underfitting (high bias) Plot graphs of different models vs. costs and lamba values vs. cost to see if it has high bias or variance of J_{train} and J_{cv}.

Increasing the size of neural networks increases performance, need to regularize, it is more computationally expensive, NNs usually high variance.

Error Analysis - look at examples the model misclassified to diagnose problems - add certain data that can help from error analysis

Data Augmentation - changing/distorting a training example slightly to make more examples and improve the algo.

Data Synthesis - creating artificial data

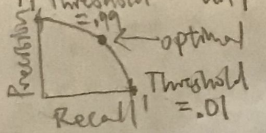
Transfer Learning - use NN parameters from another project (similar) and use those initial weights for model - teaches the model to detect similar patterns

Precision and Recall - if dataset is skewed (more examples that output a certain value), precision/recall table is useful relative to when our model was correct vs when we predicted that class and was wrong, and correctness vs. when we predicted another class but it was actually the original class

Precision: model predicted a class, how accurate were these predictions (1) Recall: ground truth label was this class, how accurate were we for this label?

$$\text{precision} = \frac{\text{true positives}}{\text{all predicted positives}} \qquad \text{recall} = \frac{\text{true positives}}{\text{all actual positives}}$$

Often, there is a tradeoff between precision and recall



Tradeoff is determined by sigmoid functions threshold

To combine precision and recall, use F1 Score $= \frac{1}{\frac{1}{2}(\frac{1}{P}+\frac{1}{R})} = 2\frac{PR}{P+R}$ "Harmonic Mean" P=precision, R=recall — for the case of balancing the two (highest value is better tradeoff)

metric of model's "accuracy"

Regression: compare MSE cost of J_{train} and J_{cv}

Classification: compare % accuracy of J_{train} and J_{cv}

Decision Trees: a tool, tree-like structure that classifies predictions from a set of conditionals for features on how to split.

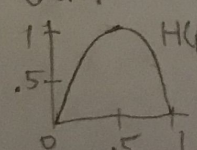
A decision tree must generate many decision nodes (how to split features) and how many. - splits to get 'pure' batches (all 1 class)

Entropy: measure of impurity of data

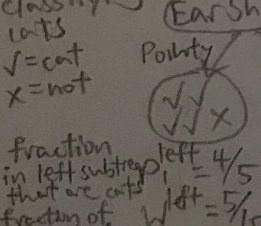
p_1 = fraction of examples with common class $\quad p_0 = 1 - p_1$

50-50 split $H(0.5)=1$ has entropy $=1$, very impure (not pure) very pure split 100-0 or 0-100, entropy $=0$ $H(0)=0=H(1)$ $H(\frac{5}{6})=.65$ "not that impure"

$$H(p_1) = -p_1\log_2(p_1) - p_0\log_2(p_0)$$
$$H(p_1) = -p_1\log_2(p_1) - (1-p_1)\log_2(1-p_1)$$



Information Gain: how much a feature split on a node reduces entropy (purifies data) choose feature split with highest information gain



$p_1^{node} = 5/10$ fraction of positive examples in root

fraction in left subtree that are cats $p_1^{left} = 4/5$ $p_1^{right} = 1/5$

fraction of node on left $w^{left} = 5/10$ $w^{right} = 5/10$

$$\text{Information Gain} = H(p_1^{node}) - \underbrace{\left(w^{left}H(p_1^{left}) + w^{right}H(p_1^{right})\right)}_{\text{Weighted entropy}}$$

One Hot Encoding - split up one feature with k values into k features with values 0 or 1

Ex.

Ear Shape	Round	Pointy	Oval
Round	1	0	0
Pointy →	0	1	0
Oval	0	0	1

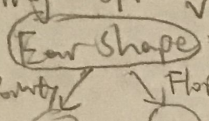
Can also be used for Neural Nets

Continuous valued features: choose a threshold to split data that maximizes information gain.

Regression Trees - decision trees with continuous y-labels.

Ex. Guessing weights of animals

Choosing a Split



Variance root node: 20.51

Weights: 7.2, 9.2, 8.4, 7.6, 10.2
Variance: 1.47
$w^{left} = 5/10$

Weights: 8.8, 15, 11, 18, 20
Variance: 21.87
$w^{right} = 5/10$

Regression Information Gain

$$\text{Variance}^{root} - (w^{left}(\text{variance}^{left}) + w^{right}(\text{variance}^{right}))$$

$$20.51 - (\tfrac{5}{10}(1.47) + \tfrac{5}{10}(21.87))$$
$$= 8.84$$

$\frac{\#\text{ in subtree}}{\#\text{ in root node}}$

Stop splitting when chosen criteria: depth of tree, information gain threshold, # of examples in node, purity of node, or combination

Sampling with Replacement: make training set by choosing random values (keep them in set after choosing) to make multiple, varying training sets for ensemble of trees - duplicates are expected

Random Forest Algorithm:

For b=1 to B decision trees:
Make "new" training set by sampling with replacement all m examples
Train decision tree on new dataset

To make prediction, run it through all generated decision trees and get average output

For Random Forest Algorithm, when choosing split feature, limit options to $k < n$ (try $k = \sqrt{n}$) to diversify # different decision trees

Bagged Decision Tree - same thing but all features can be split randomly (fewer types of generated decision trees than Random Forest)

XGBoost Algorithm - even better decision tree algorithm that prioritizes making decision trees for misclassified examples, also built in regularization and stop criteria, etc

NNs vs. Decision Trees

Neural Networks	Good for Structured & Unstructured data	Slow training	Black Box
Decision Trees	Only good on structured data	Faster training	A bit more interpretability

Clustering - unsupervised learning technique for grouping data into categories

K-means Clustering Algorithm - gets "center of mass/cluster" of k cluster centroids

Randomly initialize k cluster centroids $\mu_1, \mu_2, \ldots, \mu_K$

Repeat until convergence:

Get distance of all points to cluster centroid, assign the cluster w/

loop all points m: c^i = cluster closest to x^i which is $\min_k = (x^i - \mu_k)^2$

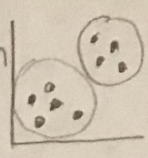
cluster closest to point x^i, $= c^i$ (1 to K)

Move all clusters to center point of points closest to it. (1 to K) for each centroid 1 to K:

μ_k = average in cluster k

$X.shape = m \times n$

(μ_K) centroids.shape $= K \times n$



Cost Function (Distortion function) $\mu_k = \frac{1}{|C_k|} \sum_{i \in C_k} x^i$ → sum of points in cluster over # of points in cluster

$J(c^1, \ldots, c^m, \mu_1, \ldots, \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^i - \mu_{c^i}\|^2$ ← distance

c^i = index of cluster (K clusters) x^i is assigned

μ_k = cluster centroid k

μ_{c^i} = cluster centroid x^i is assigned

K-means optimizes cluster centroids to minimize distortion cost

K-means Random Initialization - initialize cluster centroids at random examples and run it, getting lowest distortion cost

For i=1 to 100: ← 50-1000 iterations of K-means

Randomly initialize K cluster centroids at the training examples

Run K-means to convergence, get $c^1 \ldots c^m$ (cluster #th example belongs to) and $\mu_1 \ldots \mu_K$ (the cluster centroid)

Compute cost function, store it → $J(c^1 \ldots c^m, \mu_1 \ldots)$

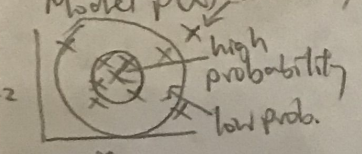
* Pick sets of clusters w/ lowest cost

The number of clusters K to use is ambiguous, can use **elbow method**, but usually depends manually decide for the purpose of the program & tradeoffs.

Anomaly Detection - algorithm to detect unusual inputs deviating from what is expected

Density estimation - probability that input is familiar

Model $p(x)$ anomaly



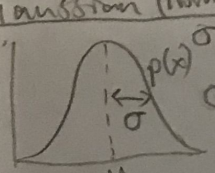
↑ prob = not anomaly

↓ prob = anomaly for some threshold ε epsilon

Examples: manufacturing

Fraud detection, product inspection, fake accounts, monitor data centers

Gaussian (Normal) Distribution - bell shaped curve with mean μ, variance σ^2



$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$

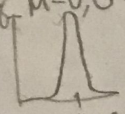
μ = mean

σ = std deviation

μ changes horizontal shift

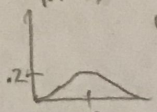
σ changes width

constants → sigmas



Area under curve = 1 b/c probabilities must sum to 1.

probabilities of examples to determine NOT anomalies $p(x)$



How to determine μ and σ: Called maximum likelihood estimation

$\mu = \frac{1}{m} \sum_{i=1}^{m} x^i$ ← mean

$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^i - \mu)^2$

Density Estimation - get probability that an example is "normal" with n features

Training set: $m \times n$ $[\vec{x}^1, \vec{x}^2, \ldots, \vec{x}^m]$

$p(\vec{x}) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdot \ldots \cdot p(x_n; \mu_n, \sigma_n^2)$ → probability of each feature across all examples in being "normal" multiplied together → probability this example is "normal"

Distribution for example w/ n features $= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$ ← multiply

Anomaly Detection Algorithm

1. Choose n features for training example that might show anomalies
2. Fit parameters $\vec{\mu}$ $(1 \times n)$ and $\vec{\sigma}^2$ $(1 \times n)$

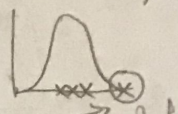
$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m} \vec{x}^{(i)} \rightarrow [\mu_1, \mu_2, \ldots \mu_n] \qquad \vec{\sigma}^2 = \frac{1}{m}\sum_{i=1}^{m} (\vec{x}^{(i)} - \vec{\mu})^2 \rightarrow [\sigma_1, \sigma_2, \ldots, \sigma_n]$$

3. Compute $p(x)$

$$p(\vec{x}) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(\frac{-(x_j - \mu_j)^2}{2\sigma_j^2}\right) \rightarrow$$ probability of 1 example not being an anomaly

4. If $p(\vec{x}) < \varepsilon$, is anomaly

Algorithm punish heavily $p(\vec{x})$ if any features have low probability b/c it multiplies all n probs, making $p(x)$ ↓.



heavy punished

Real-number evaluation

- test how well the algorithm is working, making changes:

If you have some labeled data of anomalous and non-anonymous, do this:

split into 3 sets

Training set: $[\vec{x}^1, \ldots \vec{x}^m]$ assume normal examples

Cross validation set: $[(\vec{x}^1_{cv}, y^1_{cv}), \ldots (\vec{x}^{m_{cv}}_{cv}, y^{m_{cv}}_{cv})]$

Test set: $[(\vec{x}^1_{test}, y^1_{test}), \ldots (\vec{x}^{m_{test}}_{test}, y^{m_{test}}_{test})]$

} include same # of some anomalous examples, labeled.

← in CV set, tune ε and x_j features

Alternative: no test set, just training set and CV set ← with anomalies, tune ε and x_j.

Use if we don't have many anomaly examples – but can't test, may overfit

- Consider applying evaluation metrics ① true pos, false pos, ... ② precision/recall; ③ F1-Score

precision = $\frac{\#\text{predicted 1, was right}}{\#\text{model predicted 1}}$ b/c dataset is often skewed w/ anomaly detection

recall = $\frac{\#\text{predicted 1, was right}}{\#\text{ground truth label } y \text{ was } 1}$

F1-Score = $2\frac{P \cdot R}{P + R}$ prediction accuracy $[0,1]$

Anomaly Detection VS. Supervised Learning

Anomaly Detection	Supervised Learning
- Small # positive examples (0-20), large # negative ($y=0$) examples	- large # positive and negative examples
- if you expect many new types of anomalies in the future, this will detect examples supervised learning might not catch.	- Algo detects examples similar to what it has seen before, good if next example anomalies are expected to be similar
Ex Unique forms of fraud*	Ex spam* = similar common spam emails
*New manufacturing defects	manufacturing defects - scratches
*Monitoring machines in data center	* Predict weather (sunny, rainy, etc)
	* Disease classification (known diseases)

Choosing What Features to Use

distribution of values for a feature

It is helpful to make the features more Gaussian (bell-shaped) by using log or some function of x_j

Error analysis – look at incorrect examples, change features (or add), or combine features

Recommender Systems

$n_m \times n_u$

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)	x_1 (romance)	x_2 (action)
Love at Last	5	5	0	0	0.9	0
Romance Forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. Karate	0	0	5	?	0	0.9

$n_m = 5$ #movies

$n_u = 4$ #users

$n = 2$ #features

$r(i,j) = 1$ if user j has rated movie i (otherwise 0)

$y^{(i,j)}$ = user j rating on i

w^j, b^j = user j parameters

x^i = feature vector for movie i

m^j = #movies user j rated

Predict user j's rating for movie i as $y^{(i,j)} = w^j \cdot x^i + b^j$

$x^i = 1 \times n$ $w = n \times 1$

$x^3 = [0.99\ 0]$ $w^1 = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$ $b^1 = 0$

$w^1 \cdot x^3 + b^1 = 4.95$

Goal: learn w^j, b^j

Cost Function for 1 user (given features get weights)

$$J(\vec{w}^j, b^j) = \frac{1}{2}\sum_{i:r(i,j)=1}^{m^j} (\vec{w}^j \cdot \vec{x}^i + b^j - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{k=1}^{n}(w_k^j)^2$$

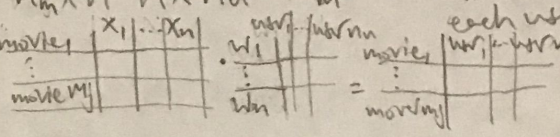
choose w and j to minimize cost btwn actual rating and predicted

Learn w for all users: only rated movies

$$J(\vec{w}^1,\dots,\vec{w}^{n_u}, b^1,\dots,b^{n_u}) = \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}^{m^j} (\vec{w}^j \cdot \vec{x}^i + b^j - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(w_k^j)^2$$

sum → $1 \times n_u$ cost on all movies for each user

$n_m \times n \cdot n \times n_u = n_m \times n_u$



If we don't have features $x_1 - x_n$, we can find them if we know weights w and b.

ex. $w^j \cdot x^i + b^j \rightarrow x^i = [1\ 0]$ → find features for each movie i

Cost function for 1 movie i given weights (features x^i)

$$J(\vec{x}^i) = \frac{1}{2}\sum_{j:r(i,j)=1}^{m^j} (\vec{w}^j \cdot \vec{x}^i + b^j - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{k=1}^{n}(x_k^i)^2$$

Use models predicted to find out whether or not to recommend a user something.

Features all movies

$$J(\vec{x}^1,\dots,\vec{x}^{n_m}) = \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}^{m^j} (\vec{w}^j \cdot \vec{x}^i + b^j - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(w_k^j)^2$$

Collaborative Filtering - recommend items based on users' ratings collaboratively

- data of multiple users' ratings to predict features for movies, w and b, must learn more features and user parameters
- x to predict user ratings

Cost put together:

$$J(w,x,b) = \frac{1}{2}\sum_{(i,j):r(i,j)=1} (w^j \cdot x^i + b^j - y^{(i,j)})^2 + \underbrace{\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(w_k^j)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(w_k^i)^2}_{\text{regularization}}$$

Gradient Descent

repeat:

$$w_i^j = w_i^j - \alpha\frac{\partial}{\partial w_i^j}J(w,b,x)$$
$$b^j = b^j - \alpha\frac{\partial}{\partial b^j}J(w,b,x)$$
$$x_k^i = x_k^i - \alpha\frac{\partial}{\partial x_k^i}J(w,b,x)$$

parameters are w, x, b

Binary Labels - use sigmoid function % user liked it, binary classification

Ratings mean 1 - user engaged, 0 - not engaged, ? - not shown

$f_{(w,x,b)}(x) = g(w^j \cdot x^i + b^j)$, $g = \frac{1}{1+e^{-z}}$

Binary Loss: $L(f_{(w,b,x)}(x), y^{(i,j)}) = -y^{(i,j)}\log(f_{(w,b,x)}(x)) - (1-y^{(i,j)})\log(1-f_{(w,b,x)}(x))$ loss for 1 rating

$$J(w,b,x) = \sum_{(i,j):r(i,j)=1} L(f_{(w,b,x)}(x), y^{(i,j)})$$

Mean Normalization - needed b/c new users would have ratings 0. Want "initial" rating prediction

change y to $y - \mu$ ← row mean of y. So, for new user with $w=0=b$, with new $f_{(w,x,b)}(x)$, predictions $= \mu$ → average of other users' ratings for new user's initial ratings.

change to $f_{(w,x,b)}(x) = w^j \cdot x^i + b^j + \mu_i$ ↑ *can use column mean in some cases

so prediction is same

Finding Related Items - how related are items to one?

Got feature vectors, smallest distance is item k and item x^i (similar)

$$\sum_{l=1}^{n}(x_l^k - x_l^i)^2 \quad \text{also} = \|x^k - x^i\|^2$$ smallest val is most related to x^i.

Limitations

- tough for new items w/ few user ratings *cold start - also for new users hard to recommend
- doesn't use additional site information about user (only the rating)

Collaborative Filtering
- Recommend items to you based on other users' ratings

VS.

Content-based Filtering
- Recommend items to you based on user features and item (including ratings) to find good matches

Content-based Filtering

$\vec{x}_u^j$ user features for user j ex. age, movies watched, ratings, ...

$\vec{x}_m^i$ movie features for movie i ex. year, genre, reviews, ratings

Predict rating user j on movie i:

$\vec{v}_u^j \cdot \vec{v}_m^i$

$\vec{v}_u^j$ computed from $\vec{x}_u^j$ features; $\vec{v}_m^i$ computed from $\vec{x}_m^i$ features

shape of v_u and v_m must be same

= convert user & movie feature into vectors w/ same shape

vectors - use neural networks "deep learning"

Getting v_u and v_m

$x_u \to v_u$ "users" network

ex

$\vec{x}_u^i \to$ [128] $\to$ [64] $\to$ [32] $\to \vec{v}_u^j$

$x_m \to v_m$ "movies"/items network

$\vec{x}_m^i \to$ [256] $\to$ [128] $\to \cdots \to$ [32] $\to \vec{v}_m^i$

1 user j on 1 movie i

Prediction: $\vec{v}_u^j \cdot \vec{v}_m^i$

can use sigmoid for classification: $g(\vec{v}_u^j \cdot \vec{v}_m^i)$

NN chooses $\vec{v}_u^j$ & $\vec{v}_m^i$ to minimize cost function (how far from actual y rating) from selected features x_u^j & x_m^i

Cost function

$$J = \sum_{(i,j):r(i,j)=1} (\vec{v}_u^j \cdot \vec{v}_m^i - y^{(i,j)})^2 + \text{NN regularization term}$$

$\vec{v}_u^j$ is vector length 32, represents user j w/ features x_u^j

$\vec{v}_m^i$ is vector length 32, represents movie i w/ feature x_m^i

Find Similar Items: $\|v_m^k - v_m^i\|^2$ ← compare movie k and movie i; small distances, more similar

*This can be pre-computed (overnight?), so each movie has similarity score for each movie ($n_m \times n_m$) for quick access; the top similar movies to a given movie

Recommending movies from large catalogue: ① Retrieval, ② Ranking

Retrieval - generate large list of possible items *trade off w/ more items → ↓ performance vs. fewer for ↑ speed

ex.
1. For last 10 movies user watched, pick 10 similar movies for each
2. Pick 10 movies from 3 most viewed genres users
3. Top 20 movies in country

prune out items

- then filter list; remove duplicates, items already watched

*analyze how many retrieved items to have high prob. recommendations

$\|v_m^k - v_m^i\|^2$ pre-computed?

$p(y^{(i,j)}) = 1$

Ranking - use the list (~100 items) and rank items with trained model

- order by highest rating

$x_u \to$ [ ] $\to$ [ ] $\to$ [ ] $\to v_u \odot \to$ prediction

$x_m \to$ [ ] $\to$ [ ] $\to$ [ ] $\to v_m$

* can pre-compute v_u and v_m for each user and movie and just do $v_u \cdot v_m$ to get prediction quickly

Ethics

Different ways to configure recommender systems:
- movies that users will rate higher ✓
- products user will purchase ✓
- ads optimizing for clicks ⚠
- generating most profit ⚠
- optimize for max watch time ⚠

⚠ can be harmful → not serving customer's needs/wants, results may not be relevant

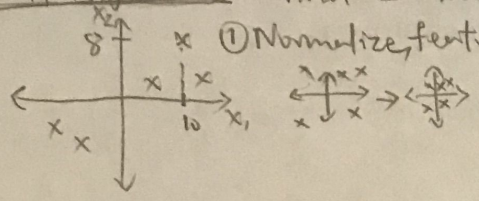
transparency

Principal Components Analysis (PCA) – reduce # of features to be easier to visualize

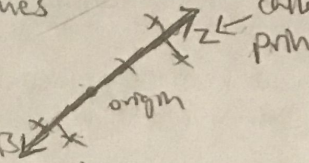
ex. features $x_1, x_2, \ldots, x_{100} \rightarrow$ new features z_1, z_2 — combine (lose some info)

many features → 2 or 3 features

PCA Algorithm – find z axes that reduces # features and captures data



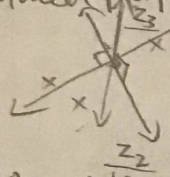
① Normalize, feature scaling values — uses new z-axes



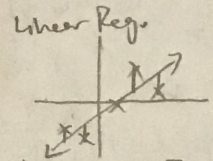
called principal component

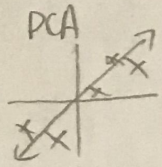
② Choose z-axis that minimizes (or maximize variance) distance from line to point

- each new principal component (z-axis) is placed perpendicular to others



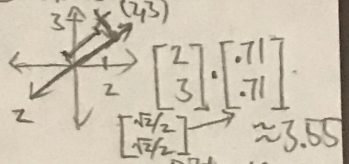
3 principal components





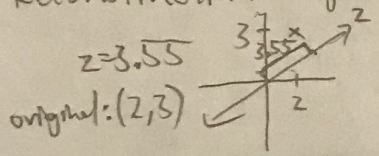
- PCA is NOT linear regression
  - linear regression minimizes distance along y-axis, whereas PCA treats all features x equally and reduce # of axis and maximize variance of points

How distance from point to z-axis is calculated



Distance from origin to perpen. point intersecting w/ example

"Reconstruction" – given z, reconstruct initial features



$3.55 \times \begin{bmatrix}\sqrt{2}/2\\\sqrt{2}/2\end{bmatrix} = \begin{bmatrix}2.52\\2.52\end{bmatrix}$ → estimates (2.52, 2.52)

can't get original features, only can estimate

Implementing PCA

① perform feature scaling
② "fit" data to obtain 2 or 3+ principal components
③ examine how much variance the principal components have (how much original data is preserved)
④ transform data onto new axes → # features will be reduced, can plot now

PCA used for visualization mainly (old), can be used for data compression or speeding up training models

Reinforcement Learning – training model to make decision w/ max reward, given states (s) of the current environment → action of model

(s, a, R(s), s') — for ambiguous contexts (+1, -10)

state, action, reward, new state

"stop"/done

return = $R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots$ (until terminal state)

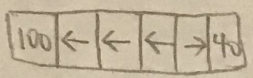
return depends on reward & discount factor

Discount factor $\gamma = 0.9$ — depends on actions model takes

Policy $\pi(s) = a$ — function that maps from a state to actions

Goal of reinforcement learning is to find policy π that tells the model what action a to take given state s.

Markov Decision Process (MDP) - future only depends on current state, nothing else

	Mars Rover	Helicopter	Chess
states	6 states	position of helicopter	position on board
actions	← →	how to move control stick	possible move
rewards	100, 0, 40	+1, -1000	+1, 0, -1
discount factor γ	0.5	0.99	0.995
return	$R_1+\gamma R_2+\gamma^2 R_3+\dots$	$R_1+\gamma R_2+\gamma^2 R_3+\dots$	$R_1+\gamma R_2+\gamma^2 R_3$
policy π		Find $\pi(s)=a$	Find $\pi(s)=a$

State-Action Value Function (Q Function)

$Q(s,a)$ = Return after taking action a from state s and taking optimal action after

The action that yields the highest return at a given state is the optimal action

- best return from state s is $\max_a Q(s,a)$ and a is the best action

Q^* = optimal Q function - get Q for all states and actions

Bellman Equation

s = current state
a = current action
s' = state after taking action a
a' = action you take in state s'

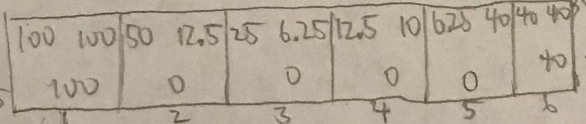
R(s) = reward in current state

Return from behaving optimally from state s'

$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$

immediate reward

Return from current state and Q function return of state after taking action a over all possible actions a'.



$\gamma = 0.5$

Terminal state: $Q(s,a)=R(s)$ last state is just reward

Ex. s=2, a=→, s'=3

$Q(2,\rightarrow) = R(2) + 0.5 \max_{a'} Q(3,a')$

$= 0 + 0.5(25) = 12.5$

s=4, a=←, s'=3

$Q(4,\leftarrow) = R(4) + 0.5 \max_{a} Q(3,a')$

$= 0 + 0.5(25) = 12.5$

$Q(4,\leftarrow) = 0 + 0.5(0 + 0.5(0 + 0.5(100)))$

$= .5(.5(.5(100))) = 12.5$

In other words,

$Q(4,\leftarrow) = R(4) + \gamma(R(3) + \gamma(R(2) + \gamma(R(1))))$

Or more simply

$Q(4,\leftarrow) = R(4) + \gamma R(3) + \gamma^2 R(2) + \gamma^3 R(1)$

MDP

Random (stochastic) Environment

- sometimes the model may take actions that aren't optimal due to the environment, so actions may be random
  ex. 90% chance it performs action, 10% other

So, across multiple tests, there may be different returns

Then, have to maximize average value of return

Expected Return = Average$(R_1+\gamma R_2+\gamma^2 R_3+\dots)$

$= E(R_1+\gamma R_2+\gamma^2 R_3+\dots)$

Goal: choose policy $\pi(s)=a$ to maximize average returns

Modified Bellman Equation: $Q(s,a) = R(s) + \gamma E(\max_{a'} Q(s',a'))$

immediate reward; gamma; expected returns

Continuous State Spaces – oftentimes, an agent can have many continuous features describing its state, not just a set # of states

Ex. Continuous state features for...

car:
- x x-position
- y y-position
- θ angle direction
- $\dot{x}$ x-velocity
- $\dot{y}$ y-velocity
- $\dot{\theta}$ θ-velocity

helicopter:
- x x-position
- y y-position
- z height
- roll left/right tilt
- pitch forward, backward tilt
- yaw direction

(rate of change: $\dot{x}$, $\dot{y}$, $\dot{z}$, roll, pitch, yaw)

States usually have many features ranging many values

"Continuous state MDP"

Example: Lunar Lander – land LL safely w/ reinforcement learning

Actions:
- nothing
- left thruster
- main thruster
- right thruster

States:
- x x-position
- y y-position
- $\dot{x}$ velocity
- $\dot{y}$ velocity
- θ tilt
- $\dot{\theta}$ angular velocity
- l left grounded
- r right grounded

(l, r: 0 or 1)

Rewards:
- Get to landing pad +100 – +140
- Reward for moving away/toward the pad
- Crash −100
- Soft landing +100
- Leg grounded +10
- Fire main engine −0.3
- Fire side thruster −0.03

Goal: learn policy $\pi(s) = a$ and pick action a that maximizes return

$\gamma = 0.985$

Calculating $Q(s,a)$ – State-action value function (return)

Input state and action, output return Q pred. w/ neural net

$$\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{matrix} \} s \\ \} a \end{matrix} \rightarrow \boxed{0 \\ \vdots \\ 0}_{64} \rightarrow \boxed{0 \\ \vdots \\ 0}_{64} \rightarrow \boxed{0}_{1} \quad Q(s,a) = y$$

a is one hot encoded "do nothing" action

12 inputs

Here, compute all actions a at state s:
$Q(s,\text{nothing})$; $Q(s,\text{left})$; $Q(s,\text{main})$; $Q(s,\text{right})$
Pick action a that maximizes $Q(s,a)$

To train the network, we need training examples $x = (s,a)$ and labels $y = Q(s,a)$

We use Bellman Equation to get data (x,y) that we can use to train neural network to predict return Q of state and action

$$\underbrace{Q(s,a)}_{x} = \underbrace{R(s) + \gamma \max_{a'} Q(s',a')}_{y}$$

$(s, a, R(s), s')$ ← tuple

Training examples

x	y
$x^1 = (s^1, a^1)$	$y^1 = R(s^1) + \gamma \max_{a'} Q(s^1, a')$
$x^2 = (s^2, a^2)$	$y^2 = R(s^2) + \gamma \max_{a'} Q(s^2, a')$
⋮	⋮
x^{10000}	y^{10000}

size: 8 4

ex. Try random s and a's and store the reward R and next state s'

$(s^1, a^1, R(s^1), s^1)$
$(s^2, a^2, R(s^2), s^2)$
$\vdots$
$(s^{10000}, a^{10000}, R(s^{10000}), s^{10000})$

used to create training data

initially don't know Q function exact

Learning Algorithm to get Q Function (Deep Q-Network (DQN)) – model that predicts return Q(s,a) for any s,a

Initialize neural network with random weights for Q(s,a)

Repeat, improving model: w/ 3-Greedy Policy

1. Take random actions (s,a,R(s),s') and store most recent 10,000 tuples ← Replay buffer or "experience replay"
2. Train neural network: copy old weights to Qnew NN, use mini-batch on Qnew & keep improving it
   Create training set of 10,000 examples using assume Q NN ≈ Q* optimal Q
   $x=(s,a)$ and $y=R(s)+\gamma \max_{a'} Q(s',a')$ or $y=R(s)$ if terminal state
   Bellman Equation ← calculated w/ Qnew neural network w/ inaccurate params – Q will be inaccurate but improves initially from training NN over time
   Train NN with dataset, Qnew w/ $MSE(y_{new},Q)$
   Q=Qnew w/ ε-Greedy policy
   Use new, improved model, keep training with slightly better model Q

$Q=f_{w,b}(x)\approx y$ predicts return, y on 1 state

Architecture

NN Improvement – inefficient to run inference for Q for each action. Instead, simultaneously predict Q for all actions.

$\vec{x}=[s]=[x\ y\ x\cdot y\ x^2\ y^2\ \dots]$ → 0…0 (64) → 0…0 (64) → 4 outputs: Q(s,nothing), Q(s,left), Q(s,main), Q(s,right)

8 inputs

Only input s and output all possible actions Q-values

Then, pick a that maximizes Q(s,a)

Now, it's more efficient to get Bellman: $Q(s,a)=R(s)+\gamma \max_{a'} Q(s,a')$ and a is given sometimes or faster to get max

ε-Greedy Policy – when create training data, we said to "take random actions", how do we do that?

← usually choose optimal action based on "inaccurate" model, but also let it try some

Because we randomly initialized parameters in NN, model initially predicts Q terribly and may not explore different actions.

So, ① with probability 0.95 choose action a maximizing Q(s,a) "Greedy, Exploitation"

(ε=0.05) ② with probability 0.05 pick random action – "Exploration"

*When model starts w/ terrible predictions, start ε high. Then gradually decrease ε

ε = probability to make random action

ε = 1.0 → 0.01

Algorithm Refinements: Mini-batch & Soft Updates

Mini-batch – regular gradient descent is slow b/c you sum all m examples for each step → can be ex. 100,000,000

– group examples in subset/"batches" m', and each iteration of training, only use one subset → also mini-batch is used more often

– converges slow but way fast computationally

Instead of using 10,000 examples, use 1,000 each iteration – faster!

Soft Update – Qnew (newly trained NN) can by chance be worse than Q (existing NN), so we don't want to completely overwrite it (maybe Q has some great weights)

Q=Qnew

keep some old params, change some to new

$W=0.01W_{new}+0.99W$

$B=0.01B_{new}+0.99B$

can set these #s

no good → W=Wnew, B=Bnew

Gradually converges to parameters for Q and is more reliable

State of Reinforcement Learning

Limitations
- way easier to work in a simulation than real world (on pc vs physical robot)
- fewer applications than supervised and unsupervised learning for
- is sick, research is being done, could have more applications in future

Complete DQN Algorithm

create memory_buffer to store experiences, initialize ε, # steps to update NN
initialize 2 neural nets Q and Qnew w/ same random weights
repeat each full simulation "episodes" _ times:
  reset environment, keep initial state
  for each time step, repeat:
    pass state to Q NN to predict action, use ε-Greedy Policy to choose action
    take action in environment, store new state, reward, done
    store experience tuple in memory_buffer
    if condition to update (train) is met: (only every few steps, subset of experiences accumulated)
      choose mini-batch of random experiences to use to train Q NN
      // train NN w/ both Q and Qnew NNs using the mini-batch data
      adjust weights of Q NN w/ loss of Q prediction and Qnew = Bellman Equation,
      Q NN trained w/ mini-batch gradient descent ↳ = R(s) if terminal state,
      use Soft Update to update Q NN weights with parts or R(s) + γ max_a Qnew(s', a') otherwise
      of weights of Q and Qnew
      after updating Q NN, set Qnew weights to Q to prep for next training
    set state to next state
    (keep repeating until episode ends)
  update ε value

Done!

Topics Gone Over

Supervised Machine Learning: Regression and Classification
- Linear regression, logistic regression, cost function, gradient descent

Advanced Learning Algorithms
- Neural networks, decision trees, ML advice

Unsupervised Learning, Recommenders, Reinforcement Learning
- Clustering, anomaly detection, collaborative filtering, content-based filtering, reinforcement learning