Machine Learning ~~Midterm~~ Final Collaborative Study Guide

Note!!!

- I (who will remain anonymous) really hope everyone can benefit from this study guide!
- Please comment if you feel like some information may be wrong but you're hesitant to change it. (If you're confident it's wrong, then *change it.*)
- PLEASE CONTRIBUTE A LITTLE if you're gonna use it``
- ***https://www.cc.gatech.edu/~bboots3/CS4641-Fall2018/Lecture23/Final_Review.pdf***
- Final Midterm study guide:
- Feel free to make new outline topics according to the one prof said in class
- Useful Comparison chart for different algorithms: https://www.dataschool.io/comparing-supervised-learning-algorithms/
- ***excellent youtube videos:*** search " μ" and then a topic ("pca", "svm", etc)
- Happy studying =) <3 <3 <3
- https://www.youtube.com/channel/UCTmAYxRV7H9NTdgC9bNixvw
- i'm so exhausted from studying from this final :(

Table Of Contents

Similarities / Differences Between Algorithms

- Decision tree - local
  - Bias: prefer small decision trees Y
    - High bias with small decision trees (underfit possible)
    - low bias with large decision trees but higher variance (because overfit possible)
  - Search algorithm: greedy (always makes the choice that seems to be the best at that moment)
  - Heuristic function: information gain (choose split with maximum change in entropy)

        - Entropy is the disorderliness, best features are chosen based off how much it can decrease randomness and get closer to 50/50.
    - Overfitting solution: pruning
        - Pre-pruning: use validation set to determine when to stop growing the tree or just stop growing "when data split is not statistically significant" (Lect 3 Slide 8)
        - Post-pruning - After building decision tree, remove nodes that decrease performance on validation set

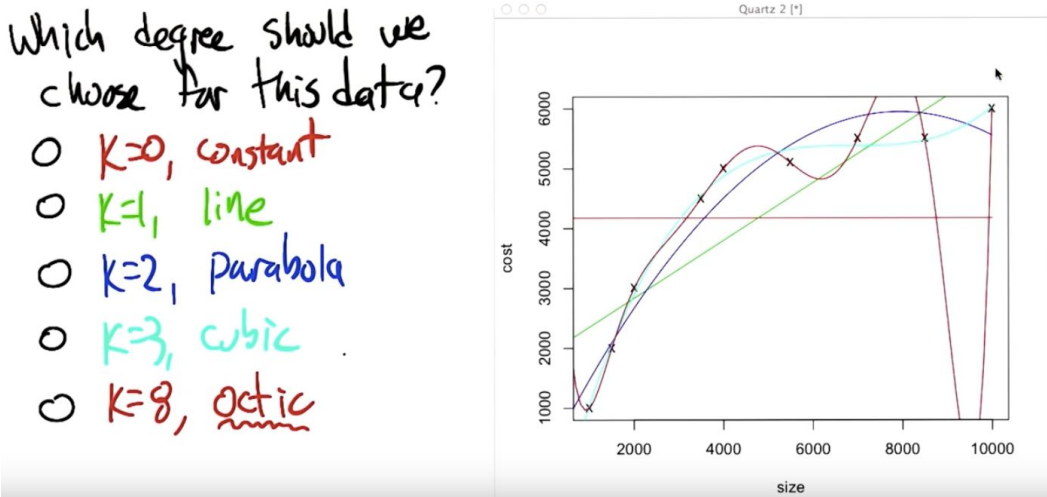
    Shallow vs Deep Pruning -
    - On datasets:
    - Constructing a decision tree is to find all the attributes with highest information gain.
    - Strengths include:
        - Can model Any Boolean functionμ
        - Fast and simple to implement
            - Testing time is O(log n)
        - Can convert to rules – Handles noisy data
        - can learn non-linear relationships
        - fairly robust to outliers
            - **Because outliers don't impact the Information Gain enough to create an erroneous split that might overfit the data**
    - weaknesses include:
        - Univariate splits/partitioning using only one attribute at a time --- limits types of possible trees
            - **Restriction Bias**
        - Large decision trees may be hard to understand
            - **More complex decision trees may be overfit**
        - Requires fixed-length feature vectors (since adding a new feature would result in re-building the tree completely)
            - **This is not true for Naive Bayes because it's based on probabilities rather than looking at the dataset itself**
            - **Naive Bayes assumes features are independent of each other, so adding a new feature, just requires an iteration to calculate P(new feature|class) instead of recalculating**
        - Non-incremental (i.e., batch method) (since adding a new instance would result in re-building the tree completely)

- Decision stump
    - Def: one-level decision tree with one internal node (the root), which is immediately connected to the terminal nodes (its leaves)
    - Weak learner
        - So you can use them in ensemble learning
    - High Bias (doesn't fit training data very well)
    - Low variance (doesn't overfit to training data, model is fairly replicable)

- Linear regression (gradient descent) - global

- Fights overfitting by using regularization
- Heuristic function: Mean squared error
- Algorithm: Gradient descent/Calculus/Anything that helps optimize the heuristic function.
  - Gradient descent requires a convex function to guarantee that a random point converges to the absolute minimum; this can be guaranteed by careful selection of the cost function
- Linear regression must have a global minimum.
- Overfitting/variance: combated with regularization
- Should be the first algorithm you try when using ML.
- Least squares linear regression - convex

- Polynomial regression - global
  - Finding coefficients: w ≈ $(X^T X)^{-1} X^T y$
  - Order of polynomial: (k <= n - 1, where n is the number of data points; k is degrees freedom)



  - The best choice above is K=3. Since it does not overfit and it does a good job of estimating all the points. k=8 heavily overfits the data and would be unlikely to accurately predict test data.

- the perceptron - global
  - Decision Boundaries are hyperplanes that split the region into two regions where one is classified "Yes/Positive" and the other is classified "No/Negative"
  - No vertical decision boundaries!
  - If data is linearly separable, then perceptron will *converge* to global optimum. If data is not linearly separable, then perceptron will fail completely in its learning - will reach a local optimum.
  - While the perceptron algorithm is guaranteed to converge on some solution in the case of a linearly separable training set, it may still pick any solution and problems may admit many solutions of varying quality. The perceptron of optimal stability, better known as the linear support vector machine, was designed to solve this problem
- logistic regression - global
  - Heuristic function: Maximum likelihood estimation (log loss)
  - Uses regularization to fight bias

    - Calculates the **probability** of an event happening
    - Log regression model is a sigmoid function
    - $$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^{\mathsf{T}}\boldsymbol{x}}}$$
    - Softmax for multi-class problems (multiple one vs. rest classifiers)

- neural nets - local
    - Very expressive, can express arbitrary functions
    - Goes to a local optima using optimization algorithm such as gradient descent
        - GD tends to work well in practice
        - delta error backpropagation
    - Prone to overfitting - needs large training dataset to generalize well
    - Slower to train
    - Faster to query
    - Can be an offline learner: does all the learning before queries
    -
- 1NN (1 Nearest Neighbor)
    - No training error (no training at all, just storing)
    - Very strongly fit to training data - high variance, low bias
    - Lazy learner: “procrastinates” and does all the learning when queried
    - shatter infinite vc-dimension
- KNN (K Nearest Neighbor)
    - Extremely fast to “train” (O(1))
    - Slow to query, O(log (n) + k) - since it builds general model only *upon* query (ie., lazy learner)
        - Naive approach even slower
    - Example of instance based learning
    - Simple, intuitive, and works well in practice
        - Generally requires strong domain knowledge for good selection of a distance metric and value of k
- Ensemble
    - On average performs the best out of all SL-ML algorithms.
    - Bootstrapping
        - Sort of like cross validation
        - Take a random subset of your training examples (with replacement) to train and then test it on the remaining examples (as validation set).
    - Bagging
        - Use bootstrapping to train.
        - Train a classifier for your bootstrap replicates.
        - Estimate performance on your out-of-bootstrap data (validation set)
        - Average the output of all classifiers to come up with a hypothesis
        - Reduces variance

- Boosting
  - If I have some examples that I haven't performed well on, reweight my examples to **make the misclassified ones more important.**

1: Initialize a vector of n uniform weights $\mathbf{w}_1$
2: **for** $t = 1, \ldots, T$
3: Train model h_t on X, y with weights $\mathbf{w}_t$
4: Compute the weighted training error of h_t
5: Choose $\beta_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
6: Update all instance weights:

$$w_{t+1,i} = w_{t,i} \exp\left(-\beta_t y_i h_t(\mathbf{x}_i)\right)$$

7: Normalize $\mathbf{w}_{t+1}$ to be a distribution
8: **end for**
9: **Return** the hypothesis

$$H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \beta_t h_t(\mathbf{x})\right)$$

  - Over every timestep, we:
    - Train the model and come up with a distribution (a line or decision stump, typically) that divides the plane into positive and negative examples)
    - Find a weak classifier for the hypothesis of that timestep
      - A weak classifier is a classifier than can always predict *something* better than just guessing. Ideally, though, we want this weak classifier to produce small error (noted by ε)zero
  - d instances.

  - Understanding the pseudocode:
    - β_t will always be positive because error ε is always between 0 and 1
      - `Taking the natural log of $\frac{1-\varepsilon}{\varepsilon}$ will always produce something positive if ε is between 0 and 0.5
    - Consider the terms in the exponent of the weight update step
      - $-\beta_t y_i h_t(\mathbf{x}_i)$
      - Both y_i and h_t are values either -1 or 1.
      - So, if the example is correctly classified, this portion will be $-1 * -1$ or $1 * 1$, so always = 1. If it is misclassified, it will be -1.
        - This just determines the sign of this exponent based on whether the example was classified correctly
      - So, since the β_t is given a negative sign, we can see that this exponent will be negative if the example is correctly classified and it will be positive if the example is misclassified.
      - marginalization
  - We then return our final hypothesis at the end
  - STRENGTHS OF BOOSTING:

- Fast and simple to program
- No hyperparameters to tune (besides T)
- No assumptions on the weak learner
- Always drives down the test error even after the training error reaches zero

■ WHEN BOOSTING CAN FAIL:

- Given insufficient data
- Overly complex weak hypotheses
  - If you use complex models (Neural nets, etc.) as the learner
- Can be susceptible to noise
- When there are a large number of outliers

- SVM - global
  - Use SVMs when there are **multiple data points that are close together**. The SVM will help to separate the close together points.
  - If using a linear kernel, your data must be linearly separable
  - Does not overfit
  - Find the maximum margin hyperplane
  - Training time is high, not suited for large datasets
  - Dual Representation makes optimizing easier
    - Can't handle noise very well for a linear svm
  - STRENGTHS -
    1. Find globally best model
    2. Good generalization
    3. Works well with few training instances
    4. Efficient Algorithm
    5. Amenable to kernel trick..

- Naive Bayes

- Finds the probability of predicting particular hypothesis given the data using Bayes rule
- The calculation procedure
- Advantage:
  - Fast to train
  - Fast to classify
  - Not sensitive to irrelevant features
  - Handles real and discrete data
  - Handles streaming data

  Disadvantage:

  -Assume **conditional** independence of feature (ends up being fine in practice)

How will these algorithms perform on various datasets?

Few samples, Few attributes - SVM

1. Many samples, Few attributes - DT

2. Few samples, Many attributes - SVM
3. Many samples, Many attributes - Boosting/NN

Global vs local optima during learning?

Gradient descent will lead to a local optima if the loss function is not convex

Parametric methods (summarizes data with a set of parameters of fixed size):

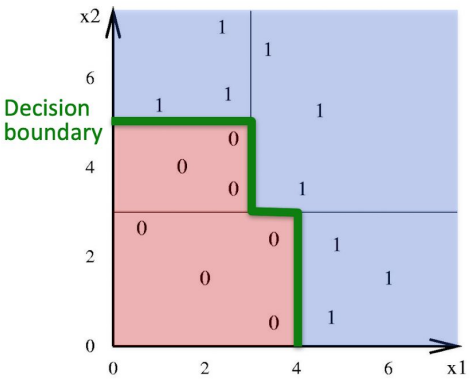
- Logistic Regression
- Perceptron
- Naive Bayes
- Simple Neural Networks
- Linear SVM

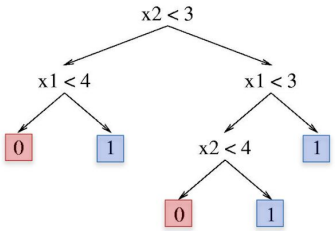
Nonparametric method:

- KNN
- ID3
- Kernel SVM

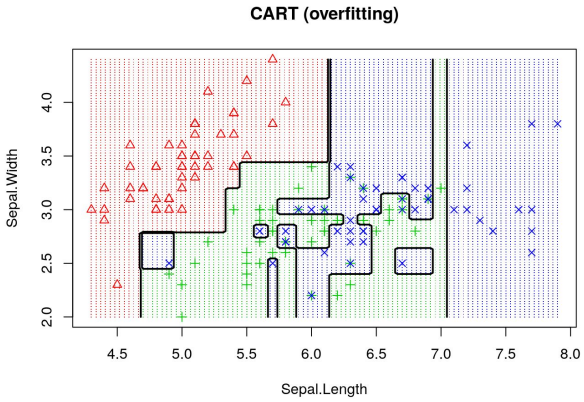
Decision boundaries

- Decision tree

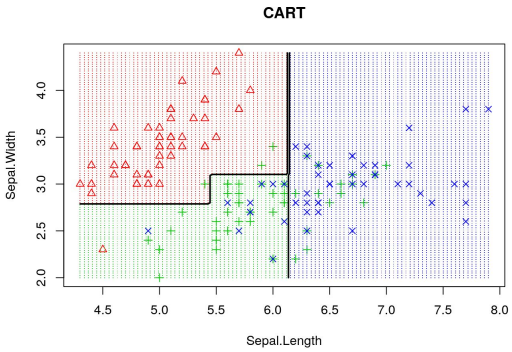




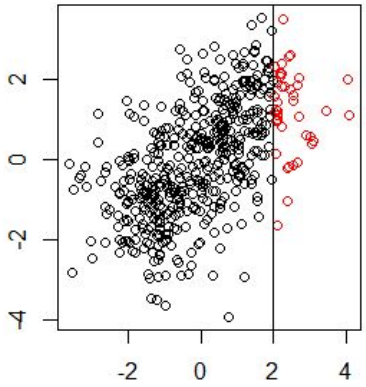
  - Overfit:



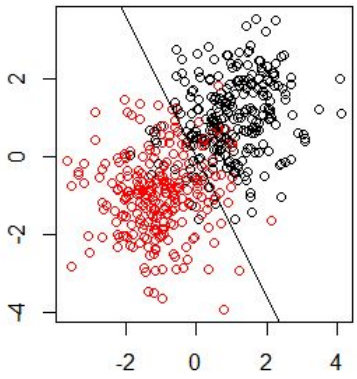
- Pruned:



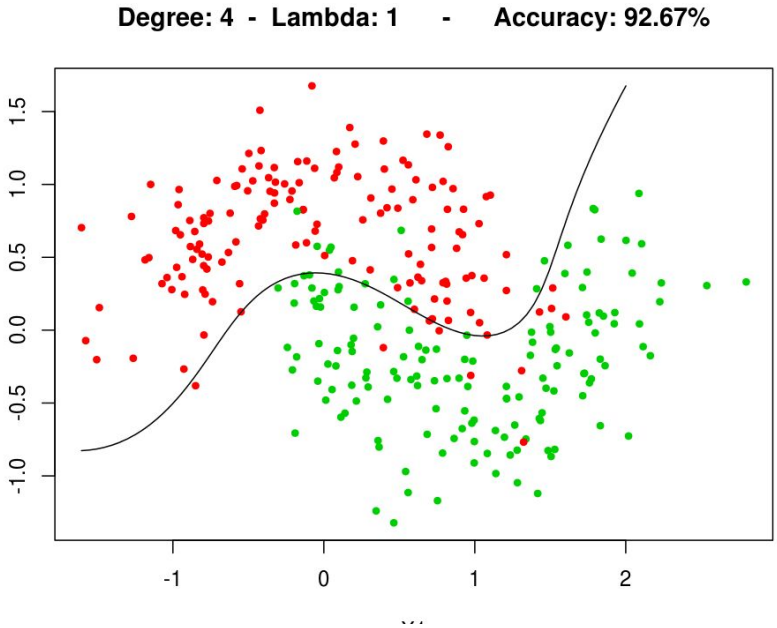
- Decision stump - axis-aligned line



- linear reg - line (linear regression is not a classifier, so it shouldn't have a decision boundary)



- polynomial regression - polynomial line (https://stats.stackexchange.com/questions/148638/how-to-tell-the-difference-between-linear-and-non-linear-regression-models/148713#148713) ?

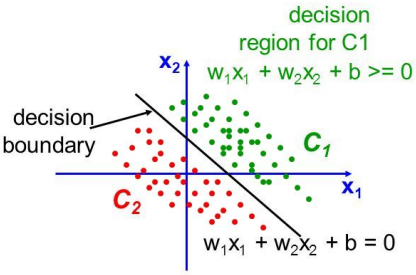


- the perceptron

Perceptron: Classification

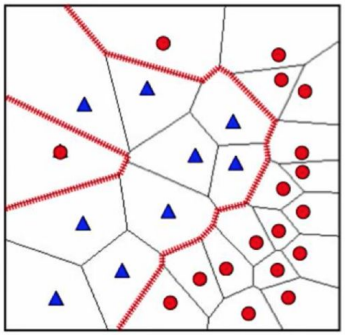
- The equation below describes a hyperplane in the input space. This hyperplane is used to separate the two classes C1 and C2

$$\sum_{i=1}^{m} w_i x_i + b = 0$$

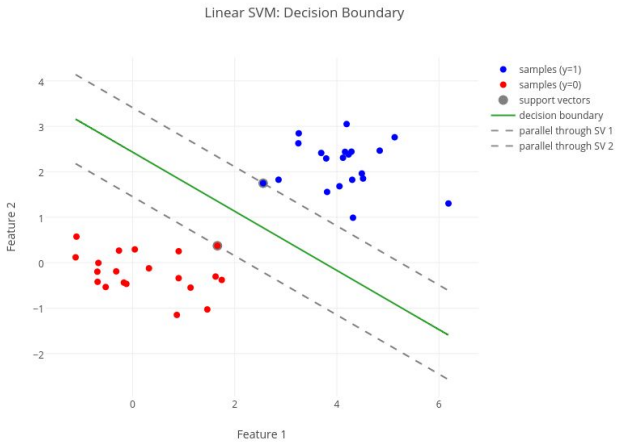


Neural Networks NN 2 5

- logistic regression
- neural nets - It is hard to determine why neural nets make certain decision boundaries
- 1NN - Arbitrary (Voronoi tessellation)



- KNN - arbitrary
- ensemble, boosting - axis aligned because underlying learners are decision stumps
    - Kind of carves out a pixelated shape, as Professor showed us in class
- SVM - linear / kernel dependent?
    - Linear case: the line should look as though it's equidistant between the two "clusters" (see below)



- Naive Bayes - MLE (no decision boundary) **not true it has a boundary - see the link below**.
- Good document: http://michael.hahsler.net/SMU/EMIS7332/R/viz_classifier.html (this is brilliant)

Theory

- Training vs. Testing (see Vocab)
- Generalization
  - Goal of Machine learning
- Overfitting, underfitting (see Vocab)
- bias vs variance
  - https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/
- cross validation
  - Using several sets of data in order to measure error
- shattering, VC dimension (see Vocab)

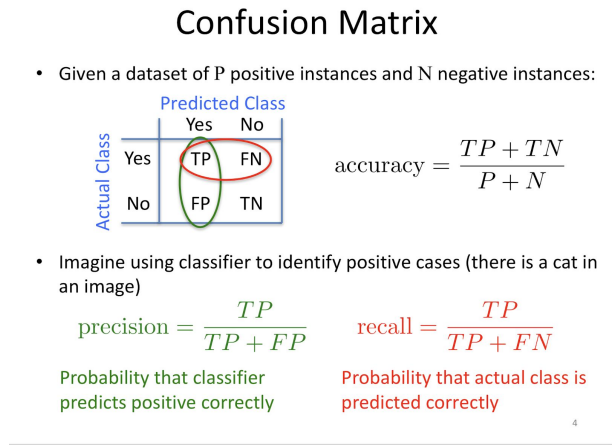
Supervised Learning

- Difference between UL and RL

SL	- works with **labeled** datasets and provides **immediate feedback** on if a sample was "good" or "bad" - "optimizing model" means that it labels data well
UL	- does not give *any* feedback on how *right* some classification is - "optimizing model" means that it clusters such that it scores well
RL	- does not give *immediate* feedback - "optimizing model" means that there's a behavior that scores well

- Machine learning algorithms <P,T,E>
  Each machine learning problem can be precisely defined as the problem of improving some measure of performance P when executing some task T, through some type of training experience E.
- Classification VS Regression (See Vocab)
- Error, accuracy
  - FP = Type I Error
  - FN = Type II Error
  - Accuracy = (TP + TN) / (TP + TN + FP + FN) (Over all classes)
  - Error = 1 - accuracy
- Confusion matrix:
  divided up into TP (true positive), TN (true negative), FP (false positive), FN (false negative)
  -

Confusion matrix	Pred False	Pred True
Actual false	TN	FP
Actual True	FN	TP



- Precision = TP / (TP + FP)
  - Memory tip: Precision starts with p, prediction also starts with p, you're looking at TP vs. all predicted positives.
- Recall = TP / (TP + FN)
  - Memory tip: Recall starts with R, Real starts with R, so you're looking at TP vs. the "real" positives
- F1 = (2PR) / (P+ R)
  - Use this score because accuracy fails on unbalanced datasets
  - Is this (2*precision*recall)/(precision+recall)? Yes.

Perceptron, Linear regression, Logistic regression

- Regularization (relation to bias and variance)
  - Low bias and low variance together is hard to achieve
  - Higher regularization term leads to more bias less variance
- Learning, gradient descent
- Loss functions: difference between label and model prediction.
- Features

- **Perceptron:**
  - $h(\boldsymbol{x}) = \text{sign}(\boldsymbol{\theta}^\intercal \boldsymbol{x})$ where $\text{sign}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$
  - Uses the following update rule each time it receives a new training instance ($x^{(i)}$, $y^{(i)}$:

$$\theta_j \leftarrow \theta_j - \frac{\alpha}{2}\underbrace{\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)}_{\text{either 2 or -2}} x_j^{(i)}$$

- Make no changes if the prediction matches the label. Otherwise, adjust θ
- Rewrite as $\theta_j \leftarrow \theta_j + \alpha y^{(i)} x_j^{(i)}$
- Since α only scales θ by a constant, we can eliminate it and our perceptron rule becomes

  Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$ j[
-
- Cost/loss function:

$$J_p(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \max(0, -y^{(i)}x^{(i)}\boldsymbol{\theta})$$

  - $\max(0, -y^{(i)}x^{(i)}\boldsymbol{\theta})$ is 0 if the prediction is correct
  - Otherwise, it is the confidence in the misprediction
- Online Learning - the learning mode where the model update is performed each time a single observation is received
- Batch Learning - the learning mode where the model update is performed after observing the entire training set
- A single perceptron only can work for linearly separable data.

- Linear regression:

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

- Cost/loss function:
  - Fit by solving $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
- Learning: Gradient Descent
  - Choose initial value for θ
  - Until we reach a minimum, choose a new value for θ to reduce $J(\theta)$
    - Learning rate, α, must not be too small or too large
      - If too small - convergence will take forever
      - If too large - might never converge and repeatedly overshoot
  - Don't need to worry about local minima since the least squares objective function is convex
  - Extending Linear Regression to more complex models
    - Transform the inputs for linear regression in order to make the data be fit by a linear function
- Regularization: aims to find a fairground between having too high bias and too high variance
  - A method for automatically controlling the complexity of the learned hypothesis
  - Works well when we have a lot of features, each that contributes a bit to predicting the label
  - Idea: penalize large values of θj

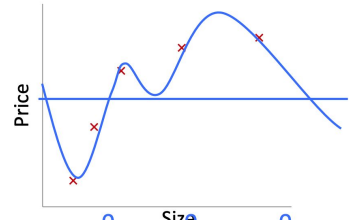
- Linear regression objective function

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{d}\theta_j^2$$

model fit to data — regularization

  - λ is the regularization parameter ($\lambda \geq 0$)
  - No regularization on θ_0!

- What happens if we set λ to be huge (e.g., 10^{10})?



$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

(0) (0) (0) (0)

  - If lambda is too large, then the model will just make a straight line in order to not be complex
  - If lambda is too small, then the model will be complex

- Logistic Regression:
  - Classification based on probability, unlike perceptrons.
    - I.e., learn $p(y \mid x)$
  - Uses sigmoid function
  - Predicts y = 1 if $h\theta(x) >= 0.5$, predicts y = 0 if $h\theta(x) < 0.5$
  - Cost/loss function:

    Can't just use squared loss as in linear regression:

$$J(\boldsymbol{\theta}) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

  - Learning:
    - Identical to linear regression, except the model $h\theta(x)$ is the sigmoid function
    - Note:

      Using the logistic regression model

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^{\top}\boldsymbol{x}}}$$

      results in a non-convex optimization

      - However, with a log term added, turn into convex

  - Regularization: same as Linear Regression regularization

Instance Based Learning- KNN:

- The Curse of Dimensionality: as the number of features/dimensions increases, the amount of data needed to generalize accurately grows exponentially

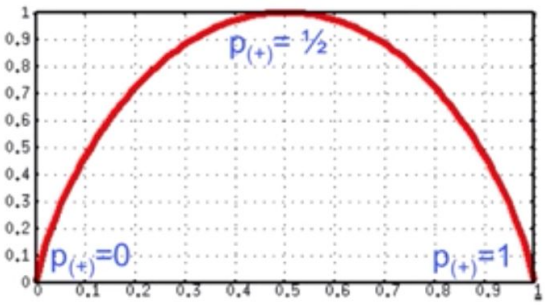
Neural Net

- Structure (layers)
  - Input layer → hidden layer(s) → output layer (contains the prediction $h\theta(x)$)
  - Each layer has a number of nodes
    - Each node has an activation α_i^j, unit i in layer j
    - Also has a weight matrix controlling function $\theta^{(j)}$, mapping from layer j to layer $j + 1$
- Activation functions (linear, logistic)
  - Usually a nonlinear function
  - Logistic activation function is the sigmoid function, which activates after the threshold similar to logistic regression
  - $$g(z) = \frac{1}{1 + e^{-z}}$$
- Backpropagation
  - If the output of the network is correct, no changes are made
  - If there is an error, **weights are adjusted** to reduce the error
  - The trick is to **assess the blame for the error and divide it among the contributing weights**
  - Each hidden node j is "responsible" for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects
  - $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node
  - Then, **the "blame" is propagated back** to provide the error values for the hidden layer
- Restriction bias
  - Restriction bias is the representational power of an algorithm, or, the set of hypotheses our algorithm will consider. So, in other words, restriction bias tells us what our model is able to represent.
  - NNs can model **continuous functions with <u>one</u>** big enough **<u>hidden layer</u>**
  - NNs can model **arbitrary functions** with an extra hidden layer = **<u>at least 2 hidden layers</u>**
  - HOWEVER, **these concepts, while feasible, can cause overfitting**
    - **Cross-validation can help decide network structure and when to stop training to combat overfitting**
  - NN training has the same familiar **concave plot for CV error** that helps us decide when to stop training (a.k.a. how many epochs)
- Preference Bias (Inductive Bias)

- Preference bias is simply what representation(s) a supervised learning algorithm prefers. For example, a decision tree algorithm might prefer shorter, less complex trees. In other words, it is our algorithm's belief about what makes a good hypothesis.
- **NN weights are initialized with small, random values b/c:**
  - **Helps avoid local minima**
  - **Variability in training to prevent repetition of errors (such as not halting)**
  - **Large weights can lead to overfitting because you can represent arbitrarily complex functions**
- Thus, because of these properties, NNs prefer
  - **Low complexity, simpler explanation = simpler NN structures**
  - **Think Occam's Razor**
    - **Why use complex when simple do trick**
- 

Decision Tree (working knowledge):

- Splitting
  - Pick best attribute (splits data most evenly)
    - ID3 uses max information gain
      - Gain(S,A) = Entropy(S) - $\sum_{v} \frac{|Sv|}{|S|}$ Entropy(S_v)
      - We want entropy to go down as much as possible when choosing an attribute, meaning max information gain
- • Entropy / Conditional Entropy
  - Measures the impurity of S
    - S = subset of training examples
  - Remember p log p
  - $H(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$
  - How many bits are needed to tell if item X is positive or negative (true/false)
  - Entropy for a 50/50 split = 1
    - Ex: 3 yes / 3 no => H(S) = - $\frac{3}{6} \log_2 \frac{3}{6}$ - $\frac{3}{6} \log_2 \frac{3}{6}$ = 1 bits
  - Entropy for a 100/0 (completely certain) split = 0
    - Ex: 4 yes / 0 no => H(S) = - $\frac{4}{4} \log_2 \frac{4}{4}$ - $\frac{4}{4} \log_2 \frac{4}{4}$ = 0 bits
  - A good training set for learning has close to 1 bits for entropy



Entropy $H(X)$ of a random variable X

$$H(X) = -\sum_{i=1}^{n} P(X=i)\log_2 P(X=i)$$

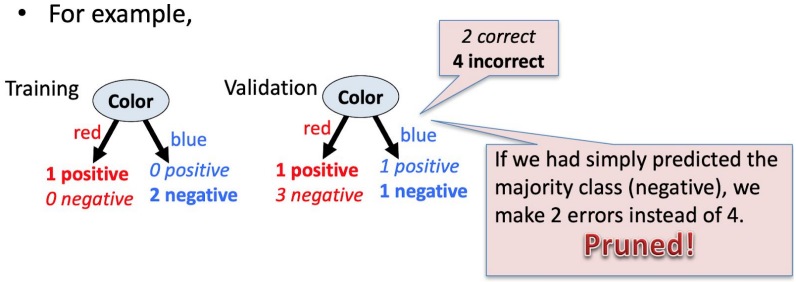
Conditional entropy $H(X|Y)$ of X given Y:

$$H(X|Y) = \sum_{v \in values(Y)} P(Y=v)H(X|Y=v)$$

- Information Gain
  - The expected reduction in entropy of target variable Y for data sample S
  - Which attribute is most useful for discriminating between classes to be learned
  - Used to decide ordering of attributes in nodes of a decision tree

Information Gain = entropy(parent) – [average entropy(children)]

- • Pruning (deep vs shallow / stumps)
  - Done by replacing a whole subtree by a leaf node
  - Replacement is done if the expected error rate in the subtree is greater than in the single leaf
    - For example,



- • Overfitting
  - Consider possible noise
    - Two examples may have same attributes but different classifications
    - Instances may be labelled incorrectly
  - Some attributes may be irrelevant, leading to overfitting
    - Meaningless regularity in the data

Hypothesis $h \in H$ **overfits** training data if there is an alternative hypothesis $h' \in H$ such that

$$error_{train}(h) < error_{train}(h')$$

and

$$error_{\mathcal{D}}(h) > error_{\mathcal{D}}(h')$$

  - How to avoid overfitting?

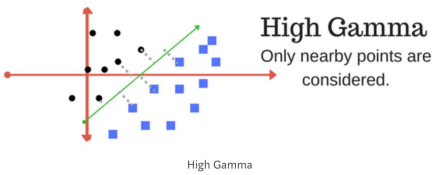
- Stop growing when data split is not statistically significant
- Get more training data
- Remove irrelevant attributes
- Post-prune the tree

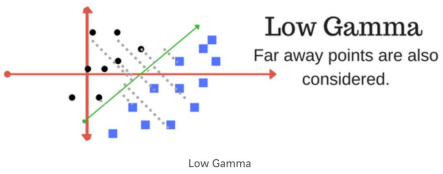
Ensembles

- Bagging
  - Training several different learners on different subsets of the training set (**bootstrap replication**)
  - Querying all of them and returning the mean (regression) or mode (classification) as your predicted label
- Adaboost (bias/variance, when do you stop training)
  - Usually uses high bias classifiers
  - Adaboost *almost-never* overfits, given we use weak learners - it overfits in presence of noise
  - Reduces both bias and variance
  - Linear combination of weak classifiers
- Properties of individual learners (what can they be, high/low accuracy)
- Successful ensembles require diversity; each classifier should make different mistakes

L\SVM (We only need to know about linear kernel)

- What are support vectors?
  - Support Vectors are the coordinates of observations that lie closest to the decision surface (i.e the hyperplane)
  - These are the observations which are the most difficult to classify.



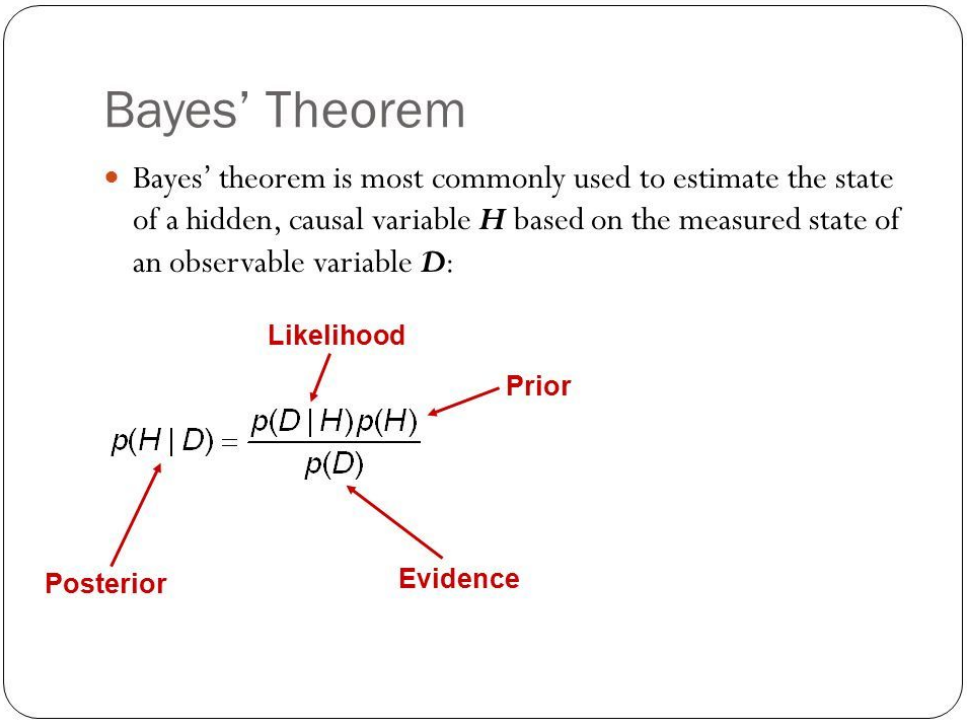


  - So are the support vectors chosen by tuning gamma, like in this example? Could someone correct/verify this? **Is this in the slides?**
    - Gamma is a parameter for a linear svm that essentially controls what points should be considered when placing the hyperplane to separate the classes
  - They have direct bearing on the optimum location of the resulting

    - Given labeled training data, the algorithm outputs an optimal hyperplane, which categorizes new examples (link to help understand: https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72)
    -
- margin (how does changing support vectors change decision boundary/margin)
    - Changing support vectors shifts the margin because the distance from the separating hyperplane and the support vectors are what we are attempting to maximize. The distance from the hyperplane on the positive side vs. the one on the negative side is called the margin.
- Kernels are used to transform data into correct format
    - Linear, polynomial, gaussian

Naive Bayes

- Basic probability, Bayes Rule
    - Bayes rule: $$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$
    - Derived from conditional probability formula, derivation explanation found here: https://www.eecs.qmul.ac.uk/~norman/BBNs/Bayes_rule.html
- Prior, likelihood, posterior



- Independence assumption and why does it help?
    - The independence assumption assumes that all attributes of X are conditionally independent, given Y
    - Allows us to easily estimate the joint probability distribution by taking a product
        - Instead of directly learning the joint prob. Distribution

- If you're still having trouble understanding, this link explains it pretty well: https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
- Useful for large datasets
- Fast to train
- It can only learn linear decision boundaries
- Only drawback is its assumption of independent attributes

Vocab

- **Classification** - Taking an input and mapping it to a discrete number of labels (True or False, M/F/X)
- **Regression** - Predicting an output to a continuous real valued function
    - All Classification problems can be turned into a regression problem
- **Training Set** - set of labels from input to output used to train the ML Algo
- **Testing Set -** set of labels from input to output used to test how well your ML Algo generalizes
- **Validation Set** - set of labels from input to output used to gain an understanding of where the model starts to overfit
- **Supervised Learning** - Take examples of inputs and outputs, now given a new input, predict an output
- **Bias** - How close your predictions are to your input data (Difference in expected value of the estimator and the true value)
    - High bias - Underfitting, model makes multiple errors when trying to predict
    - Low bias - Model makes correct predictions, more closely fits the training data
- **Variance** - How much your prediction models differ from changes in the input data (how sensitive it is to changes in the input data)
    - High variance - Overfitting, model is sensitive to the data, fits to noise, and unable to generalize
    - Low variance - Model is able to generalize all inputs and not fit to noise.
- **Variance vs. Bias Tradeoff** - We want our models to have low bias and low variance, however these normally have a tradeoff.
- **Regularization** - Penalty given to complex models, increases bias and decreases variance
- **Shatter** - a model class can shatter a set of points if for every possible labeling over those points, there exists a model in that class that obtains zero training error
- **VC Dimension** - the maximum number of points that can be arranged so that the class of models can "shatter" those points; used as a measure of the power of a particular class of models.
    - For example, the VC dimension of a hyperplane in 2D is 3 (in *d* dimensions it is *d+1*). A sine wave has infinite VC dimension.
    - A problem is PAC Learnable iff its VC dimension is finite
- **Weak Learner** - a classifier that is only slightly better than random guessing. Has high bias
- **Linearly separable** - The data can be separated by a line, plane, or hyperplane.
- **Lazy Learner** - Does no "training" but rather has all the computation when trying to predict
- **Precision** - Of the samples you predicted true, how many were actually true
    - **True positives / (True positives + False positives)**
- **Recall** - Of the samples that are actually true, how many did you predict to be true

    - **True positive / (True positives + False negatives)**
- **No Free Lunch** - If you average the performance of a model across all possible learning problems, it will do no better than guessing
- **Occam's Razor** - ***Why use many words when few words do trick***
    - Why use complex when simple do trick
    - Minimal complexity produces best result (prevents overfitting)

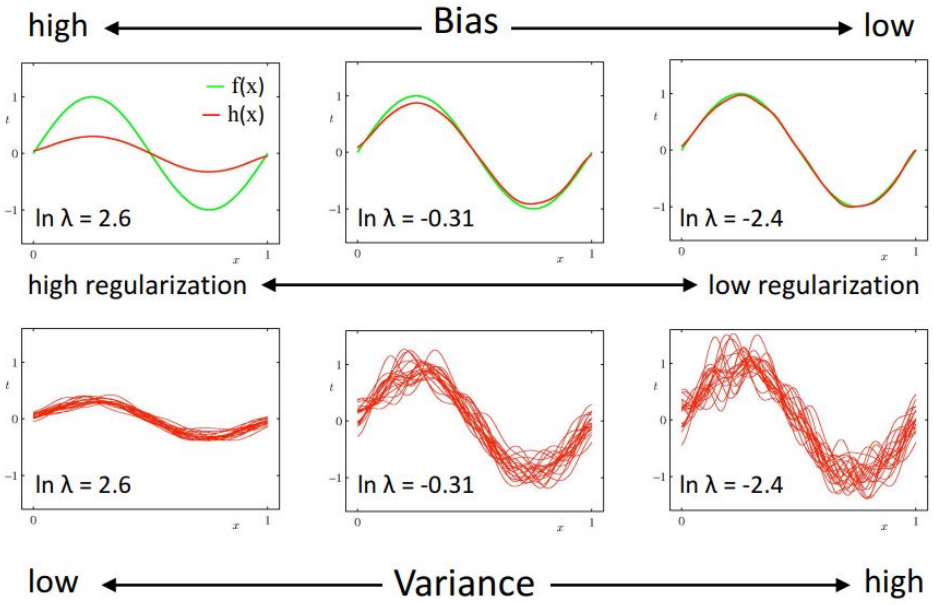
Experimental Protocol

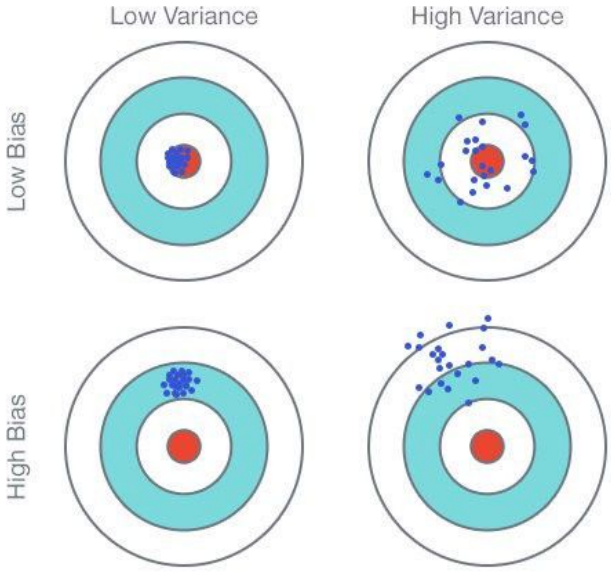
- Never mix your training and testing set - splitting the data into a training and testing set should be the FIRST step
- Need a healthy ratio of labels
    - It is problematic if you have a lot of one label, and not a lot of another.

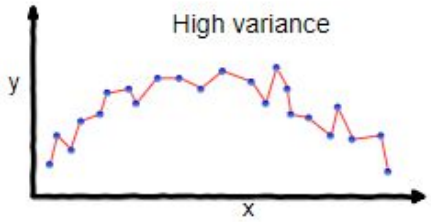
Learning Theory

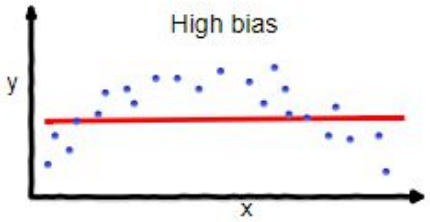
Often, there is a tradeoff between bias and variance. If our model is too "simple" and has very few parameters, then it may have large bias (but small variance); if it is too "complex" and has very many parameters, then it may suffer from large variance (but have smaller bias).

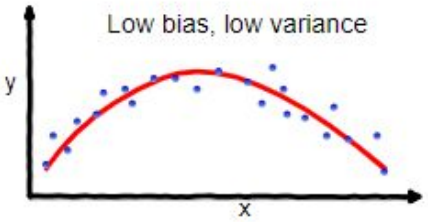
Illustration of Bias-Variance











overfitting

underfitting

Good balance

Randomized Optimization

- **Local Search / Gradient Descent**
  - **Local Search**: Use single current state and move to neighboring states
    - Find or approximate the best state according to some objective function
    - Only optimal if the space to be searched is convex
  - **Idea:** start with an initial guess at a solution, and incrementally improve it until it is a solution.
  - **Hill-Climbing Search**: "Like climbing Everest in thick fog with amnesia"
    - Move in the direction of increasing evaluation function until you can't (reached the peak (local optimum))
    - Iteratively trying to maximize the fitness function
    - $s_{next} = arg_s max\ f(s)$
    - Greedy local search
    - Don't look ahead of the immediate neighbors
  - **Gradient Descent**: procedure for finding $arg_x min f(x)$
    - $x_{i+1} \leftarrow x_i - \eta f'(x_i)$
      - Step size η is small (0.1, 0.05)
    - Newton's method: $x_{i+1} \leftarrow x_i - \eta f'(x_i) / f''(x_i)$
      - Takes a more direct route to the minimum (converges quicker), but is more expensive to compute
- **Advantages**
  - Uses very little memory

    - Finds often *reasonable* (if not optimal) solutions in large or infinite state spaces.
        - If the space is convex, the result will be optimal regardless of where you start
- **Drawbacks**
    - **Local Maxima:** peaks that aren't the highest point in the space but the algorithm thinks it's the highest
    - **Plateaus:** the space has a broad flat region that gives the search algorithm no direction (random walk)
    - **Ridges:** dropoffs to the sides; steps to the North, East, South, and West may go down, but a step to the NW may go up.
        - If the agent doesn't find that direction, then may think that it's at a local optimum
        - This is an issue with overly simple neighbor functions
    - Flat regions in optimization landscapes are very difficult for the optimization algorithms
- **Random Restarts**
    - Choose multiple starting values and use the algorithm on them.
    - This fights against local optima.
    - Do as many random restarts (saving the optimal solution) as your time and computation budget allow
    - If you know something about the distribution of the space and start locations, can take random samples of the distribution to bias the starting location choices (steer the sampling based on knowledge of the problem or previous samples)
    - We can also intentionally pick starts that are distant from each other to ensure we consider the whole feature space
- **Simulated Annealing**
    - SA = hill-climbing + non-deterministic (non-greedy) search
    - Picks a move randomly instead of the best one, if this move y is better than the state we are in currently, then we choose it. If not, we choose it with some probability.
    - With a higher T (temperature), probability of "locally bad" move is higher
    - T decreases as algorithm runs longer
        - High T - exploration, low T - exploitation
        - Terminate when T decreases to 0
        - With a slow temperature decay, we should theoretically find the global optimum (though this isn't very practical)
    - This randomization avoids getting trapped at a local optimum
        - Allows you to gradually move up to a local optimum, but can also escape it
- **Local Beam Search**
    - Start with *k* random states instead of only one
    - Major difference with random-restart hill climbing: Information is shared among *k* search threads.
        - The next *k* states chosen at the best *k* successor states
    - Weak version of GA
- **Genetic Algorithms**
    - Also starts with k random states
    - New states generated by either a single state mutation of a single state and/or reproduction by combining two parent states
    - Each *chromosome* represents a possible solution and is made up of a string of genes
        - Analogous to a state
    - Reproduction (crossover)

        - Selects genes from two parent chromosomes and creates two new offspring
        - In practise, more than two parents can be (and have been) used (*https://en.wikipedia.org/wiki/Genetic_algorithm*)
        - Randomly choose a crossover point
        - Selecting parents (with replacement) proportionally to their fitness
    - Mutation
        - With a certain probability (usually low), randomly change a gene in the new offspring
        - For binary encoding - flip one of the bits
        - For real value numbers, sample from a gaussian to find a new number
    - Mutation and selection are the most beneficial parts of the algorithm. In practice, crossover just increases diversity but doesn't really help that much
    - *Opinion is divided over the importance of crossover versus mutation (https://en.wikipedia.org/wiki/Genetic_algorithm)*
    - Positive points:
        - Highly parallelizable
        - Random exploration can find solutions that local search cannot
        - Don't need to calculate the fitness function derivative
            - GA is good when evaluating a function that is not that hard
    - Negative points:
        - Large number of tunable parameters (difficult to replicate performance from one problem to another)

Unsupervised learning

- **Difference with Supervised Learning, & Reinforcement Learning**
    - Supervised learning uses labeled data pairs (**x**, y) to learn a function $f: X \rightarrow Y$

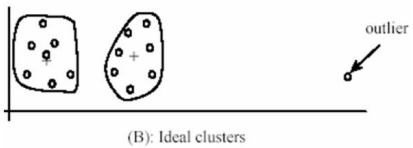
	From input **x**, output:
UNSUPERVISED	Summary **z**
SUPERVISED	Prediction **y**
REINFORCEMENT	Action **a** to maximize reward **r**

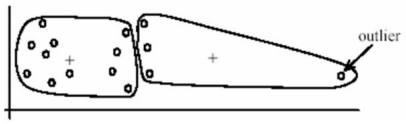
What We'll Cover in this Course

- **Supervised learning**
  - Decision trees
  - Regression
  - Gradient Descent
  - Neural Networks
  - Instance-Based Learning
  - Ensemble Learning
  - Support Vector Machines
  - Bayesian learning & Inference
  - Learning theory
- **Unsupervised learning**
  - Randomized Search
  - Clustering
  - Feature Selection
  - Dimensionality Reduction
- **Reinforcement learning**
  - Markov Decision Processes
  - Value Iteration
  - Policy Optimization

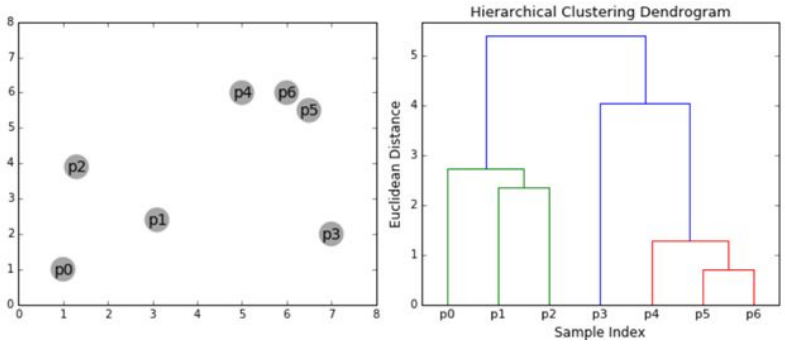
- **Clustering vs. Feature Selection vs Dimensionality Reduction**
  - **Clustering**
  - **Feature Selection**
  - **Dimensionality Reduction**

Clustering

- **K-means Clustering**
  - Iteratively re-assign points to the nearest cluster center
  - K-Means(*k*, X) algorithm
    - Randomly choose *k* cluster center locations (centroids)
    - Loop until convergence:
      - Assign each point to the cluster of the closest centroid
      - Re-estimate the cluster centroids based on the data assigned to each cluster
  - Finds the **local optimum** of $arg_S min \sum_{i=1}^{k} \sum_{x \varepsilon S_i} \left\| x - \mu_i \right\|_2^2$
    - $S = \{S_1, \ldots, S_k\}$ is a partitioning over $X = \{x_1, \ldots, x_n\}$ s.t. $X = \bigcup_{i=1}^{k} S_i$ and $\mu_i = mean(S_i)$
  - Pros
    - Finds cluster centers that minimize conditional variance (good representation of data)
    - Easy to implement
  - Cons
    - Need to choose k (amount of clusters)
    - Sensitive to outliers
    - 

- 

    - Prone to local minima
    - All clusters have the same parameters (e.g. distance measure is non-adaptive)
    - Hard clustering (each instance is only assigned to exactly one cluster)
  - *Very* sensitive to the initial points chosen
    - run K-Means many times each with different initial centroids
    - Can also seed the centroids using a better method than randomly choosing the centroids (e.g. farthest-first sampling)
  - K-Medoids
    - Represent the cluster with one of its members, rather than choosing the mean of its members
    - Choose the member (data point) that minimizes cluster dissimilarity
  - **Importance of similarity measure**
- **Agglomerative / Hierarchical Clustering**
  - agglomerative: Start with each point as its own cluster and iteratively merge the closest clusters
  - Cluster similarity could be based on average distance between points, maximum distance, minimum distance, distance between means, distance between medoids
  - Can dynamically find the number of clusters (k) without prior domain knowledge
    - Number of clusters is found via a threshold based on max number of clusters or based on distance between merges
  - **Dendrograms**
    - Dendrograms are tree diagrams. Applied here, they can be used to show the hierarchy of clusters through different iterations.
    - With agglomerative clustering, the “bottom-up” approach, the dendrogram will have many clusters at the bottom. As you go up through progressive iterations, the distance between clusters increases (as each clusters’ ‘definition’ starts including more points and actual clusters become defined.
    - With divisive clustering, the dendrogram would look the opposite. As a “top-down” approach, it will have 1 cluster at the bottom and then split on upwards, decreasing the distance between clusters as the number of clusters grows.
- **Hierarchical vs. Agglomerative Clustering**
  - It looks like slides didn't define hierarchical clustering, so here's a quick note about it.
  - **Hierarchical clustering** tries to build a hierarchy of clusters, meaning that we want a spectrum of clusterings ranging from many small clusters to a small amount of large clusters.
  - **Agglomerative clustering** is one form of hierarchical clustering where we iteratively merge clusters into larger ones, starting with each data point being its own cluster.
  - The other approach, **divisive clustering**, starts with one large cluster and iteratively splits the clusters until each data point is its own cluster.



- **Gaussian Mixture Models and EM**
  - **Assumptions**
    - K components. The i'th component is ω_i
    - Component ω_i has an associated mean vector μ_i qq
    - GMM Assumption - Each component generates data from a Gaussian with mean μ_i and covariance matrix $\sigma^2 I$ = Σ_i
  - **Relationship to K-means (soft)**
    - Clustering typically assumes that each instance is given a "hard" assignment to exactly one cluster - doesn't allow for uncertainty in class membership or for an instance to belong to more than one cluster
      - K-means is hard clustering
    - GMM is *soft clustering* - give probabilities that an instance belongs to each of a set of clusters
    - Each instance is assigned a probability distribution across a set of discovered categories
    - When you have to choose - take the cluster with the highest probability
  - **Should have working knowledge of fitting a GMM**
    i. Pick a component at random. Choose component i with probability $P(\omega_i)$
    ii. Datapoint ~ $N(\mu_i, \sigma^2 I)$ for GMM -- Datapoint ~ $N(\mu_i, \Sigma_i)$ for General GMM
    - Mixture models
      - Weighted sum of a number of pdf's (probability distribution functions) where the weights are determined by a distribution π

$$p(x) = \sum_{i=0}^{k} \pi_i f_i(x)$$

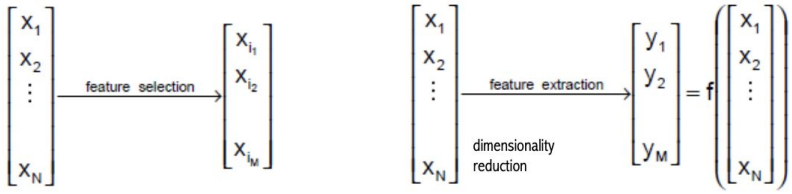
    - GMM
      - The weighted sum of a number of Gaussians where the weights are determined by a distribution π, $\sum_{i=0}^{k} \pi_i = 1$

$$p(x) = \sum_{i=0}^{k} \pi_i N(x|\mu_k, \Sigma_k)$$

      - - typo: should be μ_i and Σ_i
  - Algorithm:
    - Expectation step - compute the "expected" clusters of all data points

        - $P(\omega_i|x_k, \lambda_t)$
        - Evaluate a Gaussian at x_k
    - Maximization step - estimate μ and Σ given our data's class membership distributions
        - $\mu(t+1)$

Feature Selection

- Given a set of *n* features, the goal of feature selection is to select a subset of *d* features (d < n) in order to minimize the classification error
- Insights into which factors are the most *representative* of your problem
- Avoid curse of dimensionality - amount of data grows *exponentially* with number of features $O(2^N)$
- Steps
    - 1) search the space of possible feature subsets
    - 2) pick the subset that is optimal or near-optimal with respect to some objective function
    - Search strategies - optimal or heuristic
    - Evaluation strategies - filter or wrapper methods
- **Difference compared with Dimensionality Reduction**
    - 

    - Feature selection
        - Only a small number of features need to be computed (i.e. faster classification)
        - The measurement units (length, weight, etc.) of the features are preserved
    - Dimensionality reduction - feature extraction
        - All features need to be computed
        - The measurement units (length, weight, etc.) of the features are lost
- **Filter vs. Wrapper Methods**
    - Filter
        - Evaluation is ***independent*** of the classification algorithm
        - The objective function evaluates feature subsets by their information content (typically inter-class distance, statistical dependence, or information-theoretic measures)
        - **Advantages**
            - Fast execution - filters generally involve non-iterative computation on the data set (which can execute much faster than a classifier training session)
                - Generality - since filters evaluate the intrinsic properties of the data, rather than their interactions with a particular classifier, their results exhibit more generality
                - The solution will be "good" for a larger family of classifiers
        - **Disadvantages**

        - Tendency to select large subsets - since the filter objective functions are generally monotonic, the filter tends to select the full feature set as the optimal solution. This requires the user to select an arbitrary cutoff on the number of features to be selected
    - Wrapper
        - Evaluation uses criteria ***related*** to the classification algorithm
        - The objective function is a pattern classifier, which evaluates feature subsets by their predictive accuracy (recognition rate on test data) by statistical resampling or cross-validation
        - **Advantages:**
            - Accuracy - wrappers generally achieve better recognition rates than filters since they are tuned to the specific interactions between classifier and the dataset
            - Ability to generalize - wrappers have a mechanism to avoid overfitting, since they typically use cross-validation measures of predictive accuracy
        - **Disadvantages:**
            - Slow execution - the wrapper must train a classifier for each subset (or several classifiers if CV is used)
            - Lack of generality - solution is tied to the bias of the classifier used in the evaluation function. The “optimal” feature subset will be specific to the classifier under consideration
- Search strategies
    - An exhaustive search would require examining all nCd possible subsets - number of subsets grows combinatorially, making exhaustive search impractical
    - **Sequential Forward Selection (SFS)**
        - Heuristic search
        - Performs best when optimal subset is small
        - Limitation - unable to *remove* features that become non-useful after the addition of other features
        - 1) the best *single* feature is selected (using some criterion function)
        - 2) *pairs* of features are formed using one of the remaining features and this best feature - best pair selected
        - 3) triplets of features are formed using one of the remaining features and these two best features - best triplet selected
        - 4) continue until a predefined number of features are selected
    - **Sequential Backward Selection (SBS)**
        - Heuristic search
        - SBS performs best when the optimal subset is large
        - Limitation - unable to *reevaluate* the usefulness of a feature after it has been discarded
        - 1) criterion function is computed for all *n* features
        - 2) each feature is deleted one at a time, the criterion function is computed for all subsets with *n-1* features, and the worst feature is discarded
        - 3) each feature among the remaining *n-1* is deleted one at a time, and the worst feature is discarded to form a subset of *n-1* features
        - 4) the procedure continues until a predefined number of features are left
    - **Bidirectional Search (BDS)**

        - Applies SFS and SBS simultaneously
            - SFS is performed from the empty set
            - SBS is performed from the full set
        - To guarantee that SFS and SBS converge to the same solution
            - features already selected by SFS are not removed by SBS
            - Features already removed by SBS are not added by SFS

Dimensionality Reduction

PCA

- Orthogonal projection of data onto lower-dimension linear space that maximizes variance of projected data and minimizes mean squared distance between data point and projections
- Algorithm:
    - Given data in $\{x_1, \ldots, x_n\}$, compute the covariance matrix Σ
        - *X* is the *n x d* data matrix
        - Compute data mean (average over all rows of *X*)
        - Subtract mean from each row of *X* (centering the data)
        - Compute covariance matrix $\Sigma_{dxd} = X^T X$
    - PCA basis vectors are given by the eigenvectors of Σ
        - $\{q_i, \lambda_i\}_{i=1\ldots n}$ eigenvectors, eigenvalues of Σ
        - $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$
        - Each column of Q gives weights for a linear combination of the original features
            - E.g. $0.34\,feature1 = 0.04\,feature2 - 0.64\,feature3 + \ldots$
- Can ignore the components of lesser significance - choose the first *k* eigenvectors based on their eigenvalues
    - Final data set has only *k* dimensions
    - Lose some information but if eigenvalues are small enough, you don't lose much
- Re-projected data matrix given by $\widehat{X} = X\widehat{Q}$
- This provides best reconstruction through the orthogonal line. It provides the smallest L_2 error.
    - Reconstruction: moves from N to M Dimensions
- The **disadvantage** of using PCA is that the discriminative information that distinguishes one class from another might be in the low variance components, so using PCA can make performance worse.
    - 'assumes that features that present high variance are more likely to have a good split between classes
- Doesn't discard less relevant features, but linearly transforms them into new attribute
- A principal axis with eigenvalue 0 = irrelevant feature but might still be useful(relevance vs. usefulness)
    - Relevance refers to information gain in an ideal learner (the Bayes optimal classifier)
    - Usefulness refers to decrease in error in a *particular* learner
- PCA looks for properties that show as much variation across classes as possible to build the principal component space. The algorithm use the concepts of variance matrix, covariance matrix, eigenvector and eigenvalues pairs to perform PCA, providing a set of eigenvectors and its respectively eigenvalues as a result.

Understand meaning of principal components

- Principle component #1 points in the direction of **largest variance**
    - Also finds direction **mutually orthogonal** to first component found

    - Each subsequent principle component is orthogonal to the previous ones and points in the directions of the largest variance of the residual subspace

Orthogonal projection onto lower dimensional space
Maximize variance / minimize squared error
Use for preprocessing
Applications

- Visualization
    - Visualizing the clusters of the handwritten digit data set
        - From high dimension to 2 dimensions (though only 16% of the variance is explained in the 2 dimensions)
- facial recognition
    - Eigenfaces
        - The eigenvectors of the covariance matrix of the probability distribution of the vector space of human faces
        - The *standardized face ingredients* derived from statistical analysis of many pictures of human faces
        - A human face may be considered to be a combination of these standard face ingredients
    - To generate a set of eigenfaces
        - Large set of digitized images on human faces is taken under the same lighting condition
        - Images normalizes to line up the eyes and mouths
        - The eigenvectors of the covariance matrix of the statistical distribution of face image vectors are then extracted
        - Eigenvectors - eigenfaces
- Image compression

Markov Decision Processes

• Definition:

A Markov decision process (MDP) is a discrete time stochastic (random) control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and under the control of a decision maker. MDPs are useful for studying optimization problems solved via dynamic programming and reinforcement learning.

- Defined by
    - Set of states
    - Set of actions
    - Transition function $T(s, a, s')$
        - probability that a from s leads to s' - P(s'|s,a)
        - Aka transition model, dynamics
    - Reward function $R(s, a, s')$

- Start state
- Maybe a terminal state

• **What does "Markov Assumptions" Mean?**

- Markov First Order Assumption means that the next state actions only depend on the evidence being observed at the current state, regardless of past states.
  - Given the present state, the future and past are independent

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \dots S_0 = s_0)$$

$$=$$

  - $$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$
  - When you're in a state, can reason which action to take next without having to consider your previous states

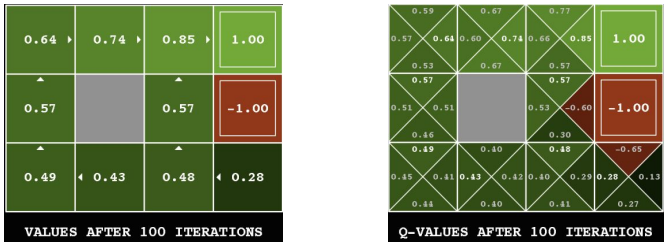
• **Policies:**

- A mapping from states to actions: given the agent is at some state, what action should it take?
- $\pi : S \rightarrow A$
- π^* = optimal action from state *s*
  - Optimal policy - maximizes the expected utility if followed (we want to find this)
- Explicit policy - defines a reactive (reflex) agent

• **MDP search trees:**

- Can take you a discrete set of actions at any state
- Node = state, edge = action, child = next state
- When take action *a* (go down one edge), can end up at any of the other children because stochastic transition function
- $(s,\ a,\ s')$ is a transition

• **Reward vs Utility vs Value vs Q function:**

- <u>Value = utility</u>
- $V^*(s)$ = expected utility of starting in s and acting optimally (average sum of (discounted) rewards)
  - If we know the value at every state, know the optimal policy
  - With value functions, a greedy policy will be optimal no matter the state
- $Q^*(s,\ a)$ = expected utility starting out having taken action *a* from state *s* and (thereafter) acting optimally
- 

- $$V^*(s) = \max_a Q^*(s,a)$$
  $$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$
  $$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$
  ← Bellman Equations

- Definition of “optimal” utility via expectimax recurrence
- Gives a simple one-step lookahead relationship amongst optimal utility values

• **Discount factors gamma:**

- Alters how the algorithm weights near-term rewards vs. long-term rewards. At limit gamma = 0, the agent only care about the instant rewards.
- $0 <= \gamma <= 1$ -- Values of rewards exponentially decay (if not 0 or 1)
- Each time you descend a level in the tree, you multiply the discount once
- Helps algorithms converge (especially with infinite time horizon/state space)
- Smaller discount factor - shorter term focus

• **Value Iteration:**

- $$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

1. Randomly initialize a Q-table.
2. In each iteration, compute Q_{k+1} values for all possible states and actions based on Q_K. Pick the best Q_{k+1} for each state to be V_{k+1}.
3. Iterate until V converges.
   - Converges to optimal value function
   - Lower discount factor - faster convergence
   - $O(S^2A)$ - for each iteration
     - Does bellman equation for every state - look at every action and every possible future state for each action
   - Every iteration updates both the values and (implicitly) the policy but does not track the policy.
   - Needs more iterations than policy iteration before converging because algorithm runs until the values converge. Policy converges way before

• **Policy Evaluation**

$$V_0^\pi(s) = 0$$

- $$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$
- Iterate until values converge
- $O(S^2)$

Policy Improvement (Extraction)

- $$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_i}(s')\right]$$

• **Policy Iteration**

1. Randomly initialize policies for each state
2. In each iteration, compute value function of given policy. (Policy evaluation)
3. Update the policy by picking a new policy that results in a better value function.
4. Iterate until policy converges (**not value**):

- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- Can converge faster than value iteration.
- Value iteration and policy iteration are both offline, model-based planning. They differ only in whether we plug in a fixed policy (policy iteration) or max over actions (value iteration)
- Policy evaluation
  - Calculate utilities for some fixed policy (not optimal utilities) until convergence
- Policy improvement
  - Update policy using one-step look-head with resulting converged (but not optimal) utilities of the future values

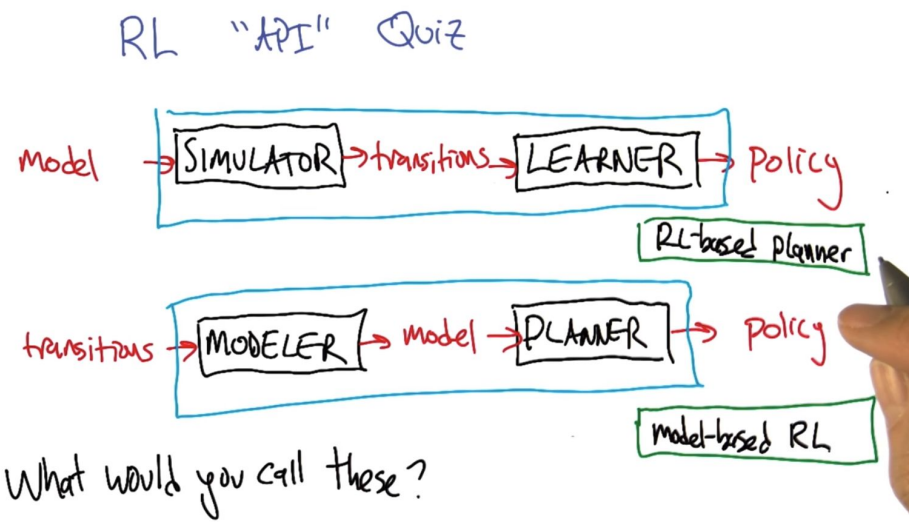
Reinforcement Learning

Definition, difference compared with MDP problems

- Receive feedback in the form of rewards
- Agent's utility is defined by the reward function
- Must learn to act so as to maximize expected rewards
- All learning is based on observed samples of outcomes
- Know the set of states, the actions per state. **NOT transition model or reward function**
  - We don't know which states are good or what the actions do
  - MDP problems know transition model and reward function

• Model-based vs Model-free

- Model-based
  - Learn empirical MDP model
    - Learn an approximate model based on experiences
    - Count outcomes s' for each s
    - Normalize to give an estimate of $\widehat{T}(s,\ a,\ s')$
    - Discover each $\widehat{R}(s,\ a,\ s')$ when we experience (s, a, s')
  - Solve the learned MDP
    - Solve for values as if the learned model were correct
    - Using (e.g.) value iteration
- Model-free
  - Temporal difference learning
  - Experience world through episodes $(s,\ a,\ r,\ s',\ a',\ r',\ s'',\ a'',\ r'',\ s''',\ ...)$
  - Update estimates each transition $(s,\ a,\ r,\ s')$
  - Over time, updates will mimic Bellman updates



Passive Reinforcement Learning

- Policy evaluation
    - Fixed policy
    - Don’t know transition model or reward function
    - Goal: learn the state values
    - Learner is along for the ride”
        - No choice in actions to take
        - Just execute the policy and learn from experience
        - NOT offline planning - you actually take actions
    - Done for every state - takes a lot of data and time
- Direct Evaluation
    - Goal - compute values for each state under the policy
    - Average together observed sample values
        - Act according to the policy
        - Every time you visit a state, write down what the sum of discounted rewards turned out to be
        - Average those samples
    - Pros
        - Easy to understand
        - Doesn’t require knowledge of T, R
        - Eventually computes the correct average values, using just sample transitions
    - Cons
        - Each state must be learned separately
        -
    - Can’t use policy iteration because we need T and R

Temporal Difference Learning

- MODEL-FREE way to do policy evaluation
- Update V(s) each time we experience a transition (s, a, s’, r)
- Likely outcomes s’ will contribute updates more often
- Policy is fixed - doing evaluation

- Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$
- Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha * sample$
- . $V^{\pi}(s) \leftarrow V^{\pi} + \alpha(sample - V^{\pi}(s))$

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

-
- Keeps a running average of values
    - Alpha - weights values we learn later in the process higher (early values are probably incorrect anyways)
- Problems
    - If we want to turn values into a new policy, we're sunk

• **On-policy vs. Off-policy**

- https://stats.stackexchange.com/questions/184657/what-is-the-difference-between-off-policy-and-on-policy-learning
    - With (small) probility epsilon, act randomly
    - With (large) probability 1 - epsilon, act on current policy
    - Random actions keep the agent thrashing around the space once learning is done
    - One solution - lower epsilon over time
- Exploration functions
    - Explore areas whose badness is not (yet) established, eventually stop exploring
    - $f(u, n) = u + k/n$ - estimate *u*, visit count *n*
    - New Q update:

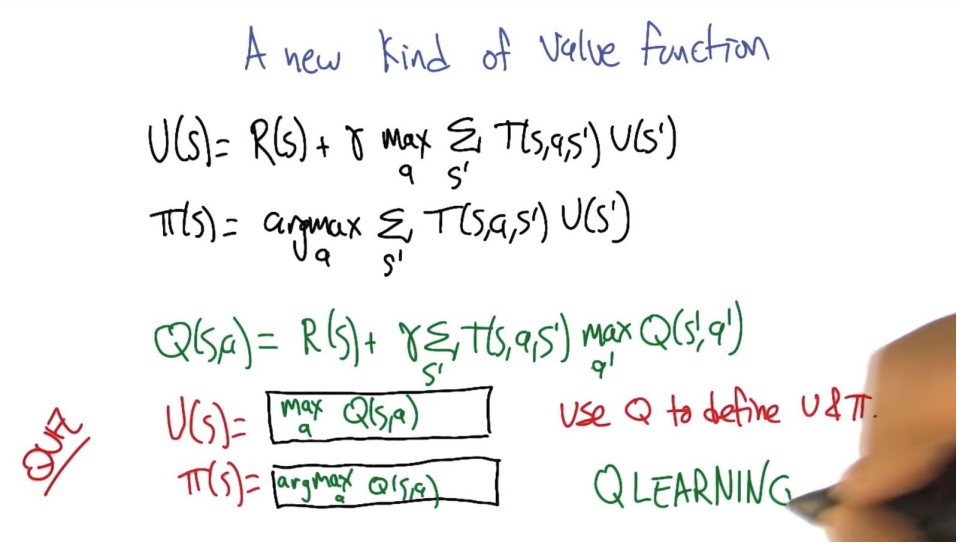
$$Q(s,a) \leftarrow_{\alpha} R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$$

- Exploration means trying random actions, this helps discover the underlying MDP. Exploitation means following the so-far optimal policy, this helps maximize rewards.

• **How is Q value calculated?**

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\left[r + \gamma \max_{a'} Q(s',a')\right]$$

-
- **Alpha closer to 1 - more weight on new sample**
- **Alpha closer to 0 - more weight on past**
- Decrease the learning rate to converge to the average value
    - Don't decrease too quickly because then initially if you have a weird streak, then won't have enough leverage in the future to make up for it



Regret

- Made mistakes along the way
- Regret - measure of total mistake cost
    - The difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimize regret by (e.g.) using exploration functions over random exploration
    - Optimally learning to be optimal
- Regret will never be 0, just about the difference in regrets of different exploration strategies

• **Approximate Q-learning**

- Generalize across states
- Basic Q-learning keeps a table of all q-values
    - Realistically - can't possibly learn about every single state (too many states to visit them all in training)
- Describe state using a vector of features (properties)
    - Features are functions from states to real numbers (often 0 or1) that capture important properties of state
    - Pacman example features:
        - Distance to closest ghost
        - Distance to closest dot
        - Number of ghosts
        - 1 / (dist to dot)^2
        - Is pacman in a tunnel? (0 or 1)
    - Want the features to be general and apply to every state
    - Make sure features are all on the same scale (0 <= feature value <= 1)
        - Normalize it
- Linear value functions
    - Can write a q function (or value function) for any state using a few weights

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

  - For Q(s,a), can also look at f(s')
- Advantage - experience is summed up in a few powerful numbers
  - Only need to learn *{w_1, …, w_n}*
- Disadvantage - states may share features but actually be very different in value
  - Aliasing
  - If 2 different states, no matter how you choose weights, they'll have same q values
  - Comes down to how good / descriptive your features are
  - You want similar Q value states to have similar "goodness"

- Q-learning with linear Q- functions

$$\text{transition} = (s, a, r, s')$$

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}] \qquad \text{Exact Q's}$$

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a) \qquad \text{Approximate Q's}$$

  - Adjust weights of active features only
  - If something unexpectedly bad happens, blame the features that were on
    - If the feature was 0 - that weight did nothing wrong
    - Also allows the sign of the feature to play a role
    - Features with high absolute values contributed a lot, so affect those weights more
  - No Q-value table. When looking up Q(s,a) or Q(s',a') - must use the linear Q-function expression to calculate them

With linear Q-functions: relation to least squares regression

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left(y_i - \sum_k w_k f_k(x_i)\right)^2$$

- Update by the derivative

$$\text{error}(w) = \frac{1}{2}\left(y - \sum_k w_k f_k(x)\right)^2$$

$$\frac{\partial\ \text{error}(w)}{\partial w_m} = -\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

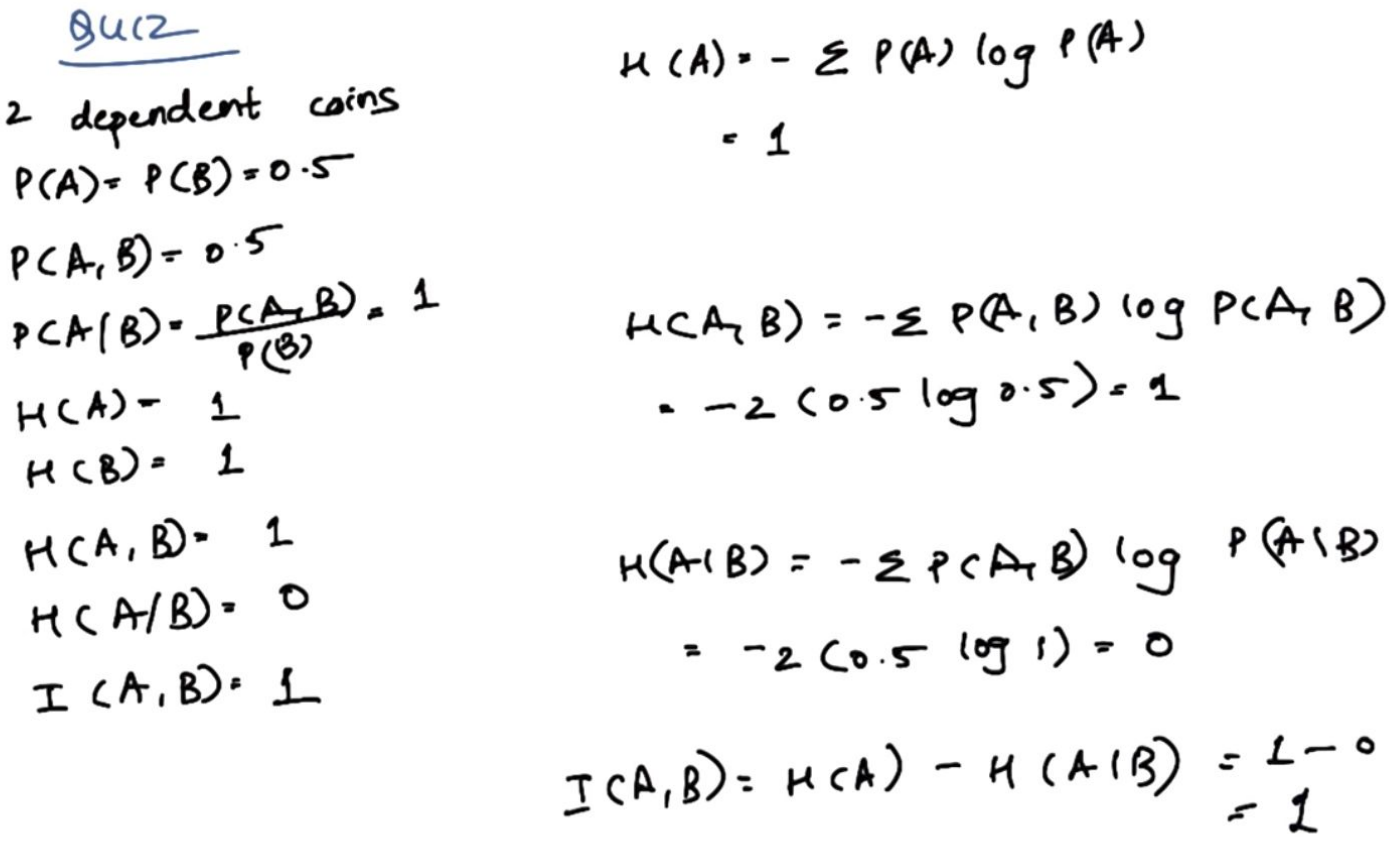
$$w_m \leftarrow w_m + \alpha \left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

  - Error * feature value * alpha

$$w_m \leftarrow w_m + \alpha \left[r + \gamma \max_a Q(s', a') - Q(s,a)\right] f_m(s,a)$$

- Justification for why we have update how it is

Information Theory （don’t think we need this?)



Midterm Solutions/Notes

How this works:

- I take notes by writing down the question and the answer to the problem in 1 sentence.
- If it’s a true or false or some equivalent, bolded words like **can** indicate what the question was
- I need help explaining some solutions. I use the words **also** and **explain** to indicate information in addition to the question. If you see something wrong, just edit it!

3. Linear decision boundaries: Perceptron

Non-linear decision boundaries: polynomial regression, decision tree depth 2, knn

Also I think logistic regression, SVMs with linear kernels and naive bayes produce linear decision boundaries.

5. ID3 can produce suboptimal decision trees.

6. ~~Max depth of decision tree **can** be > # of attributes~~
   ~~**Explain:** You can make more complex boundaries for the same set of attributes~~
   ~~**This is wrong - the max depth will be n-1 (n = # of attributes) if you have a fully complex DT that classifies only 1 sample at a time.**~~
   - I'm pretty sure what I wrote down was correct. Byron's slides have a graph with 2D (ie 2-attributes) data and some crazy complex decision tree that would require several layers.
   - For continuous space, I am sure the same attribute can be used multiple times (Refer to Udacity course video) to narrow down the classification, so yes **it can be > # of attributes**
7. Max depth of decision tree **must** be less than number of training instances.
   Explain: I know ID3 requires instances on either side of a split, which might be what this is referencing, but I don't know why you couldn't just make a massive, randomly optimized decision tree.
   worst case is that each leaf has one instance, and it is a very unbalanced tree, resulting in depth of n.
8. Splits in lower parts of decision trees **are** more likely to be modelling noise. Note that low means farther from root!
9. Gradient Descent **may** converge to a local, non-global optimum
10. Decision trees and logistic regression **can** produce the same decision boundary.
    Also I think any pair of classification algorithms are capable of producing the same decision boundaries given special initializations and data. Maybe perceptron won't? I dunno.

11 - 15: Do the following algos guarantee global optima or merely local optima?

11. ID3 decision trees: local
12. Perceptron: global

Explain: I guess perfect is optimal for perceptron, For perceptrons if the data can be linearly separable then guaranteed to find global optima.

13. Logistic regression: global

Explain: its loss is convex http://mathgotchas.blogspot.com/2011/10/why-is-error-function-minimized-in.html

14. SVM: global

Explain: https://math.stackexchange.com/questions/1127464/how-to-show-that-svm-is-convex-problem
SVM has a convex optimization function → global optimum

15. 2 Layer NN, logistic activations: local

16-18: Which loss function does each use?

16. ID3 Decision tree: zero-one loss

Explain: each instance is either right or wrong

17. Logistic regression: log loss

Also I think this is called cross-entropy loss

18. SVM – hinge loss

Also neural nets and linear regression have loss functions as a hyperparameter. MAE (L1) and MSE (L2) are common. "Exponential loss" is a distractor in these questions.
Linear Regression uses Sum of Squared Errors

Theory and "Theory"

19. expected loss = bias^2 + variance + noise
20. There **are** at least 1 set of 4 points in R^3 that can be shattered by the hypothesis of all planes in R^3

Explain: According to VC dimension the number of points needed to make sure a hypothesis can shatter all of them is equal to the dimension + 1 i.e. need at least d+1 points for a R^d instance space. **Should be at most 4 points?**

21. VC Dimension of 1-NN is infinity

Linear Models

22. If gamma is the weight of regularization term in loss function in linear regression, increasing gamma makes bias increase bias and variance decrease.

23. Linear discriminants:

· Adding new features increases likelihood to overfit

HELP – did I write this down right? If so, why is this true?

· Regularizing model **can but may not** increase performance on testing data **True because It could cause underfitting because it reduces variance and increases bias**

· Adding features to model **can but may not** increase performance on testing data **True because either these new features are advantageous to consider so the learner factors them in, or they're not so it ignores them (i.e. assign weight zero, etc.)**

24-28. probably better to just provide scans

29. If in logistic regression, if $h_{theta}(x) = .8$, $p(y=1|x, theta) = .8$, and $p(y=0|x, theta) = .2$

Neural Nets

30. Neural nets…

· Don't optimize convex objective function

· Can be trained with stuff other than gradient descent

· **Can use a mix of activation functions**

· **Can perform well when # of parameters > number of data points**

31. In NNs, nonlinear activation functions such as sigmoid or tanh..

· Don't speed up backprop compared to linear units - **are you sure about this?**

· **Do help learn decision boundaries**

· Are applied to units other than output

· May output values not between 0 and 1

Also – sigmoid is 0 to 1, tanh is -1 to 1, relu is 0 to inf

32. If you only have linear activation functions, multi-layer networks are **not** more powerful than single layer

HELP – How would you answer this kind of thing if it's not linear? Every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer, so I guess if it's only one layer, then there's no hidden layer.

33. This is a garbage question. The question is as follows, and the "correct" answer is bolded.

Which can be implemented by a Neural Net?

· **Logistic Regression**

· KNN

· SVM

Also when I approached Nolan and said that you could just use a perceptron with hinge loss as its activation as taught in some online classes, he agreed with me, and said that logistic regression is the only answer that "aligned with the framework of neural networks taught in class." - **I mean, it's the simplest version of it I guess.**

SVMs

34-35. I should scan this one too but I won't. The main thing here is that if points are linearly separable and you move one point around, the angle of the decision boundary does not change as long as it is not part of the support vector created by 2 points, since the 2-point support vector determines the angle.
36-38. What happens when you remove a support vector from a training set? I asked Nolan for clarification, and he means you remove the points that make up a support vector in a linearly separable training set. When you do that, the size of the maximum margin can **increase or stay the same**.

Ensembles

39.

· Individual learners **may have** high error rates in good ensemble learners.
· All learners **are not** required to training points **- there's no way this is worded correctly**

Explain – this is bagging

· Ensembles **can** have different types of learners
· Cross-validation **can** be used to tune weights of learners

40. In adaboost, you **can** keep iterating after performance on training data is 0 to improve performance on test data.

What's important is the validation set, not the training set per se.

41. In adaboost, weak learners added in later rounds focus on more difficult instances.
. Difficulty refers to how often an instance is misclassified in earlier iterations.
42. Primary effect of early boosting iterations on classifier ensemble is to **reduce total bias**
43. Effects of later boosting iterations on classifier ensemble is to **reduce total bias and total variance**

Explain – Although increasing model complexity generally increases variance in decision trees, adaboost instead sort of just gets better for no good reason

44. Garbage question.

Naïve Bayes

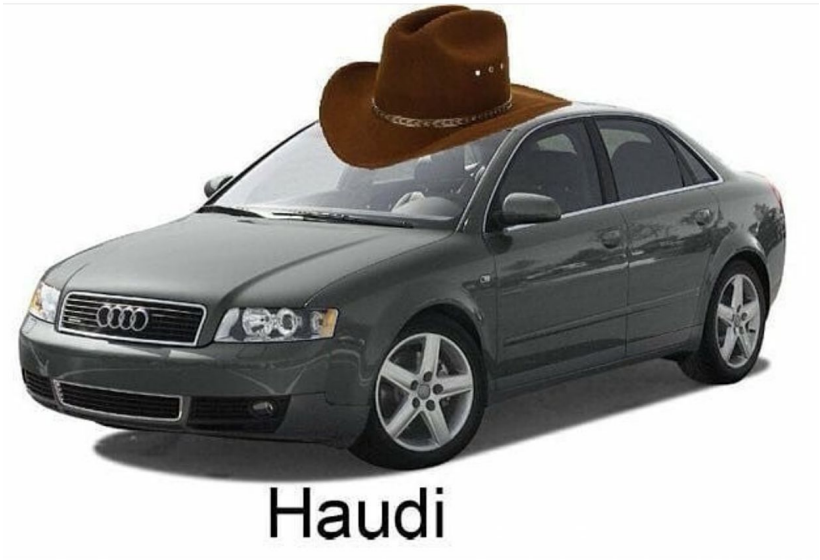
45. If NB and 1-NNeighbor have identical confusion matrix, choose NB since its **decision boundary** is probably better than the weird 1-NN boundary
46. Training naïve bayes classifier with infinite training examples **would not** guarantee zero training or test error

Help – is there anything magical along these lines anyone knows?
https://www.cs.cmu.edu/~tom/10701_sp11/midterm_sol.pdf
See the solution of problem 1.1. Basic idea: it's a probabilistic approach, so nothing is guaranteed.

47. KNN and NB don't both assume conditional independence

Help – can someone explain? lol
http://www.cs.virginia.edu/~hw5x/Course/TextMining-2018Spring/_site/docs/PDFs/kNN%20&%20Naive%20Bayes.pdf
On page 6, it showed that kNN only used Bayes' rule. On page 27-29, it showed NB also assumed conditional independence.
-> **I mean, KNN doesn't really use independence in its algorithm or its decisions. It's just looking at the k Nearest Neighbors.**
NB uses conditional independence for sure. The conditional independence for NB is on the features.
KNN kind of uses conditional independence but it's on samples not on features.

Additional material



I appreciate this^