1 Decision Trees

Def: **Decision tree** is method for approximating discrete-valued target functions, the learned function is a decision tree (can be represented as a series of if then rules)

Goal is to figure out what questions to ask, in what order, and what to do with the answers

Appropriate problems for decision tree learning

- instances represented with attribute-value pairs
- target function has discrete output values
- disjunctive descriptions may be required
- training data may contain errors
- training data may contain missing attribute values

Def: task in which you have to classify example into discrete set of possible categories is a **classification problem**

Basic d-tree learning algorithm: ID3 The ID3 algorithm infers decision trees by growing from the root down down, greedily selecting the next best attribute for new decision branch added to the tree.

1. each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples
2. best attribute is selected and used as the test at the root node of the tree
3. descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node
4. repeat using the training examples associated with each descendant node to select the best attribute to test at that point in the tree
5. forms a greedy search for an acceptable decision tree in which the algorithm never backtracks to reconsider earlier choices

But how to choose between attributes?

Def: **Information gain** is a statistical property that measures how well a given attribute seperates training examples according to target classification, ID3 uses information gain measure to select among the candidate attributes at each step while growing the tree

Def: **Entropy** characterizes impurity of a collection of examples. Given collection S where each $s \in S$ can take on c possible values, entropy is

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

where p_i is the proportion of S belonging to class i. For c different classes, max entropy is $\log_2 c$.

Information gain measures expected reduction in entropy, as follows, given set of examples S, and attribute A, the *information gain* of attribute A is

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where $Values(A)$ is the set of all possible values of attribute A, S_v is the subset of S for which attribute A has value v

ID3 algorithm terminates when either (1) every attribute has been included in current path on tree or (2) the training examples associated with the leaf node all have the same target value.

ID3 searches a space of hypotheses for one that fits the training examples. The hypotheses space is the space of all decision trees, because every finite discrete-valued function can be classified with decision tree, hypotheses space will always contain the target function. ID3 doesn't backtrack, so might converge to a locally optimal solution that is not globally optimal (extension: post-pruning). ID3 uses all training examples at each step to make decision, so its less sensitive to errors in training data.

Approximate inductive bias for ID3: shorter trees preferred over larger trees, higher information gain closer to the root - a tree is only grown to be just as large as is needed to classify

Issues with decision trees

- avoid overfitting data

  Def: given a hypothesis space H, hypothesis $h \in H$ **overfits** the data if $\exists$ hypothesis $h' \in H$ st h has smaller error than h' on training examples, but h' has smaller error on entire distribution of instances.

  to prevent: can stop growing tree earlier, overfit then post prune your tree, use training set and validation set, reduced error pruning (only prune if helps reduce error), rule post pruning, use attribute selection other than information gain

- continuous valued attributes - pick a threshold that gives best information gain

2 Naive Bayes

Naive bayes applies to learning tasks where each instance x is described by tuple of attribute values, and the target function $f(x)$ takes on values from a finite set V. Set of training examples is given as tuples of attributes $(a_1, \ldots, a_n)$, and classifier predicts a value v_{NB}. This is a generative model.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are *conditionally independent* given the target value - this is what makes it "naive".

Model takes $O(CD)$ parameters, where C is number of classes (e.g. type of document) and D is the number of features (e.g. total number of words in a document)

Classifier is

$$v_{NB} = argmax_{v_j \in V} P(v_j) \prod_i P(a_i|v_j)$$

Naive bayes involves learning step where you estimate $P(v_j)$ and $P(a_i|v_j)$, based on frequency in training data, this makes the learned hypothesis, and then you use this to classify incoming examples. There is no explicity search through the hypothesis space.

Calculate probability $P(c|v_j = 0)$ by taking

$$\frac{n_c}{n}$$

Where n is the number of examples where $v_j = 0$ holds, and n_c is number of those n examples where c holds. You run into a problem if there is a zero probability attribute or value - it zero's out your classification. Fix this by instead calculating probabilities as an *m-estimate of probability:*

$$\frac{n_c + mp}{n + m}$$

n_c and n are defined the same as above, and p is the prior estimate of the probability being determined, and m is a constant known as equivalent sample size. Typically, if attribute has k possible values, set p to be $\frac{1}{k}$ (e.g. if attribute is binary, $p = \frac{1}{2}$).

If we are classifying text documents (bag-of-words model), the m-estimate of $P(w_k|v_j)$ is

$$\frac{n_k + mp}{n + m} = \frac{n_k + 1}{n + |\text{Vocabulary}|}$$

$m = |\text{Vocabulary}|$ is the number of distinct words in the document, n is the total number of word positions in training examples who's value is v_j, n_k is the number of times word w_k is found in the n word positions ($p = 1/|\text{Vocabulary}|$)

Psuedo-code for text classification is on page 183 of Mitchell.

3 Perceptron

Perceptron is process of finding a good linear decision boundary for given data. Follows the neural model of learning.

Perceptron is *online* (only looks at one example at once) and *error driven* (only updates parameters if it mis-classifies something).

Perceptron Algorithm

`Start` Initialize $\bar{w}^*$

- For $iter = i, \ldots, ITER$
  - For $i = 1, \ldots, n$
    if $\text{sign}(\bar{X}^*(i) \cdot \bar{w}^*) \neq f(\bar{X}(i))$
    * $\bar{w}^*_{new} = \bar{w}^*_{old} + f(\bar{X}(i)) \cdot \bar{X}^*(i)$

`Output` $\bar{w}^*$

Can add a bias term to shift classification to be more positive or more negative.

The decision boundary for a perceptron in a D-dimensional space is always a (D-1)-dimensional hyperplane.

Heuristic for interpreting perceptron weights: sort all weights from largest (positive) to largest (negative) and take the top ten and bottom ten. Top ten are most sensitive for positive classification, bottom ten are most sensitive for negative classification. Works best with binary values, scale features ahead of time.

Perceptron has *converged* when it can make a pass through all the data without making any updates (every example classified correctly). Perceptron will converge when data is linearly seperable, will not otherwise.

Def: If you are given a data set and hyperplane that classifies it, the **margin** is the distance between the hyperplane and the nearest point. Larger margin $\rightarrow$ easier to classify. Given a data set D, weight vector w, bias b margin is

$$margin(D, w, b) = \begin{cases} \min_{(x,y)\in D}\{y(w \cdot x + b)\} & \text{if } w \text{ seperates } D \\ -\infty & \text{else} \end{cases}$$

Def: **Margin of a data set** is the largest attainable margin on dataset D

$$margin(D) = \gamma = \sup_{w,b}\{margin(D, w, b)\}$$

Perceptron convergence Theorem: Suppose the perceptron algorithm is run on a linearly seperable data set D with margin $\gamma > 0$. Assume that $||x|| \leq 1 \ \forall \ x \in D$. Then the algorithm will converge after at most $\frac{1}{\gamma^2}$ updates.

Problem with perceptron is that it counts later points more than initial ones, a wrong classification on the 10,000th step will mess up the weight vector for the first 9,999 steps. Fix this with **voting**: weights that survive in the vector a long time get more say than a newer weight. This is known as the **voted perceptron**

$$\hat{y} = \text{sign}\Big(\sum_{k=1}^{K} c^k \text{sign}(w^k \cdot \hat{x} + b^k)\Big)$$

it is theoretically very accurate, but completely impractical to store all of the voting powers for all of the weights. More practical is **averaged perceptron**

$$\hat{y} = \text{sign}\Big(\Big(\sum_{k=1}^{K} c^k w^k\Big) \cdot \hat{x} + \sum_{k=1}^{K} c^k b^k\Big)$$

Averaged perceptron is generally better than perceptron, but does not have early stopping and will lead to overfitting.

Perceptron Limitations

- decision boundaries can only be linear, XOR problem
  Fix this with feature combinations, feature mappings, combine multiple perceptrons in a neural network, use kernels

4 Regression

4.1 Linear Regression Models and Least Squares

We have input vector $X^T = (X_1, X_2, \ldots, X_p)$ and want to predict real valued output Y. Linear regression model has the form

$$f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j$$

Typically we have a set of training data $\{(x_1, y_1), \ldots, (x_N, y_n)\}$ from which we estimate parameters β. Each x_i is vector $(x_{i1}, x_{i2}, \ldots, x_{ip})^T$ of feature measurements for the ith case. Most popular estimation method is **least squares**, where you pick $\beta = (\beta_0, \beta_1, \ldots, \beta_p)^T$ to minimize the residual sum of squares RSS

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - f(x_i))^2 = \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_i)^2$$

We minimize the $RSS(\beta)$ with the following value

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

Predicted values for input vector x_0 are are given by $\hat{f}(x_o) = (1 : x_0)^T \hat{\beta}$, $\hat{y}_i = \hat{f}(x_i)$.

Gauss-Markov Theorem: the least squares estimates of the parameters β have the smallest variance among all linear unbiased estimates. *Note* restricting to unbiased estimators is not a good idea.

Logistic regression classifier is a *linear model*.

4.2 Shrinkage Methods

By retaining a subset of the predictors and discarding the rest, subset selection produces a model that is interpretable and has possibly lower prediction error than the full model. However, because that is a discrete

process, produces high variance and doesn't generalize well. Shrinkage methods are more continuous, thus don't suffer as much from the high variance.

Ridge Regression shrinks regression coefficients by imposing a penalty on their size.

$$\hat{\beta}^{\text{ridge}} = \text{argmin}_\beta \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 \leq t$$

Lasso is a shrinkage method like the ridge, but with subtle differences

$$\hat{\beta}^{\text{lasso}} = \text{argmin}_\beta \sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

$$\text{subject to } \sum_{j=1}^{p} |\beta_j| \leq t$$

4.3 General regression

Overfitting occurs when a model captures idiosyncracies of the data rather than generalizing. Too many parameters relative to the amount of training data.

How to detect/prevent overfitting:

- use more data
- evaluate on parameter tuning set
- **regularization**
- take a bayesian approach

Regularization introduce a penalty term for the size of the weights

Regularization approaches

- L-2 (closed form in polynomial time)

$$E(w) = \frac{1}{2}\sum_{n=0}^{N-1}(t_n - y(x_n, w))^2 + \frac{\lambda}{2}||w||^2$$

- L-1 (can be approximated in polynomial time)

$$E(w) = \frac{1}{2}\sum_{n=0}^{N-1}(t_n - y(x_n, w))^2 + \lambda|w|_1$$

- L-0 (NP complete optimization)

$$E(w) = \frac{1}{2}\sum_{n=0}^{N-1}(t_n - y(x_n, w))^2 + \lambda\sum_{n=0}^{N-1}\delta(w_n \neq 0)$$

  L-0 norm represents the optimal subset of features needed by a regression model.

Curse of dimensionality: Increasing dimensionality of the feature space exponentially increases the data needs. (dim(Feature space) = number of features)

5 SVM

5.1 Maximum Margin Classifiers

$$y(x) = w^T\phi(X) + b$$

Training data set comprises N input vectors $x_1, \ldots, x_N$, with target values $t_1, \ldots, t_N$ where $t_i \in \{-1, 1\}$ and new data points x are classified according to the sign of $y(x)$

Kernel function takes two vectors and computes their dot product in a higher dimension, say ϕ is mapping to higher dimension, kernel then is

$$k(x, x') = \phi(x)^T\phi(x')$$

Kernel must always be positive definite (or semi-definite) to ensure corresponding optimization problem is well defined. New classification with kernel

$$y(x) = \sum_{i=1}^{N} a_n t_n k(x, x_n) + b$$

A constrained optimization of this form will satisfy the KKT conditions

$$a_n \geq 0$$

$$t_n y(x_n) - 1 \geq 0$$

$$a_n\{t_n y(x_n) - 1\} = 0$$

Which means that for every data point, either $a_n = 0$ or $t_n y(x_n) = 1$. The data points for which $a_n \neq 0$ are called **support vectors**, and correspond to points that lie on the maximum margin hyperplanes in the feature space. Once model is trained, *most of the data points can be discarded*, only need to support vectors to classify.

Data set may not be linearly separable in the two-dimensional data space, it will be linearly separable in the nonlinear feature space defined implicitly by the nonlinear kernel function. Thus the training data points can be perfectly separated in the original data space.

Slack variables ($\xi_n \geq 0$) introduced to allow data to be on "wrong side" of classifier, but give (non-infinite) penalty to these incorrect classifications, penalty increases with distance from boundary. Classification constraints replaced with

$$t_n y(x_n) \geq 1 - \xi_n \quad n = 1, \ldots, N$$

This helps with overlapping class distributions, but still is sensitive to outliers.

Andrew Ng Notation: features are x, labels are $y \in \{-1, 1\}$, classifier is

$$h_{w,b}(x) = g(w^T x + b)$$

$g(z) = 1 if z \geq 0$, and $g(z) = -1$ otherwise.

Def: Given a training example $(x^{(i)}, y^{(i)})$, the **functional margin** of (w, b) wrt the training example is

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b)$$

A large functional margin represents a confident and correct prediction. Prediction only ever depends on the *sign* of the vector $w^T x + b$.

Given a training set $S = \{(x^{(i)}, y^{(i)} | i = 1 \ldots, m\}$ *define* the functional margin of (w, b) wrt S to be the smallest of the functional margins of the individual training examples

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}$$

Note w is orthogonal (at 90^o) to the seperating hyperplane (so the dot product will be zero).

Def: **Geometric margin** of (w, b) wrt training example $(x^{(i)}, y^{(i)})$ is

$$\gamma^{(i)} = y^{(i)}\Big((\frac{w}{||w||})^T x^{(i)} + \frac{b}{||w||}\Big)$$

The geometric margin is invariant to the scaling of parameters.

Given a training set $S = \{(x^{(i)}, y^{(i)} | i = 1 \ldots, m\}$ *define* the geometric margin of (w, b) wrt S to be the smallest of the geometric margins of the individual training examples

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}$$

Finding an **optimal margin classifier** is an optimization that can be solved with *quadratic programming.*

5.2 Langrange Duality

Consider problem of the following form:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{st} \quad & h_i(w) = 0,\ i = 1, \ldots, l \end{aligned}$$

Def: **Lagrangian**

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

The β_is are **lagrange multipliers**, find and set $\mathcal{L}$'s derivatives to zero

$$\frac{d\mathcal{L}}{dw_i} = 0; \quad \frac{d\mathcal{L}}{d\beta_i} = 0$$

and solve for w and β.

Consider the following, a **primal** optimization problem:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{st} \quad & g_i(w) \leq 0,\ i = 1, \ldots, k \\ & h_i(w) = 0,\ i = 1, \ldots, l \end{aligned}$$

To solve, begin by defining **generalized Lagrangian**

Def: **generalized Lagrangian**

$$\mathcal{L}(w, \alpha, \beta) = f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

The *lagrange multipliers* here are the α_i and β_i. Consider the quantity (the *primal*)

$$\theta_{\mathcal{P}}(w) = \max_{\alpha,\beta:\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta)$$

The optimal value of this (primal) objective is

$$p^* = \min_w \theta_{\mathcal{P}}(w)$$

Now if we look at a different problem (the *dual*)

$$\theta_{\mathcal{D}}(\alpha, \beta) = \min_{w} \mathcal{L}(w, \alpha, \beta)$$

The optimal value of this (dual) objective is

$$d^* = \max_{\alpha,\beta:\alpha_i \geq 0} \theta_{\mathcal{D}}(w)$$

It can be shown that

$$d^* = \max_{\alpha,\beta:\alpha_i \geq 0} \min_{w} \mathcal{L}(w, \alpha, \beta) \leq \min_{w} \max_{\alpha,\beta:\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta) = p^*$$

(this follows from the "max min" of a function always being less than or equal to the "min max"), and under certain conditions

$$d^* = p^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$$

The w^*, α^*, β^* must satisfy the **KKT conditions**

$$\begin{aligned}
\frac{d}{dw_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, n \\
\frac{d}{d\beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, l \\
\alpha_i^* g_i(w^*) &= 0, \quad i = 1, \ldots, k \\
g_i(w^*) &\leq 0, \quad i = 1, \ldots, k \\
\alpha^* &\geq 0, \quad i = 1, \ldots, k
\end{aligned}$$

and if they do, they are the solution to the primal and dual problems.

5.3 Optimal margin classifier

Use Lagrange and solve the dual to get

$$b^* = -\frac{\max_{i:y^{(i)}=-1}(w^*)^T x^{(i)} + \min_{i:y^{(i)}=1}(w^*)^T x^{(i)}}{2}$$

And know α_is will be support vectors and see the following holds

$$w^T x + b = \sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b$$

Resulting *support vector machines* will be able to classify well in high dimensional spaces.

6 Kernel trick

Def: **Feature mapping** ϕ maps from attributes to features

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$

(can be to any "features", these are an example)

Def: Given a feature mapping ϕ, the corresponding **Kernel** is

$$K(x, y) = \phi(x)^T \phi(y)$$

i.e. dot product in higher dimension.

Generally, the kernel $K(x, z) = (x^T z + c)^d$ corresponds to a feature mapping to an $\binom{n+d}{d}$ feature space. No matter the feature space that is mapped to ($O(n^d)$-dimensions), computing the kernel will only take $O(n)$ time, because we never have to store the new higher dimensional vector.

Def: **Gaussian kernel**

$$K(x, z) = \exp\Big(-\frac{||x - z||^2}{2\sigma^2}\Big)$$

Def: Given some finite set of m points $\{x^{(1)}, \ldots, x^{(m)}\}$, let a square $m \times m$ matrix, K, the **Kernel Matrix**, be defined so that its (i, j) entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$.

Mercer's Theorem Let $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, \ldots, x^{(m)}\}$, $(m \leq \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

Specifically, if you have any learning algorithm that you can write in terms of only inner products $\langle x, z\rangle$ between input attribute vectors, then by replacing this with $K(x, z)$ where K is a kernel, you can "magically" allow your algorithm to work efficiently in the high dimensional feature space corresponding to K.

7 Nearest neighbor methods

These are instance-based approaches. One disadvantage of instance-based approaches is that the cost of classifying new instances can be high (all computational complexity is at the classification step, training step just consists of saving the data in some form or another).

k-Nearest neighbor learning

Assume all instances correspond to points in an n-dimensional feature space $\mathbb{R}^n$. Let an arbitrary instance x be described via the feature vector

$$x = \langle a_1(x), \ldots, a_n(x)\rangle$$

where a_r indicates the rth attribute of instance x. **Distance** between two instances x_i and x_j is defined to be $d(x_i, x_j)$ as follows

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

Define the nearest neighbor target function $f : \mathbb{R}^n \to V$, where V is finite set $\{v_1, \ldots, v_s\}$. The value of $\hat{f}(x_q)$ (estimate of $f(x_q)$) returned is the most common value of f among the k training examples nearest to x_q.

Algorithm is as follows (discrete case)

Training algorithm:

- For each training example $\langle x, f(x)\rangle$, add the example to the list `training_examples`

Classification algorithm:

- Given a query instance x_q to be classified:
  - Let $x_1, \ldots, x_k$ be the k instances from `training_examples` that are nearest to x_q

– Return

$$\hat{f}(x_q) \leftarrow \text{argmax}_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))$$

Where $\delta(a, b) = 1$ if $a = b$, 0 else.

For the continuous case, replace

$$\hat{f}(x_q) \leftarrow \text{argmax}_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i)) \text{ with } \hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Distance weighting: weight the vote of each neighbor according to the inverse square of its distance from x_q. This can be accomplished by replacing the final line of the algorithm with

$$\hat{f}(x_q) \leftarrow \text{argmax}_{v \in V} \sum_{i=1}^{k} w_i \delta(v, f(x_i)) \quad \text{where} \quad w_i = \frac{1}{d(x_q, x_i)^2}$$

Similarly for continous

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

If all training examples considered for classification, its a *global* method, if only nearest training examples are considered, its a *local method*.

k-nearest neighbors is effective given a large training set and is robust to noisy training data.

Nearest neighbor methods are especially subject to the *curse of dimensionality*: in this case, when the distance between instances (that should be classified the same) is dominated by a large number of irrelevant attributes. Can overcome by weighting attributes differently and using cross-validation.

One additional practical issue in applying k-NEAREST NEIGHBOR is efficient memory indexing.

Terminology

- *regression* means approximating a real-valued function
- *residual* is the error $\hat{f}(x) - f(x)$ in approximating a target function

Lazy vs. Eager learning

A lazy learner has the option of (implicitly) representing the target function by a combination of many local approximations, whereas an eager learner must commit at training time to a single global approximation. The distinction between eager and lazy learning is thus related to the distinction between global and local approximations to the target function.

Instance-based learning methods delay processing of training examples until they must label a new query instance. They need not form an explicit hypothesis of the entire target function over the entire instance space, independent of the query instance. Instead, they may form a different local approximation to the target function for each query instance.

Advantages include the ability to model complex target functions by a collection of less complex local approximations and the fact that information present in the training examples is never lost.

Main practical difficulty is complexity/efficiency of classifying new instances.

8 Bagging, boosting

8.1 Bagging

Bagging ("bootstrap aggregating") predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. If perturbing the learning set can cause significant changes in the predictor constructed, then bagging can improve accuracy.

Bagging unstable (high variance) classifiers usually improves them. Bagging stable classifiers is not a good idea.

Say you have training set T with n examples.

Def: **Bootstrap sampling** is when you create a T' by sampling the n examples from T with replacement

Bagging procedure:

- create $T_1, \ldots, T_r$ (r bootstrap samples)
- train a classifier (or regressor) on each T_i seperately
- output majority vote (classification) or average (regression) of the outputs

Averaging reduces variance, but depends on the model (not all are independent). Bagging does not change bias of underlying model.

8.2 Boosting

Boosting approach

- select small subset of examples
- derive rule of thumb
- examine 2nd small subset of examples
- derive 2nd rule of thumb
- repeat T times

More formally

- given training set $(x_1, y_1), \ldots, (x_m, y_m)$
- $y_i \in \{-1, +1\}$ correct label of instance $x_i \in X$
- for $t = 1, \ldots, T$:
  - construct distribution D_t on $\{1, \ldots, m\}$
  - find *weak hypothesis* ("rule of thumb") $h_t : X \to \{-1, +1\}$ with small error ϵ_t on D_t: $\epsilon_t = Pr_{D_t}(h_t(x_i) \neq y_i)$
- output final hypothesis H_{final}

Boosting = general method of converting rough "rules of thumb" into highly accurate prediction rule.

Adaboost

- construct D_t as follows:
  - $D_1(i) = 1/m$

– given D_t and h_t:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if } y_i \neq h_t(x_i) \end{cases}$$

$$= \frac{D_t(i)}{Z_t} \cdot \exp(-\alpha_t y_i h_t(x_i))$$

where Z_t is normalization constant and $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) > 0$

- final hypothesis:

$$H_{final}(x) = \text{sign}\Big(\sum_t \alpha_t h_t(x)\Big)$$

Adaboost exhibits strong practical advantages over other boosting schemes.

Theorem

- run adaboost
- let $\epsilon_t = 1/2 - \gamma_t$
- then

$$\text{training error}(H_{final}) \leq \exp\big(-2\sum_t \gamma_t^2\big)$$

- so if $\forall\ t$: $\gamma_t \geq \gamma > 0$, then

$$\text{training error}(H_{final}) \leq \exp\big(-2\gamma_t^2 T\big)$$

- this is adaptive: don't need to know γ or T apriori, and exploit $\gamma_t >> \gamma$

Another way to give theorem: training error$(H_{final}) \leq \prod_t Z_t$

NEED TO GO OVER NOTES FROM CLASS TO FINISH THIS SECTION

9 Neural Networks

Biological motivation: complex webs of neurons, similar to how the brain works

Nueron takes a bunch of inputs and outputs a real number.

The entire field of neural computation is, in a way, attempting to understand how the brain works.

The **Artificial Neural Network** (ANN) is a closely connected set of simple units. It is a "general purpose" neural network, and is robust. Limitations include that it is slow learning, and requires a huge amount of data to learn.

Neural Network:

ϕ is decision boundary for a node: $\phi(\bar{w}^* \cdot \bar{x}^*)$

You can have perceptrons splitting data at each node, or sigmoid function:

$$\sigma(x) = \frac{1}{1+e^x}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) \rightarrow \sigma' = \sigma(1-\sigma)$$

- I - set of input nodes
- H - set of hidden nodes

- K - set of output nodes
- w_{ij} - weight of edge connecting node i to j
- *Def:* net_i to be total input to a node

$$out_i = \phi(net_i)$$

Consider one output (O), look at what weight should be

$$(\bar{x}, f(\bar{x})) \rightarrow (\bar{x}, f_1, f_2, \ldots, f_k)$$

$$\text{Error} = \frac{1}{2}\sum_{k \in K}(O_k - f_k)^2$$

Minimize error by taking partial derivatives to get

$$\frac{dE}{dw_{hk}} = \delta_k O_h$$

Thus weight should be

$$w^{l+1} = w^l - \eta \delta_k O_h$$

δ defined as

$$\delta_h = Out_h(1 - Out_h) \cdot \sum_{k \in K} \delta_k w_{hk}$$

Limitations:

- need substantial training examples
- slow learning convergence rates
- poor minima, no convergence guarantees, local minima at best

Tips for running: use multiple networks with different weights, have a *momentum term*, defined as follows

$$\Delta w^{t+1} = -\eta \frac{dE}{dw} + \alpha \cdot \Delta w^t$$

Momentum term tries to get network to converge faster.

Expressivness of nueral networks:

- underlying partition of feature space is a function, we are trying to learn this
- every bounded continuous function can be approximated to arbitrarily small error with with 1 hidden layer
- can learn very complicated functions

Regularization

How do we determine the number of hidden layers? More can lead to overfitting and less can lead to underfitting. Via *weight regularization*

$$\tilde{E} = E(w) + \lambda ||w||_2^2$$

Convolutional NN

Invariance property: Ofter we desire the output not to change under certain changes to the input. We want to be able to use "very raw data" but still get the same output

Solution: Use a convolutional layer in the neural network. Convolution of f and g is

$$f * g = \int g(t - \tau) f(\tau) d\tau$$

10 Hidden Markov Models, Viterbi algorithm

Probabilistic models for sequences of observations, $X_1, \ldots, X_T$, of arbitrary length T.

Markov Chain/Model

$$p(X_{1:T}) = p(X_1) \prod_{t=2}^{T} p(X_t|X_{t-1})$$

Def: when $X_t \in \{1, \ldots, K\}$ the conditional distribution $p(X_t|X_{t-1})$ can be written as a $K \times K$ matrix, known as **transistion matrix** A, where $A_{ij} = p(X_t = j|X_{t-1} = i)$ is the probability of going from state i to state j. Each row of the matrix sums to one, $\sum_j A_{ij} = 1$, so this is called a *stochastic matrix*.

We can simulate multiple steps of a markov chain by taking the transistion matrix to some power equal to the number of steps you want to simulate.

Def: long term distribution over states is known as the **stationary distribution** of the markov chain

Theorem Every irreducible (singly connected), aperiodic finite state Markov chain has a limiting distribution, which is equal to π, its unique stationary distribution.

Theorem Every irreducible (singly connected), ergodic Markov chain has a limiting distribution, which is equal to π, its unique stationary distribution.

Hidden Markov Models

A hidden markov model consists of a discrete-time discrete-state markov chain with hidden states $z_t \in \{1, \ldots, K\}$, plus an *observation* model $p(x_t|z_t)$. Corresponding joint distribution has the form

$$p(z_{1:T}, x_{1:T}) = \left[p(z_1) \prod_{t=2}^{T} p(z_t|z_{t-1}) \right] \left[\prod_{t=1}^{T} p(x_t|z_t) \right]$$

If observations are discrete, its common for them to be represented as a observation matrix:

$$p(x_t = l|z_t = k, \theta) = B(k, l)$$

If observations are continuous, its common for them to be represented as a conditional gaussian:

$$p(x_t|z_t = k, \theta) = \mathcal{N}(x_t|\mu_k, \Sigma_k)$$

Viterbi Algorithm

Given observations and a HMM, wish to find the maximum probability state path. We find this with the **Viterbi algorithm**. Not sure if we need to know actual algorithm - page 8 in Rabiner tutorial.

11 Dimensionality reduction: PCA

Principle components analysis: idea is given data in a d-dimensional space, project it into a lower dimensional space, while perserving as much information as possible. Choose projection that minimizes squared error in reconstructing original data.

Assume data is a set of d N-dimensional vectors $x = ((x_1)^T, \ldots, (x_d)^T)$. We can always express the kth vector as $x_k = \sum_{i=1}^{rank(x^n)} z_i^k u_i$.

PCA problem: given $M < d$, find $(u_1, \ldots, u_M)$ that minimize

$$E_M = \sum_{k=1}^{d} ||x_k - \hat{x}_k||_2^2$$

where $\hat{x} = \bar{x} + \sum_{i=1}^{M} z_i^k u_i$ and $\bar{x}$ is the mean: $\bar{x} = \frac{1}{d}\sum_{i=1}^{d} x_i$

Eigenvectors

Matrix A has eigenvector u with eigenvalue λ if

$$Au = \lambda u$$

For symmetric A (normalized) eigenvectors:

- are orthogonal
- have real eigenvalues
- form an orthonormal basis for A

Projection

Projecting an orthonormal basis is trivial.

Suppose U is our basis (formed by first k eigenvectors), and suppose we want to project a new x

$$w = (U^T U)^{-1} U^T x = U^T x$$

Back to PCA...

Note we get zero error is $M = d$ in PCA.

We want to minimize $E_M = \sum_{i=M+1}^{d} u_i^T \Sigma u_i$

$$\Sigma u_i = \lambda_i u_i$$

Where λ_i is eigenvalue and u_i is eigenvector, doing math gives

$$E_M = \sum_{i=M+1}^{d} \lambda_i$$

PCA Algorithm:

1. $X \leftarrow$ create $N \times d$ data matrix
2. $A \leftarrow$ subtract mean $\bar{x}$ from each column in X
3. $\Sigma \leftarrow$ covariance matrix of A
4. Find eigenvalues and eigenvectors of Σ
5. Principle components $\leftarrow M$ eigenvectors with the largest eigenvalues

PCA Limitations:

- requires carefully controlled data, no missing entries
- completely knowledge free method

PCA Conclusions:

- PCA finds orthonormal basis for data
- sorts dimensions in order of importance
- discards low significance dimensions to get compact description, ignore noise, and hopefully classify better
- not magic: doesn't know class labels, only works with linear variations

Principle component directions for different eigenvalues are orthogonal to each other.

For data to be centered: if you add up each vector in the data it should equal the zero vector.

12 K-means, EM algorithm

12.1 K-means clustering

Suppose we have a dataset $\{x_1, \ldots, x_n\}$ consisting of n observations of a random D dimensional variable x. Goal is to partition the data into K clusters. We want to find a set of D dimensional vectors μ_k, $k = 1, \ldots, K$, where each μ_k is the center of the kth cluster.

For each data point x_n, introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \ldots, K$ describing which of the K clusters the data point x_n is assigned to, so that if data point x_n is assigned to cluster k then $r_{nk} = 1$, and $r_{nj} = 0$ for $j \neq k$. This is known as the $1 - of - K$ coding scheme.

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j ||x_n - \mu_j||^2 \\ 0 & \text{else} \end{cases}$$

Def: **Distortion measure**

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2$$

Goal is to find vectors $\{r_{nk}\}$ and $\{\mu_k\}$ to minimize J. Solving gives

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

Sequential updating (online algorithm): for each data point x_n, update nearest prototype μ_k via

$$\mu_k^{new} = \mu_k^{old} + \eta_n (x_n - \mu_k^{old})$$

Where η_n is the learning rate parameter.

12.2 Mixtures

Def: **Gaussian Mixture Distribution**

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

Introduce a K-dimensional binary random variable z having a $1-of-K$ representation in which a particular element $z_k = 1$ and all other elements are equal to 0. The values of z_k therefore satisfy $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$. The marginal distribution over z is specified in terms of *mixing coefficients* π_k st

$$p(z_k = 1) = \pi_k$$

Expectation maximization for Gaussian mixtures Given a Gaussian mixture model, goal is to maximize likelihood function wrt the parameters

1. Initialize the means μ_k, covariances Σ_k, and mixing coefficients π_k, and evaluate the initial value of the log-likelihood
2. **E step** Evaluate the responsibilities using the current parameters

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)}$$

3. **M step** Re-estimate parameters using the current responsibilities

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k^{new})(x_n - \mu_k^{new})^T$$

$$\pi_k^{new} = \frac{N_k}{N}$$

where

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk})$$

4. Evaluate the log-likelihood

$$\ln p(X|\mu, \Sigma, \pi) = \sum_{n=1}^{N} \ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \right\}$$

and check for convergence of either the parameters or the log-likelihood. If the convergence criterion is not satisfied, return to step 2.

General EM Algorithm

Given a joint distribution $p(X, Z|\Theta)$ over observed variables X and latent variables Z, governed by parameters Θ, the goal is to maximize the likelihood function $p(X|\Theta)$ wrt Θ.

1. Choose an initial setting for the parameters Θ^{old}.
2. **E step** Evaluate $p(Z|X, \Theta^{old})$
3. **M step** Evaluate Θ^{new} given by

$$\Theta^{new} = \arg\min_{\Theta} \mathcal{Q}(\Theta, \Theta^{old})$$

where

$$\mathcal{Q}(\Theta, \Theta^{old}) = \sum_{Z} p(Z|X, \Theta^{old}) \ln p(X, Z|\Theta^{old})$$

4. check for convergence of either log likelihood or parameters, if convergence criterion not satisfied, let $\Theta^{old} \leftarrow \Theta^{new}$ and repeat from step 2.

13 Learning Theory

Def: **Sample complexity** How many training examples are needed for a learner to converge (with high probability) to a successful hypothesis?

Def: **Computational complexity** How much computational effort is needed for a learner to converge (with high probability) to a successful hypothesis?

Def: **Mistake bound** How many training examples will the learner misclassify before converging to a successful hypothesis?

13.1 True error rate, training error rate

Def: **true error** ($error_{\mathcal{D}}(h)$) of hypothesis h wrt target concept c and distribution $\mathcal{D}$ is the probability that h will missclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) = P_{x \in \mathcal{D}}(c(x) \neq h(x))$$

Def: **training error** is fraction of training examples misclassified by h.

Def: Consider a concept class C defined over a set of instances X of length n and a learner L using hypothesis space H. C is **PAC Learnable** by L using H if $\forall\ c \in C$, distributions $\mathcal{D}$ over X, ϵ st $0 < \epsilon < 1/2$, and δ st $0 < \delta < 1/2$, learner L will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ st $error_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $\frac{1}{\epsilon}, \frac{1}{\delta}, n$ and $size(c)$.

13.2 VC dimension, Shattering

Def: A set of instances S is **shattered** by hypothesis space H iff for every dichotomy of S $\exists$ some hypothesis H consistent with this dichotomy.

Def: **Vapnik-Chervonenkis Dimension** $VC(H)$, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H. If arbitrarily finite sets of X can be shattered by H, then $VC(H) = \infty$.

Note for any finite H, $VC(H) \leq \log_2 |H|$. To see this, suppose $VC(H) = d$, then H will require 2^d distinct hypothesis to shatter d instances. Hence, $2^d \leq |H| \rightarrow VC(H) \leq \log_2 |H|$.

To show that $VC(H) < d$, we must show that no set of size d can be shattered.

Generally, it can be shown that the VC dimension of linear decision surfaces in an r dimensional space (i.e., the VC dimension of a perceptron with r inputs) is $r + 1$.