UDACITY

Supervised Learning

Skill: 2/3

Time: 2 months

Perquisites: probability, statistics, linear algebra, programming, little NN.

Instructor: Professor Charles Isbell (Georgia Tech), Professor Michael Littman (Brown University).

Summarized by Ido Greenberg

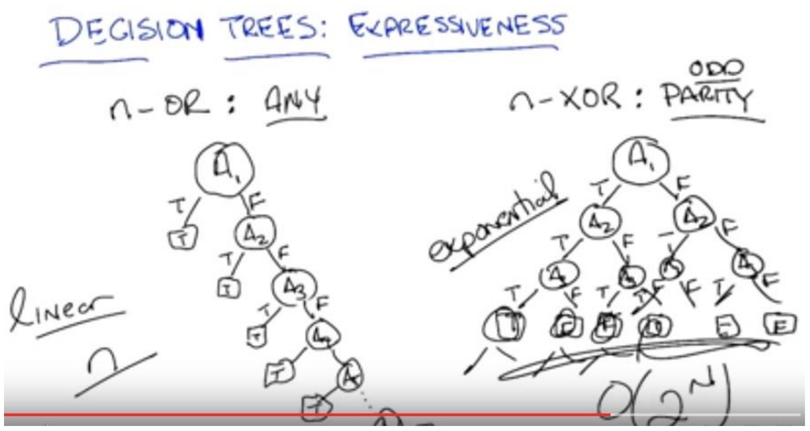
Contents

Introduction

1. Supervised = approximating functions / function induction (generalization of examples)
2. Unsupervised = describing
3. Reinforcement = behave; act while having partial feedbacks

Decision Trees

1. Decision trees are representation of classification problem based on discrete input and output.
2. They essentially just represent switch-case programming as a tree.
3. ***Decision nodes*** represent questions (usually binary), and edges represent possible answers. The next decision node is determined by the answer, and the leaves of the tree are the possible decisions/outputs.
4. When many outputs are initially possible, we wish to build the questions such that each node would split the data as equally as possible. This makes the decision tree kind of a binary search, allowing to find the correct output within minimum steps in the worst case.
5. Example: Boolean functions such as A^B can be represented by trees.
6. ***Size*** of tree = number of decision nodes as function of the input size.
    a. For example, n-or tree has size n, whereas n-xor tree has size 2^n (see below).



    b. Note: using the new variable $B = \sum A_i$, the n-xor becomes trivial (just check odd or even). That demonstrates that **good learning often requires good representation**.
7. <u>Number of possible trees</u>:
    a. Assume N binary input variables, and one binary output variable.
    b. Number of possible decision trees = number of different functions from input to output = |output space|^|input space| = **2^(2^N)** (which is ~10^19 for N=6).
8. Note that every decision node removes a part of the input space, leaving only the input entries corresponding to the chosen answer. The tree solves the classification problem if every leaf corresponds to input entries of the same class / with the same label (that label is the output corresponding to the leaf).
9. ***ID3***:

   a. That's an algorithm for building a decision tree.
   b. The algorithm: while there's a leaf corresponding to non-homogeneous input: assign the "best attribute" to this leaf (transforming it into a decision node) and create a child node for each one of the possible answers.
   c. The best attribute can be defined by maximization of the ***information gain*** – the entropy of the original node minus the (weighted) average entropy of the new nodes. This is equivalent to minimization of the randomness of the labels of the remaining data.
      i. ***Entropy*** ~ randomness (will be well-defined later).
      ii. The score of a decision node splitting uniformly-distributed n-sized data into l and n-l, can be defined by the expected amount of data in the child node: $\frac{l}{n}l+\frac{n-l}{n}(n-l)$, which maximal when $l=n-l$.

10. ***Inductive bias***: bias of the generalized model:
    a. ***Restriction bias*** – the bias derived from the tree-representation:
       i. Preventing the use of many possible functions.
       ii. Limiting us to discrete variables.
    b. ***Preference bias*** – the bias derived from our preferences (expressed through the learning method) – which trees we choose to prefer upon others. In the ID3 as defined above, we prefer trees which are:
       i. Correct wrt the training data (stop condition).
       ii. Have "good" splits at their top, hence shorter (best attribute definition).
11. Stop condition:
    a. Current stop condition (perfect classification of training data) may be impossible (if noise creates two identical objects with different labels) or non-beneficial (***overfitting***).
    b. We can use ***cross-validation*** to generate alternative stop condition – when the additional attributes do not reduce the generalization error.
    c. ***Pruning*** is a similar idea – finish creating the full tree according to training data, then check how collapsing nodes together would affect the validation set. If it reduces the generalization error, then collapse to cancel the overfitting. I didn't get why it's supposed to be more efficient than the previous one, and how it deals with the problem of impossible classification of certain training sets.
12. Continuous variables:
    a. Input: continuous variables can be handled by discretization through ranges of values.
    b. Output: this is actually regression rather than classification. Decision trees are not natural for this problem, but can be partially handled: the information gain of the "best attribute" should be defined in terms of variance in the resulted data (rather than entropy), and the final outputs can be defined as the mean of the corresponding output data or something like that.

Regression

1. The name is stuck in spite of not being very representative anymore.
2. Choosing the correct degree of polynomial regression is an art considering fit error versus complexity and overfitting.
3. Solving ***least squares*** for m-degree polynomial regression of n data points is just solving a linear system of $n \times m$.
4. Typically $m \ll n$, so the solution is not perfect. Minimizing the mean squared error (***MSE***) is achieved by $\boldsymbol{w} = \left(\boldsymbol{X^T X}\right)^{-\boldsymbol{1}} \boldsymbol{X^T Y}$, where $X_{i,j} = x_i^{j-1}$ and $Y_i = y_i$.
5. Note: **additional variables** are just additional degrees of freedom to the system, same as higher polynomial degrees.
6. ***Cross validation***: method for overfitting prevention:
   For every model (e.g. for every polynomial degree):
   a. Split training data into, say, 4 groups.
   b. For every group i, train on the other groups $j \neq i$, then compute the generalization error e_i on the chosen group.
   c. Compute the average generalization error over the 4 groups $< e_i >$.

   Finally, pick the model with the smallest generalization error.

Neural Networks

1. Inspired by networks of neurons.
2. ***Perceptron*** = the specific 1-layer NN architecture with binary output.
3. ***Gradient Descent*** (***GD***)…
    a. For linear separator, GD is impossible for the binary threshold (non-differentiable) and not effective for the linear form (since we just need to get beyond the threshold, we do not benefit linearly from keeping enlarging the weights after that).
    b. Thus the ***sigmoid*** wrapper $\frac{1}{1+e^{-x}}$ is typically used for training, and replaced by binary threshold for predicting.
4. ***Perceptron rule***: a primitive variant of GD, using differences rather than differentials.
    a. Limited to Perceptron training (classification), and has problems with stop condition (there is no derivative size indication).
    b. For ***linearly separable*** data, it guarantees **convergence to perfect separator** (not counting margins minimization), thus it is very useful in that certain case. Note that the linear separability has to be known in advance, otherwise the learning process becomes infinite.
5. <u>Multiple layers</u>:
    a. Analog and generalized from the classic Perceptron.
    b. Differentiable activators in the middle layers are necessary.
    c. Training is possible (***backpropagation***), but convergence is not guaranteed, since multiple local optima are possible.
        i. Note that for one layer, the MSE of the linear multiplication is kind of paraboloid; which is not the case for the non-linear architecture.
6. Advanced learning/optimization methods:
    a. Adding ***momentums*** to GD: this adds a term of momentum to the derivative in the GD expression. The momentum encourages movement in the same direction as before, allowing getting over local minima against small hills, without enlarging the learning rate.
    b. Higher order derivatives (e.g. Hamiltonian) are used in certain ways to improve weights updating.
    c. ***Randomized optimization*** can be more robust and explained in another course (unsupervised learning?).
7. <u>Prevention of overfitting</u> (***regularization***):
    a. Simple architecture: less layers, less nodes.
    b. ***Complexity penalty*** to loss function (it is claimed that complexity is derived from the sizes of the weights, not only the number of the weights).
8. <u>Inductive bias</u> of NN:
    a. <u>Restriction bias</u>: the lecturer claims (not proving) that the representation of NN is very **general and non-restrictive**. In particular:
        i. Boolean functions can be represented by simple NN with thresholds.
        ii. Continuous functions can be represented using 1 hidden layer and enough nodes.

iii. All functions(??) can be represented using 2 hidden layers and enough nodes.

The powerful representation increases the danger of overfitting and modeling the noise, which is a central issue in NN. Regularization methods are clearly summarized in the AI course summary.

b. Preference bias: determined mainly by the method of updating the weights (GD with possible additions – just preferring good fit to the training data) and by the initialized weights. The initialized weights are typically random (increasing variability and insensitivity to start point) and small (unclear small wrt what scale; anyway, small initial weights are supposed to encourage smaller final weights, i.e. “simpler” explanations and functions).

Instance Based Learning

1. Until now there was a clear separation:
    a. Learning = training data → function.
    b. Application = new data & function → predictions (without using the training data directly).
2. Memorizing the training data in a table of {(xi,yi)}, and predicting by just looking up for x in the table:
    a. Pros:
        i. Reliability to the training data – it gives exactly the empirical observations, rather than some approximated value ("it remembers. It's like an elephant").
        ii. No learning process required.
        iii. It is simple and non-parametric ("bacon with chocolate is delicious").
    b. Cons:
        i. No generalization (deals only with the points existing in the training data).
        ii. Complete overfitting (noise is modeled with the signal all the same).
    c. Terminology: for each query we look for the same instance in the training data, hence instance-based learning…
3. ***KNN*** (explained in AI course) solves the cons as follows:
    a. Generalization is achieved by looking for the nearest neighbors, and not necessarily the exact same point.
    b. Overfitting is reduced by using multiple neighbors, so that noise in a single data point does not cause wrong prediction.
4. Note: the definition of ***distance***/***similarity*** for finding the nearest neighbors is for the analyst to determine. That's the heart of applying the KNN.
5. KNN algorithm summary:
    a. Training data: {(xi,yi)}
    b. Hyper parameters: distance metric or similarity function, K
    c. Parameters: non
    d. Predicting:
        i. Classification: most frequent class among nearest neighbors ("voting")
        ii. Regression: mean over nearest neighbors
    e. Tie breaking:
        i. In voting: choose the more common class, random choice, remove farthest neighbor, etc.
        ii. In neighbors distances: if needed, take more than K neighbors – until no additional equally-close data points are available.
6. Possible generalization is ***weighted KNN*** which does the voting/averaging among nearest neighbors with weights expressing how close the neighbors are to the new point (w=similarity or w=1/distance, according to the terms of the problem). This both makes sense (larger weights to more significant points) and helps to break voting ties.
    a. Note that with weighted KNN, it does make sense to use K=n.
    b. If we choose K=n with weighted average, we can also replace the average by any other estimator, e.g. applying linear regression with weighted square errors. This

is called ***locally weighted regression***, and note that it requires computing the regression on every query. But since every query point has its own regression line, this method actually permits non-linear complicated functions.

7. Computational resources comparison (for simplicity, training data are assumed to be in R^1 and sorted):

		Running Time	**Space**
KNN	Learning	1	n
	Query	Log(n) + K	1
Linear Regression	Learning	n	1
	Query	1	1

   a. Note that KNN leaves all the work for the query, whereas LR does the work in advance. This property creates a natural separation between ***lazy learners*** and ***eager learners***.
8. Preference bias: locality (close points are similar), smoothness (since we typically take average for regression). The chosen metric expresses additional preference biases.
9. ***Curse of dimensionality***: for constant density of data points (meaning constant accuracy of the KNN method), the data size has to grow exponentially with the number of dimensions ("exponentially means bad"). This is problematic both for collecting the data and for implementing the query (see running time above).
   a. If some of the dimensions are degenerated or known to be much less important, then a ***weighted metric*** (assigning higher weights to the dominant dimensions) can partially deal with the curse.

Ensemble B&B

1. Idea: generate simple heuristics based on subsets of the data, then combine them.
2. Note: The difference from other methods is that here we first generate the components independently, then combining them into a complete classifier (as opposed to NN, for example, in which we first have an architecture, then determine the parameters).
3. Simple classifiers generation:
   a. The basic classifiers/rules/indicators/heuristics are intended to be as simple as possible, and don't have to be more than a little informative.
   b. To generate such simple heuristics, we just implement some learning on different random subsets of the data.
4. Combining the classifiers:
   a. Regression – can just average the heuristics outputs (***bagging*** – each random subset is a "bag", or ***bootstrap aggregation***).
   b. Classification – voting.
5. The picking of random subsets turns out to increase the robustness of the learning – probably since noise, outliers and anomalies do not affect most of the basic heuristics used. Thus, ensemble learning is **inherently good at generalization and overfitting prevention**.
6. ***Boosting***:
   a. When picking data subsets for new heuristics, give higher priority to "difficult" data samples, based on which data samples went wrong in previous heuristics.
   b. Combining the heuristics uses weighted average/voting –according to the classifiers success on the original data (without taking into account the relevance of each classifier to the new query).
      i. The weighting is intended to prevent loosing track of data samples that we already handle correctly, even when we learn mainly from the remaining "difficult" data.
7. Terminology:
   a. Error: rather than MSE, in boosting for classification we use the percent of wrong classifications as error indicator (which is – for 0/1 representation – the same up to normalization).
   b. ***Weak learner***: learner that does better than chance (i.e. >50% correct classifications), independently of the distribution of the data. In other words: for every distribution of the data (i.e. every list of weights assigned to the data points), the learner will be able to find some hypothesis/parameters that yield better-than-chance classification.
8. Boosting assumes the existence of weak learner on the given data. This assumption requires the representation of the "brain" to be strong enough, such that for every distribution of the data, the representation would allow some hypothesis which is good enough.
   a. The robustness wrt all possible distributions is required due to the way in which we pick subsets of the data for learning.
9. Boosting classification algorithm:
   a. Input: $\{(x_i, y_i)\}$, $y_i \in \{\pm 1\}$.

   b. For t=1:T
      i. Generate distribution D_t over the data.
      ii. Find weak classifier $h_t(x)$ with error $\epsilon_t = P_{D_t}[h_t(x_i) \neq y_i] < \frac{1}{2} - \epsilon$.
   c. Output: H combining all weak classifiers.
10. The distributions are generated to stress the difficult examples:
   a. $D_1(i) = \frac{1}{n} =$ uniform distribution over all the training data.
   b. $\boldsymbol{D_{t+1}(i)} = \frac{\boldsymbol{D_t(i) \cdot e^{-\alpha_t y_i h_t(x_i)}}}{\boldsymbol{z_t}}$, where $\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$, z_t =normalization. Thus:
      i. The new distribution is based on the previous one.
      ii. Every sample gets additional weight if the previous classifier failed on it, and reduced weight if the previous classifier was successful on it.
      iii. The changes in the weights are more drastic as the last classifier was better. Note that if $\epsilon_t > \frac{1}{2}$ then $\alpha_t < 0$. That's not supposed to happen (since we assume existence of weak learner for any distribution), but it's fine, since in the final classifier we will prefer to use such classifier as anti-indicative.
11. The final hypothesis is just a weighted average of the classifiers:
   a. $\boldsymbol{H_{final}(x) = sign(\sum_t \alpha_t h_t(x))}$.
12. <u>Note</u>:
   a. The assumption of existence of weak learner for every distribution is not trivial and I think it's quite difficult or even not practical to verify in advance (although they didn't refer to that).
   b. The fact that the final classifier is "good" is not trivial at all. There's a proof for that (I'm not sure how it defines "good"), but the intuition is not trivial as well. It seems that the new classifiers solving the "hard" examples could do worse with the previous example, but somehow this process keeps doing aggregately better over time.
   c. The expression of H_{final} can make a complicated classifier out of the simple basic classifiers. Note that although it is just a linear combination, each basic classifier is non-linear by itself, due to the binary threshold required for classification. Thus it's linear combination of non-linear functions, which can create a complicated classifier.
   d. "Boosting" is for taking weak classifiers and boosting them into a strong classifier.
13. Additional material:
    https://storage.googleapis.com/supplemental_media/udacityu/367378584/Intro%20to%20Boosting.pdf

Kernel Methods and SVMs

1. Linear binary classifier: $y = sign(w^T x + b)$
2. ***Margin maximization***:
   a. For linearly separable training data, in addition to correct classification, we wish to maximize the margins of the classifier, i.e. the minimal distance of the classifier boundary from the data points.
   b. This goal makes sense due to locality – we believe that close points tend to have the same labels, thus we don't want any of the known training points to be near the boundary. Small margin may be seen as overfitting, since the classifier is designed to fit the exact training data, without thinking about generalization.
   c. If we somehow normalize the hyperplane equation such that the closest two points to the plane will satisfy $w^T x + b = \pm 1$, then the margin will be $\frac{2}{||w||}$.
3. Formalization of margin maximization:
   a. **argmax**$\boldsymbol{\frac{2}{||w||}}$ **s.t.** $\forall \boldsymbol{i}: \boldsymbol{y_i}(\boldsymbol{w^T x_i} + \boldsymbol{b}) \geq \mathbf{1}$
   b. That forces correct classification with $w^T x + b$ normalized wrt the data points, and maximizes the margin under this constraint.
   c. $argmax \frac{2}{||w||} \leftrightarrow argmin \frac{1}{2} ||w||^2$. The latter is a ***quadratic programming problem***, which is solvable and known to have a unique solution.
   d. An equivalent, easier problem turns out to be:
      i. **argmax** $\boldsymbol{W(\alpha)} = \sum_i \boldsymbol{\alpha_i} - \frac{1}{2}\sum_{i,j} \boldsymbol{\alpha_i \alpha_j y_i y_j x_i^T x_j}$
      ii. **s.t.** $\boldsymbol{\alpha_i} \geq \mathbf{0}$ **and** $\sum_i \boldsymbol{\alpha_i y_i} = \mathbf{0}$
      iii. Transforming back to the original problem: $\boldsymbol{w} = \sum_i \boldsymbol{\alpha_i y_i x_i}$, $\boldsymbol{b} = -\boldsymbol{w^T x} - \mathbf{1}$ (where x is the data point with positive label +1 that minimizes $w^T x$).
4. It turns out that typically $\alpha_i = 0$ for most i's, thus only few data points (support vectors) are needed to define w (build the machine). Thus this concept is called ***support vector machines***. The supporting points are the ones close to the separator, since they determine where this separator has to lay.
   a. According to this, SVMs are neither exactly eager nor lazy: they do study in advance, but don't use all the data points explicitly.
5. When the data is not linearly separable, we wish to transform the data into a higher dimension space, yielding coordinates that allow separation.
   a. For example: $\Phi(x \in R^2) \coloneqq \left(x_1^2, x_2^2, \sqrt{2}x_1 x_2\right)$ gives $\Phi(x)^T \Phi(y) = \cdots = (x^T y)^2$, which somehow represents a circle, thus allows classification by radius rather than by direction (note that linear classification essentially uses the direction of input). This particular representation of a circle is useful, since the quadratic programming problem only needs to compute the scalar product $x_i^T x_j$, which in this case stays the same under the transformation (up to squaring), so there's no need to compute the whole transformation. This idea is called the *kernel trick*.
6. Kernel:

a. In argmaxW(α), the data points appear only once as $x_i^T x_j$, which kind of measures the similarity between the two points. In general, we can replace this term by $\boldsymbol{K(x_i, x_j)}$, which is called ***kernel***.
b. In the example above we use $K(x,y) = (x^T y)^2$, which implicitly represents the transformation Φ (defined above).
c. Common kernels:
    i. $K = (x^T y + c)^P$
    ii. $K = \exp\left(-||x-y||^2/2\sigma^2\right)$
    iii. $K = \tanh(\beta x^T y + \theta)$
d. Eventually, the important thing about the kernel is to really represent similarity, capturing our ***domain knowledge*** (i.e. what we understand about the domain and its structure).
e. Necessary condition for valid kernel function: ***Mercer Condition***: K “acts” like distance/similarity. They didn’t say explicitly, but maybe it has to be some kind of metric?

7. SVM summary:
    a. **Margin maximization** using equivalent quadratic programming problem.
    b. **Implicit coordinates transformation** using informative similarity function – **kernel**.
8. Note: since boosting always focuses on the “hard” examples, i.e. those which are close to the margins of the last classifiers, it turns out that additional boosting iterations actually widen the margins. This explains why boosting tend to avoid overfitting!
    a. **Overfitting in boosting is possible if the basic classifiers are too complicated** and cause overfitting by themselves (e.g. NN with too many layers & nodes). However, the boosting process (combination of weak classifiers) does not cause overfitting.
    b. Another thing causing overfitting in boosting is pink noise(?).

Computational Learning Theory

1. Resources in machine learning: time, space (as in every algorithm), data required for learning.
2. Inductive learning = learning from examples. Properties of inductive learning algorithm:
    a. Probability of successful learning $1-\delta$
    b. Number of examples required for learning m
    c. Complexity of hypothesis class H (determining the tradeoff between restriction bias and overfitting potential)
    d. Target deriving the approximation? ϵ
    e. How training examples are presented (online, batch…)
    f. How training examples are selected (chosen by learner, generated randomly by nature, chosen by some positive "teacher", chosen "evilly" – worst case)
3. Manner of training examples selection:
    a. How many nodes are required for a search tree with hypothesis space H?
        i. Teacher determining the nodes → 1 (Which problems are adversary?)
        ii. Learner determining the nodes → $\log_2|H|$
        iii. Constrained queries would increase the required nodes in both cases…
    b. Examples chosen by nature require the best robustness (here evil choice is just a private case of nature).
4. Definitions:
    a. ***Computational complexity*** = how much computational effort is needed for a learner to converge (to the best hypothesis in the hypotheses class))?
    b. ***Sample complexity*** (for batches) = how many training examples are needed for a learner to create a successful hypothesis?
    c. ***Mistake bounds*** (for online examples) = how many misclassifications can a learner make over an infinite run?
    d. ***Training error*** = rate of misclassified examples among training data.
    e. ***True error*** = rate of misclassified examples among all data / according to the data distribution.
5. ***Probably approximately correct*** (***PAC***) learning:
    a. We want our learner to be correct. However, it can't be exactly correct for all the examples, so we allow true ***error goal*** $0\le\epsilon\le\frac{1}{2}$ (for binary classification), and we say that it's approximately correct. But if we're unlucky and the training data does not represent the inputs distribution so well, then we may fail to achieve our error goal. Our ***certainty goal*** is that this would happen with probability $0\le\delta\le\frac{1}{2}$, and we say that our learner is probably approximately correct.
    b. A problem ("concept class") C is ***PAC-learnable*** by a learner L using hypothesis space H iff with probability $1-\delta$, L will output a hypothesis h with true error $err_D(h)\le\epsilon$, in time & samples polynomial in $\frac{1}{\epsilon},\frac{1}{\delta},|H|$.
    c. ***Version Space*** (***VS***) = all the hypotheses which are consistent with the training data.
    d. ***ϵ-exhausted*** = VS in which every hypothesis has true error smaller than ϵ.

e. We want our VS to be ϵ-exhausted. ***Haussler Theorem*** shows that given m examples in the training data, the probability of the VS to contain hypothesis with true error larger than ϵ (i.e. not to be ϵ-exhausted) is bounded: $P \leq |H| \cdot (1-\epsilon)^m \leq |H|e^{-\epsilon m}$.

f. Thus, requiring $P \leq \delta$ (i.e. that the VS will be probably ϵ-exhausted), a sufficient condition turns out to be $\boldsymbol{m} \geq \frac{\mathbf{1}}{\boldsymbol{\epsilon}}\left(\mathbf{ln}|\boldsymbol{H}| + \mathbf{ln}\frac{\mathbf{1}}{\boldsymbol{\delta}}\right)$, which is polynomial as needed.

   i. This bound grows infinitely if |H|=inf.

VC Dimensions

6. Infinite hypothesis spaces:
   a. Infinite hypothesis space makes the bound of Haussler Theorem irrelevant.
   b. Many hypothesis spaces are infinite (space of linear separators, space of NNs, space of continuous decision trees…).
   c. Sometimes although the formal hypothesis space (***syntactic***) is infinite, there are only finitely many hypotheses that really behave differently on the input space (***semantic*** hypotheses).
      i. For example, in $X = 1{:}10$, $H = \{h(x) = x \geq \theta\}_{\theta \in R}$, the hypothesis space would stay essentially the same if $\theta \in \{1 \dots 10\}$.
7. ***VC dimension = Power* of hypothesis space** = maximal size of inputs set that the hypothesis class can label in all possible ways (can "***shatter***") (i.e. the largest input for which the hypothesis space allows complete robustness).
   d. To be clear: $argmax_k$(there <u>exists inputs set</u> of size k, s.t. for <u>every labeling</u> of the inputs in the set, there <u>exists a hypothesis</u> in the class which is consistent with the inputs labeling).
   e. For example, linear separator in R^2 has VC=3 (with 4 points one can form the XOR problem which is not linearly separable; other arrangements of the points locations have similar problems.).
   f. In general, it turns out that for any hyperplane hypothesis class (i.e. linear separator) in R^d, VC=d+1. That's also the **number of parameters in the class, which indeed often equals the VC dimension**.
   g. The class of polygons in R^2 has VC=inf as separators. It is elegantly demonstrated by arranging the points on the edge of a circle.
8. It turns out that the <u>amount of data needed for learning</u> (*sample complexity*, see above) <u>can be derived from the VC dimension</u> (named after some 2 guys) of the hypothesis class, even when the hypothesis space is infinite.
   h. Expressively: $\boldsymbol{m} \geq \frac{\mathbf{1}}{\boldsymbol{\epsilon}}\left(\mathbf{8}\boldsymbol{VC}(\boldsymbol{H}) \cdot \mathbf{log_2}\frac{\mathbf{13}}{\boldsymbol{\epsilon}} + \mathbf{4}\,\mathbf{log_2}\frac{\mathbf{2}}{\boldsymbol{\delta}}\right)$.
   i. An intuition can be "m = constraints required = DoF allowed = parameters = VC".
   j. Note that essentially, |H|~2^n$_{\text{parameters}}$, thus it makes sense that in the infinite case VC(H) replaces log|H|.
9. <u>Theorem</u>: **H is PAC-learnable iff VC(H) is finite**.

Bayesian Learning

1. ML goal: learn the "best" hypothesis given data & some domain knowledge.
   a. ***Bayesian learning***:
      i. "best" = most probable.
      ii. In Bayesian terms: maximizing the posterior $P(h|D)$.
      iii. The data influences the posterior through the likelihood $P(D|h)$.
      iv. The domain knowledge influences the posterior through the prior $P(h)$.
2. Conceptual algorithm:
   a. For each hypothesis h in H, calculate $P(h|D)$ using Bayes rule, then return the one with highest probability (***Maximum a Posteriori*** hypothesis, or ***MAP***).
   b. For uniformly distributed prior we have $P(h|D)\sim P(D|H)$, hence it's just ***Maximum Likelihood*** (***ML***).
   c. Anyway, the impractical part here is iterating over H.
3. <u>Bayesian learning simplified model</u>:
   a. Training data $D=\{(x_i,d_i)\}$ are **noise-free**.
   b. The correct hypothesis is in our space ($\boldsymbol{c\in H}$).
   c. **Uniform prior**: $P(h)=\frac{1}{|H|}$ ("uninformative" prior).
   d. Noise-free ➔ deterministic observations ➔ $P(D|H)=1\ if\ \forall i\in data: h(x_i)=d_i, else\ 0$.
   e. $P(D)=\frac{|VS|}{|H|}$ is given by normalization – how many hypotheses remained in the Version Space.
   f. ➔ $\boldsymbol{P(h|D)=\frac{1}{|VS|}\ if\ h\in VS, else\ 0}$.
4. <u>Noisy model</u>:
   a. $d_i=f(x_i)+\epsilon_i$ ($\boldsymbol{\epsilon_i\sim N(0,\sigma)}$ are i.i.d.)
   b. $\boldsymbol{h_{ML}}=argmax_hP(h|D)=argmax_hP(D|h)=\boldsymbol{argmin_h\sum_i\left(d_i-h(x_i)\right)^2}$
   c. Thus we maximize the posterior by minimizing the negative log-likelihood, which is – for normally-distributed errors – equivalent to minimizing the MSE (or SSE).
   d. Note: we derived the **Least squares** principle from Bayesian approach for normally-distributed errors! In particular, for linear hypothesis class, we have that **Linear Regression** solves the Bayesian learning problem!
      i. On the other side, any other noise model will yield a different choice of hypothesis, so it's **very important to choose a reasonable noise model**.
5. Non-uniform prior model:
   a. General ***negative log-likelihood*** minimization:

$$h_{MAP}=argmax_hP(h|D)=argmin_h(-logP(D|h)-\log P(h))$$

   b. Note: <u>information theory and entropy notion</u> say that $-logP$ is exactly the length of an event with probability P, where the length can be seen as the number of bits needed to represent the event.
      i. Len(h) refers to the encoding of the hypothesis.
      ii. Len(D|h) refers to the additional description needed in order to explain the data D, given that we already know the hypothesis h.

   iii. So actually we wish to minimize the length of the explanation of the data using a hypothesis, i.e. to find the simplest explanation to the data. That's a kind of Bayesian argument for ***Occam's razor***.
   iv. This is called ***Minimum Description Length***.
   v. To apply this approach, the prior P(h) has to be defined. The choice of the prior is essentially the decision that determines the trade-off between the preference of simpler/better hypotheses and the tolerance to errors.
   vi. Note: the description length minimization approach explains why **large parameters in NNs are claimed to create overfitting all the same as lots of parameters**.

6. <u>Prediction</u> – Bayesian inference:
   a. In order to have the best prediction of labels, it's not enough to find the most probable hypothesis (posterior maximization) and compute $d = h_{MAP}(x_{new})$.
   b. Rather than that, it is needed to take all the hypotheses into account and do ***Bayesian inference***: $argmax_d\left[\sum_{h:h(x_{new})=d} P(h|D_{train})\right]$. This can be seen as a **weighted vote**, where the weights are the posteriors of the hypothesis given previous data.

Bayesian Inference

1. Bayesian inference: "representing and reasoning with probabilities".
2. Bayesian networks: "representing and manipulating probabilities over complex spaces".
   a. This was well taught in the AI course of Thrun. In particular the efficiency of the network representation compared to full joint-distribution was demonstrated.
   b. The basis to Bayesian networks is conditional independence between two variables X,Y given a third variable Z: $P(X|YZ) = P(X|Z)$.
      i. That's a simple case of joint-distribution, but it's usually the case in many applications modeled by structural dependencies.
   c. <u>Synonyms</u>: ***Bayesian Network = Bayes Net = Belief Network = Graphical Model***.
   d. Note: the (more common) directed variant of Bayesian networks must be represented by ***acyclic*** graphs.
3. Sampling:
   a. Distributions are used for 1. Get the probability of a value; 2. Generate random values.
   b. Generating random values allows simulation of a complex process.
   c. This simulation ("***sampling***") allows empirical inference.
4. Basic <u>inferencing rules</u>:
   a. ***Marginalization***: $P(x) = \sum_y P(x \,\&\, y)$
   b. ***Chain rule***: $P(x \,\&\, y) = P(x)P(y|x)$ (corresponding to the Bayes net x→y)
   c. ***Bayes rule***: $P(y|x) =$…
5. ***Naïve Bayes*** – explained through the spam example as in the AI course of Thrun.

a. The Bayes net representing the naïve Bayes model is very simple: one root node (representing the variable we wish to predict) branching directly to multiple leaves (which represent the observed independent variables).
b. Advantages:
    i. Computationally cheap
    ii. Few parameters
    iii. Parameters can be easily estimated by counting in empirical data
    iv. It allows classification of the root node using Bayesian inference
    v. It's empirically successful
        1. In spite of the strong independency assumption!
        2. One can see simple examples where the independency is wrong but yet the naïve Bayes clearly functions well (e.g. if there are n independent variables, and each one is duplicated, then we have 2n variables which are not independent, yet the Bayesian inference would still be correct).
c. ***Smoothing*** = estimate parameters with positively-initialized counters = **prevent overfitting**. That represents a **preference bias** saying that everything is possible with at least a tiny probability.

Final Project

The classic predict-Boston-housing-prices:

https://www.udacity.com/wiki/ud675/project?_ga=1.66102574.1409499416.1454510624