Kevin Chen

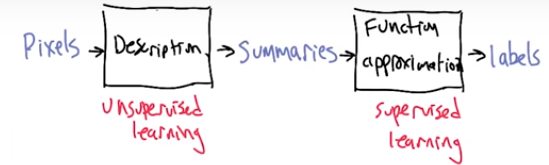
7/24/17

Machine Learning Notes

Instructors: Michael Littman, Charles Isbell, Pushkar Kolhe

**ML is the ROX (Introduction)**

* Definition of ML: the study and building of artifacts (usually computational artifacts) that learn over time based on experience.
* Three parts to this course: Supervised Learning, unsupervised learning, and reinforcement learning
  + Supervised learning: function approximation. Requires a “leap of faith” that function mappings are consistent with the data you are given and that you can generalize.
* Induction and deduction
  + Induction: going from specific examples to a more general rule. This is what supervised learning is.
  + Deduction is the opposite of induction: applying a general rule to a specific example using logic/reasoning.
* Unsupervised learning
  + Given some input, derive some kind of structure in that input.
  + Ex: you see a bunch of animals, and group them into “dog” or “cat” depending on some kind of feature you see in them.
  + Unlike supervised learning, no one is telling you how to divide up the objects in unsupervised learning.
  + Supervised learning is about approximation, whereas unsupervised learning is about description/summarization.
  + Unsupervised learning can work with supervised learning in the following way:



Instead of feeding pixels to supervised learning, you can feed in summaries from unsupervised learning to the function approximator.

* Reinforcement Learning
  + “Learning from delayed reward”: feedback in reinforcement learning may come several steps after the decision is made.
  + Ex: tic-tac-toe: you don’t get any feedback until the very end of the game, where you are told who wins.
* Comparison of These Parts of ML
  + There are some assumptions you make when using these learning techniques. Ex: assume gender is good for classification in unsupervised learning, or assume linear function is appropriate in supervised learning.
  + Both can be framed as an optimization: supervised learning -> label data well, reinforcement learning -> optimize behavior scores, unsupervised learning -> optimize cluster scores.
  + Data is key in machine learning. The algorithm becomes less central.

**Supervised Learning 1: Decision Trees**

Difference between Classification and Regression

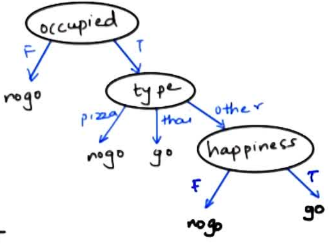
* Two main types of supervised learning: classification and regression
* Classification: input x mapped to discrete label (e.g. True or False)
  + Ex: map image of a person to male or female
* Regression: continuous value function.
  + Ex: you are given a set of points (input, output pairs). Predict the value of a new input using interpolation
* The type of output determines whether it is classification or regression: Classification has discrete options for output, while regression has continuous options for output.
  + Some regression problems can be interpreted as a classification problem with lots of discrete output options and rounding of output values (e.g. age)

Classification Learning

* Instances are vectors of attributes (or values) that define your input space
  + Ex: pixels making up picture
* The concept is the function that maps inputs (instances) to outputs.
  + Concept can be thought of as mapping objects to membership in a set. (Ex: concept of a male or female person)
* Target concept: the actual, correct concept. A particular idea we’re trying to represent
  + Ex: a target concept or function that correctly identifies males vs. females.
* Hypothesis class: set of all concepts (or functions) that we’re going to take a look at
* Sample (aka training set): set of all inputs paired with the credit label
* Candidate: a concept that you think might be the target concept.
* Testing set: a set similar to the training set. Used to test the candidate concept.
  + Training set and testing set should not be the same!
  + Want to demonstrate that you can generalize. Testing (and training) set should be representative of what you will see in the real world.

Decision Trees Representation

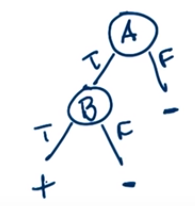
* Input is a bunch of features and output is a discrete set of labels.
* Ex: you want to decide whether to eat at a restaurant or not
  + Input: type (Italian, French, Thai, …), atmosphere (fancy, hole in the wall, casual, …), occupied? (yes/no), cost ($, $$, $$$, …), hungry? (yes/no), weather (raining, sun).
  + Notice that not all inputs are directly related to the restaurant (e.g. weather) but are nonetheless considered (possibly) important
  + Output: enter the restaurant or not
* Decision tree is a representation. We will then use an algorithm to build a decision tree.
* Decision trees are trees where:
  + Parents are decision nodes (represented by a circle). You ask a question about what value a particular attribute is.
  + The answers to the question about the attribute value are stored in the edges coming out of the decision nodes.
  + Leaves are output nodes (represented by a square)
* Example:



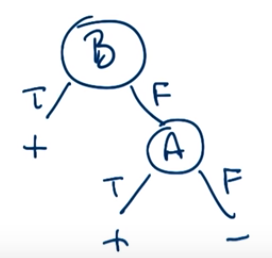
* To classify a particular input, trace starting at the top of the tree and go down.
  + Ex: not occupied, pizza type, and not happy would be classified as nogo
  + Ex: occupied, thai type, and not happy would be classified as go
* The above tree could be considered as a candidate concept, for which we use the above two test cases as a test set to verify the correctness of the candidate concept.

Decision Trees: Expressiveness

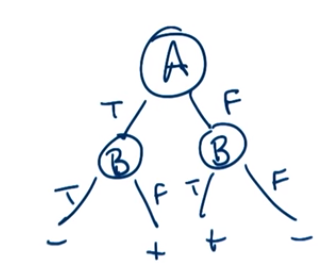
* Think of the 20 questions game. Your goal is to narrow possibilities with each question
  + You want to be able to narrow possibilities even if the answer to that question is “no”. Hence, don’t start with very specific questions.
  + Within the current set of labels that could be given, further narrow the possibilities even more. (Ex, if it’s an animal, you may ask if it’s a human.)
* Decision Trees Learning
  + Pick best attribute: split the possibilities. A good attribute splits the data such that it reduces the number of possibilities remaining for each value of that attribute.
  + Ask question about that attribute
  + Follow the answer path
  + Recurse on the previous steps until you got a single final answer.
* Using Decision Trees for AND Boolean function: A AND B:



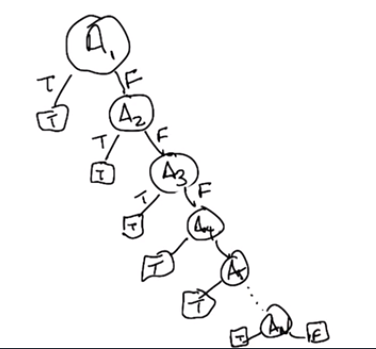
* + (A and B can be swapped)
* Using Decision Trees for OR Boolean function: A OR B:



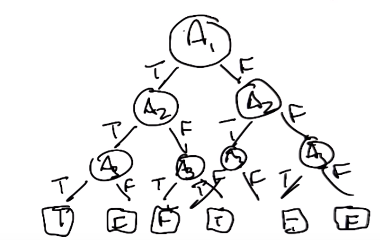
* + (A and B can be swapped)
* Using Decision Trees for XOR Boolean function: A XOR B:



* Notice AND and OR require 2 nodes, whereas XOR requires 3 (which turns out to be substantially more for larger decision trees). See below for more details:
* n-OR: ANY operator: If any attribute is true, then return true. If all false, then return false.



* + Requires “N” nodes for “N” attributes: linear decision tree
* n-XOR: Parity.
  + Even parity: if even number of attributes are true, return true; else return false
  + Odd parity: if odd number of attributes are true, return true; else return false
  + Requires the enumeration of all possibilities in the decision tree: requires nodes for “N” attributes. This is exponential.



* Notice that “any” is much simpler in complexity than “parity”.
* The complexity of the “parity” situation would be simpler if we created random variables where is the parity of . But this technique is kind of “cheating”.
* How many decision trees can you make, given “n” binary attributes and a binary output for each possible combination of attributes?
  + Can create a truth table with rows, 1 row for each of the input vectors. There are 2 choices for the output of each row making options for the output.
  + So there are a lot of decision trees that can be made, and we can’t look at each one of them.

ID3 Algorithm

* ID3 algorithm (for generating the tree):

Loop:

A <- best attribute

Assign A as decision attribute for node

For each value of A:

Create a descendant of node

Sort training examples to leaves

If examples perfectly classified:

STOP

Else iterate over leaves

* What do we define as the best attribute?
  + Information gain is one of the most used solutions
  + Essentially is the reduction of randomness over the labels after knowing a particular attribute

Where S is the collection of training examples, A is a particular attribute, v loops through all values the attribute can take, and are the examples you have with v selected.

* + Entropy is a measure of randomness. The higher the entropy, the more random things are.
  + If you have a lot of examples of one label, but fewer of the other, then your entropy is low. If there are equal number of examples, then high entropy.
* We choose the attribute that has the maximum information gain.
* Restriction vs. preference bias
  + Restriction bias: hypothesis set you care about. Here, we are restricting ourselves to only decision trees.
  + Preference bias: tells us what hypotheses from the hypothesis set we would prefer.
* Inductive bias of ID3 algorithm is related to preference bias.
  + ID3 algorithm prefers good splits at top.
  + ID3 algorithm prefers correct hypotheses over incorrect ones.
  + ID3 algorithm prefers shorter trees to longer trees.

Other Considerations about Decision Trees

* What do we do about continuous attributes? Ex: age, weight, distance
  + One solution: create attributes that contain a range. Ex: “between age 20 and 30?” True or False are the branches that come out of the attribute node.
  + You can then try to decrease the possible range of the attribute by repeating the attribute. This essentially becomes a binary search. Ex: if between age 20-30, now we ask if between 20-25.
* When do we stop?
  + Attempt 1: When everything is classified correctly. But this doesn’t work if we have noise, and for two of the same input objects, we have different outputs. This results in an infinite loop.
  + Attempt 2: No more attributes. Doesn’t work for continuous attributes.
  + You don’t want to overfit the data. This occurs when the tree is too big, and you try to perfectly classify the data. (Issue with attempt #1.)
  + Ways to deal with overfitting: cross-validation (designate a different portion of the set each time as the training set, and pick the tree with the best accuracy), stop building tree after accuracy is good enough, pruning (perfectly classify the data, and them remove some leaves if it doesn’t decrease the accuracy much)
  + Pruning is one of the best ways to deal with overfitting. When deciding the value that a leaf takes, we do a vote on which classification output is the most popular.
* Using decision trees for regression
  + Output is now continuous instead of discrete.
  + Splitting: instead of reducing entropy, you want to reduce variance.
  + Leaves: take the average (or local linear fit) of the output of the relevant examples.

**Supervised Learning 2 – Regression and Classification**

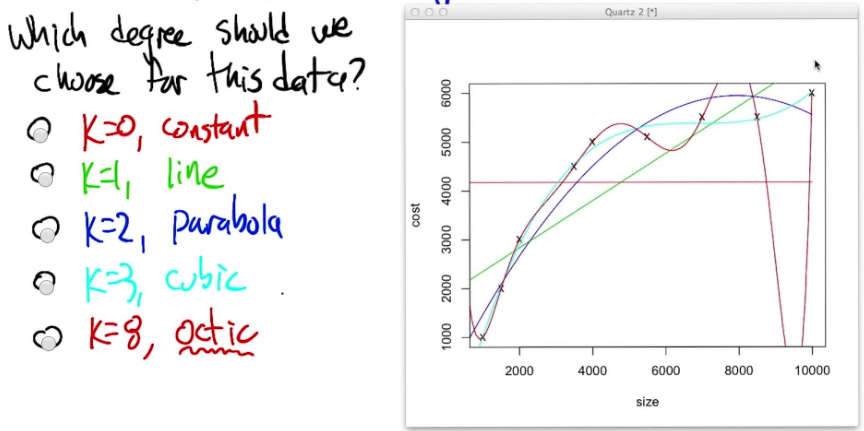
Regression

* Mapping continuous inputs to continuous outputs.
* A linear regression with slope less than 1 is referred to as “regression to mean”
  + This is because the output is closer to the mean compared to the corresponding input.
* Linear regression minimizes the mean square error (average of errors squared across all examples).
  + Can be done using calculus:

Assume you want to fit . The (square) error function is:

Where is the ith training example, with total training examples.

* Finding the minimum of this function is a calculus problem (differentiate it and then set it to zero)
* You can find a perfect fit for a points using a very high order polynomial, but that can lead to overfitting.
  + In the below example, degree 3 is the best choice. is overfitting.
  + Biggest giveaway for overfitting is if there are things happening (e.g. change of slope direction) in the curve between two adjacent inputs.

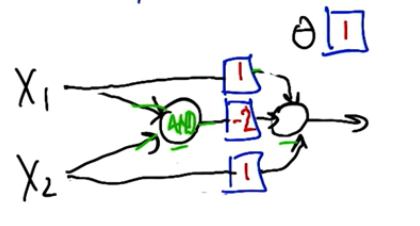


* Polynomial regression can be solved as a matrix equation
  + Assume you have training examples and are trying to fit using a third degree polynomial:
  + Rewrite the above matrix equation to be , and solve for (does not assume X necessarily has an inverse):
* Examples of where errors come from, causing us to model not function but instead the function , where is the error.
  + Sensor errors
  + Maliciously – being given bad data
  + Transcription errors
  + Unmodeled influences
* Cross Validation
  + Recall we want to assume training/test data are representative of the real world. We want to assume that the data is i.i.d from the real world.
  + Our goal is to create a model that is complex enough to fit the data without causing problems on the test set.
  + N-Fold Cross Validation algorithm: Combine the training and test data (or only training data if you don’t have a test set), and divide them equally into subsets, which are referred to as folds. Label these folds as 1, …, n. For each iteration i (from 1 to n), designate fold “i” as the test set and the rest of the folds combined to be the training set. Average the errors from each iteration.
  + Pick the model with the lowest error. Useful for determining things like what degree the polynomial should have.
  + Typically you get a “U” shaped plot between polynomial degrees and cross validation error. Pick the degree with lowest error. Degrees smaller than the best degree are underfitting the curve; degrees above are overfitting.
* Other input spaces
  + Previous examples assume input is scalar, i.e. gives you
  + How do handle vector inputs? Generalize from 2D plot to 3D planes for 2-sized vectors. Generalize to higher dimension for larger vectors.
  + How to handle discrete input?
    - Ex: use whether someone has a job to predict job score. Has job can be encoded as 1, no job as 0.
    - Ex: use hair color as input. Solution 1: RGB numeric values – 3-sized vector. Solution 2: enumerate – red = 1, beige = 2, brown = 3, …

**Supervised Learning 3 – Neural Networks**

Introduction

* A neuron (in your brain) consists of the cell body (the main part), an axon (a long wire connecting the cell body and synapses), and a set synapses (gaps between this neuron and other neurons).
  + Information (spike trains) travel down axon from the cell body when it fires, causing excitation to occur on the synapses to other neurons.
  + This is a kind of a computational unit.
* Artificial neural networks consists of “neurons” that get fired under certain conditions.
  + They are trained under a learning process
* A neuron has inputs , which are multiplied by weights . The activation is equal to . If the activation is greater or equal to the firing threshold (), then the output () is 1; otherwise, output () is 0.
  + This neuron is known as a perceptron.
* Perceptrons can be used to model binary functions. Only a single perceptron is needed for AND, OR, or NOT functions.
  + Ex: inputs models an AND function, as the output is 1 if ; otherwise the output is 0.
  + Ex: creates an OR function.
  + XOR is more complicated: Recall that XOR = OR – AND. So basically we want a normal OR between and and then subtract off a large value if and are true:



Training

* Given examples, find weights that map inputs to outputs
  + Two rules: perceptron rule (threshold outputs) and gradient descent/delta rule (unthreshold outputs)
* Perceptron rule: goal of setting the weights of a single unit so that they capture the data set
  + We have a set of input vectors x, set of corresponding target (expected) outputs y (each output is 0 or 1), set of actual outputs given by the model , and learning rate . Assume we have attributes.
  + There is a clever trick to learn the threshold , To do so, we modify the input vectors : for each input vector in x, we append the value -1 (i.e. we add ) which will be multiplied by weight . The value represents the threshold . This is a clever trick to train the threshold, as we are converting:

The addition is referred to as the bias unit.

* + The algorithm:

Repeat until change in weights () is small/zero or after finite # of iterations:

For each training example :

For each weight :

* + Explaining for formula: represents the difference between desired and actual output values. , the learning rate, controls how fast we update the weight. is used to make the change in to be a percentage of the input value.
* A data set is said to be linearly separable if there exists a linear function that correctly splits all positive examples on one side and negative on the other.
  + The perceptron rule will find such a linear function with finite iterations if the data set is linearly separable.
  + If the data set is not linearly separable, then the perceptron rule will never terminate.
* Gradient descent is more robust to non-(linearly separable) situations
  + Imagine the output of a function is not thresholded, i.e. we now want to make the raw output be close to the expected value .
  + The error function for a particular set of weights is:

where , the activation or raw output, is

* + The constant doesn’t affect the result, but makes the equations easier when taking the derivative, as shown below.
  + We want to know how changing any particular weight changes the error, so that we can change the weights in the direction that minimizes the error.
  + This looks like the perceptron rule.
* Comparison of learning rules:
  + Perceptron rule: . Guarantees finite convergence if linear separable.
  + Gradient descent: . Calculus – robust to non-linear separable. May converge to local optimum
* We don’t do gradient descent using because is non-differentiable (it’s 0 until the threshold , after which it’s 1).
  + But we can make a function that is differentiable and acts similarly: the sigmoid
* The sigmoid function:
  + As . As . So still behaves somewhat like a threshold.
  + “S”-shaped. Gradual transition from 0 to 1, mostly in between x = -5 to x = 5.
  + The derivative of is:
* Neural network structure:
  + Three types of layers: input layer, hidden layers, and output.
  + Hidden layers and the output consist of sigmoid units, each performing operation to generate the output.
  + The whole neural network is differentiable. In other words, we know how changing the weight of any unit will change the output of the entire neural network
  + Back propagation: propagate the errors backwards to learn how to change the weights to make the network produce something more like the answer.
  + There are many local optima, which may prevent you from finding the true best answer.
* Optimizing weights can get stuck at local optima. Alternative advanced strategies include to deal with local optima include:
  + Momentum: continue in the direction you were going, even if it increases error for a bit
  + Higher order derivatives: how multiple weights affect the output
  + Randomized optimization
  + Penalty for “complexity” to avoid overfitting. Penalize too many nodes, too many layers, and/or too large weights
* Restriction bias of neural nets
  + Recall a restriction bias is the representational power (i.e. the set of hypotheses that are being considered).
  + A single perceptron can represent half spaces.
  + Networks and sigmoids allows more complicated functions to be solved. (Not much restriction).
  + Can solve Boolean and with a large enough of a perceptron (thresholded units) network.
  + Can solve continuous functions with a single hidden layer.
  + Can solve any arbitrary function with two hidden layers.
  + To avoid overfitting, we usually restrict number of nodes/layers in neural network. Also can use cross-validation.
  + Large weights can also indicate overfitting.
* Preference bias of neural nets
  + Recall preference bias is the algorithm’s selection of one representation over another.
  + Typically initialize weights of neural networks with small, random values. (Randomness allows variability and avoiding local minima when training neural network. Small values help us avoid overfitting – reduces complexity.)
  + Prefer smaller weights over bigger ones.
  + Occam’s razor: entities should not be multiplied (i.e. made more complex) unnecessarily (i.e. without reducing error). This gives us better generalization error.