# Supervised Learning

## 1 Introduction

Supervised learning implies that the data is prelabeled ie already distinguished. The model is responsible for correctly predicting this outcome.

## 2 Introduction to Regression

Supervised learning is the process whereby given examples of inputs and outputs an algorithm is then given a new input and asked to predict its output.

Regression is this process specifically done on continuous inputs and outputs.

Linear regression is the attempt to model a linear relationship between a dependent variable and independent variables. Such that given the dependent variable $y$ and the independent variables $x_1, x_2, \ldots x_n$ that we then generate a function $y = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$

The $\theta$ values are referred to as weights and determine how important the corresponding independent variable is.

Gradient descent is the method by which we handling linear regression, using the sum of squared errors rather than absolute errors since sum-of-squared is differentiable.

The aim of linear regression in this case is to minimize the amount of total error across all of the provided input/output values.

There are various different error functions that can be used in order to find the model with the least error. For gradient descent sum of squared error is the best since its differentiable.

Generally we try to fit the function $f(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$ and by using a higher order function we then have more degrees of freedom. We are limited so that the max order $n$ is only $\mid X \mid$ where $X$ is the dataset of all training points.

Picking too high an order of freedom results in over fitting and the resultant model fits the data

but overcommits to the data.

Given the equation $Xw = Y$ where $X$ is the set of independent variables, $w$ is the vector of constants $c_0, c_1, \ldots c_n$ and $Y$ is the set of dependent variables. We are then able to solve for the constant vector $w$ as follows:

$$X^T Xw = X^T Y$$
$$(X^T X)^{-1} X^T Xw = (X^T X)^{-1} X^T Y$$
$$w = (X^T X)^{-1} X^T Y$$

This is using projections in linear algebra.

Training data has errors that is we model not $f$ but $f + \epsilon$. Errors can result from a number of sources.

Assume that data is independent and identically distributed. This implies that all the data comes from the same source and there are no implicit biases in the data. This is not a fundamental assumption of supervised learning however, it is an assumption of certain algorithms within supervised learning.

We can split the training set into a cross validation set (a stand-in for the actual testing data). Cross validation splits data into $n$ folds. Then test on each fold, training on the rest of the data. You then choose the degree model with the lowest error and average the coefficients together.

Without enough complexity you underfit the data. Too much and you overfit the data.

Vector inputs are possible and generalize to hyperplanes (that is we aren't limited to a small number of axes).

We can use discrete, vector, or scalar data can be input into regression models by using ordinal conversion function. **Note**: A conversion function can be approximated by using another machine learning algorithm.

Boolean vectors can be used to encode complex data.

# 3 More Regressions

Parametric regression represents a model with a number of parameters (ie regression).

K nearest neighbor finds the k neighbors closest to the input value and then averages their output values together using a couple of different metrics.

Kernel regression is similar to KNN and each data point is weighted unlike KNN where each point is simply left as they are.

Instance based methods keep the data and use it to generate a prediction when provided with an unknown label.

The parametric model is biased (as in we already have an idea of the solution) vs an instance method which is unbiased.

Parametric models are space efficient however cannot be updated easily. Training is slow but querying is fast.

Instance based methods are not space efficient, training is fast but querying is slow. Also have the added benefit of fitting patterns that are too complex for parametric models.

# 4 Regressions in sklearn

Continuous supervised learning is regression.

Continuous learning works on data sets with some type of ordering.

Target variable is the dependent variable and input variables are the independent variables.

Linear regression minimizes the sum of the squared errors. The best regression is the model that minimizes the SSE metric.

Ordinary least squares (OLS) (used in sklearn) and gradient descent are the two algorithms used to minimize SSE.

There is a fundamental ambiguity. There can be multiple lines that minimize the $\sum \mid \text{error} \mid$ but only one line minimizes $\sum \text{error}^2$.

Furthermore, using SSE makes the implementation easier as the function is differentiable.

SSE does not account for the number of data points and thus as the data set size grows the SSE grows as well. The SSE metric can not compare fits between two different data sets.

$R^2$ explains how much of my change in the output is explained by my change in the input and is bounded $0 < r^2 < 1.0$. $R^2$ is not generally affected by the number of data points.

Regression requires data that fits its structures well. Data can be transformed before hand as a fix.

Reinforcing the previous notes concerning regression vs classification note that discrete labels and decision boundary indicate supervised classification method. In the case of regression the object is have a continuous output and a line of best fit. Classification uses accuracy as its measure where regression uses sum of squared error.

Multivariate regression is regression on several independent variables reducing to a single output variable.

# 5 Decision Trees

Classification maps a set of independent variables to a set of discrete labels.

Instances are inputs, vectors of independent values of input.

Concepts are represented by functions. A relationship between some input and output. A concept is formally defined in this case as a set of things.

The target concept is the concept from the possible concept space that most effectively describes the inputs in terms of the outputs by some metric.

The hypothesis class is the set of concepts that are possibly useful to use.

A sample is a training set of inputs (independent variables) and labels (outputs).

Candidate is a concept that we evaluate to be an effective target concept.

The testing set is a set of inputs and outputs that is specifically used to evaluate the competency of the candidate concept.

The representation of a decision tree is a n-ary tree. Where each set of children the node represents a decision and the node represents a specific selection against that decision. Decision trees only use data that is relevant, it will ignore any data that is not required to model the data properly.

**Note**: Decision trees start from the root node.

Decision trees attempt to create the most equal splits possible when splitting a decision tree. The deeper questions are predicated on the earlier possibilities.

The algorithm for decision trees can be defined as follows:

1. Pick the best attribute (gives the most even split).
2. Asked the question
3. Follow the answer path
4. Until you get an answer go to 1

A decision tree building algorithm follows all possible path until all possible questions are answered. Whereas the above algorithm only defines how to use a prebuilt decision tree.

A decision that keeps the data as it is, is less useful than any possible decision split that splits the data in any capacity. A decision that doesn't actually split the data is essentially a mapping of the current state of the data to itself and simply wastes cpu time.

Decision trees can be considered a more general class of the behavior of a transistor. It can model most logical operators. A decision tree can be considered a form of logical primitive in this case.

Not all decision trees are commutative but they do exist.

The number of nodes in a decision tree represents how complex it is. The trees for n-or and n-and are linear that is there are $n$ nodes that need to be traversed.

The n-xor tree on the other hand, $2^n - 1$. The n-xor tree is exponential, and is a NP hard problem.

Splitting on a choice we construct a decision table to represent all the possibilities. The resulting space in which a decision tree can exist, of the many possible choices we must consider is $m_{output}^{\prod^{n_{decisions}} |k_n|}$ where $k$ the cardinality of the decision outputs for a given decision.

An algorithm like that is very expressive, meaning that the hypothesis space is very large. However, this means that the algorithm we use to search this space needs to be extremely efficient in order to prevent nearly infinite runtimes.

## 5.1 ID3

The following algorithm is used to efficiently build decision trees.

We begin by looping forever until a solution is found

- $A \leftarrow$ best attribute
- Assign $A$ as decision attribute for node
- For each value of $A$, create a decedent of node
- Sort training examples to leaves
- If examples are perfectly classified stop. Else iterate over the leaves

Information gain is a mathematical representation of the reduction in randomness and is defined $\text{Gain}(S, A) = \text{entropy}(S) - \sum_v \frac{S_v}{S} \text{entropy}(S)$.

Entropy is defined as a measure of impurity in a bunch of examples.

The formula for entropy is defined $-\sum_v p(v) \log_2(p(v))$ where $p(v)$ is the probability of seeing a value. Entropy in machine learning refers to the homogeneity of a dataset

The best attribute is the one that maximizes information gain.

ID3 terminates under two conditions: When there is no relevant data left to split on (ie its perfectly classified) or when there are no more attributes available to divide the data with.

The ID3 algorithm is recursively called against itself unless a terminating condition is met.

Information gain is calculated and then sorted for each remaining attribute during the process of building the decision tree.

ID3 is a greedy algorithm as it has no complex look ahead behavior and only focuses on the immediately next set of nodes.

As the number of examples (the size of the training data) and/or the number of attributes to analyze increases, the likelihood the tree returned by ID3 is suboptimal.

With continuous data bucketing can be used to enable ID3 to use it.

If entries are missing from data there are several ways to fill the gaps.

- You can replace the gap with the most common (in the data) element for that specific attribute.
- You can also assign probabilities to each possibility for the given attribute and then randomly assigning gaps using a pseudo random generator until no gaps remain.

Since ID3 fits until there is no fitting left to do, it can drastically overfit the data. There are two major ways to deal with this:

1. Stop growing the tree before it becomes too large
2. Pruning the tree once it is done

A limit to the growth can be established through a limit on the layers/depth. Cross validation can also be used to help establish the best tree structure.

Pruning however requires testing the fully built tree against the pruned trees and seeing if it the pruned tree performs better on the test data.

Finally we are able to manipulate the formula used for the information gain analysis. If there are many buckets and not enough data we can wind up with sparse separation which is difficult for the information gain formula to evaluate.

The formula called Gain Ratio overcomes the aforementioned difficulty by measuring how evenly and how broadly the data is distributed. It is expressed: $\text{GainRatio}(S, A) = \frac{\text{Gain}(S,A)}{\text{SplitInformation}(S,A)}$ where SplitInformation is defined $\text{SplitInformation}(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$

There are two major types of bias:

1. Restriction bias: Refers to the hypothesis set (anything outside is not considered)
2. Preference bias: What hypotheses in the hypothesis set are preferred

ID3 prefers splits at the top, correct trees and trees that are shorter.

### 5.2 Decision Trees Continued

Use range buckets in order to represent bounded continuous attributes.

Using separate bucketing methods to extract discrete value attributes allows continuous values to be reused in the same child-to-root path for a given decision trees.

Fitting the training data perfectly is a non-starter since there is the possibility of noise in the data.

A complex decision tree conflicts with occam's razor. Using cross validation it is possible to get a more efficient tree by comparing different trees against one another.

Besides hard limits to prevent a tree from growing to large, the information gain function can add weights to cause it to prefer breadth first growth.

Information gain is not easily used on regression. So we need to change the spitting criteria. It is possible to specify a local output function for each leaf.

Another way to measure the decision when to stop is a voting function applied over the leaf nodes.

## 6 More Decision Trees

Linearly separable data is data that can be partitioned by a $y = m_0x_0 + \ldots + m_nx_n + b)$.

Decision trees allow multiple yes/no questions (binary decision tree).

Analogue: Decision tree is a 8bit coast building algorithm.

In sklearn, the default split criterion is the Gini index.

Decision trees are easy to use and easy to grow. Easy to display. Bigger classifiers can be built using decision trees using ensemble methods.

Decision trees are prone to overfitting with lots of data and be careful with parameter tunes. Outliers can drastically affect the growth of the tree if the tune does not modify the minimum split criteria.

## 7 Neural Networks

Via approximation, artificial neural networks are cartoonish in comparison to actual neurons.

An artificial neuron has a set of inputs $X$ with a set of weights $W$ to the threshold of the neuron $\theta$ and an output $y$.

The activation is defined as $\sum_{i=1}^{k} x_i \times w_i$. If the activation is equal to or exceeds the threshold than the neuron outputs.

The artificial neuron is referred to as a perceptron and is considered the most basic form on an artificial neuron.

Perceptrons compute a half plane, since the activation function is linear, the decision boundary that they compute is always linear. can express basic logical constructs (or, and, not) etc with a single perceptron unit.

When using a bounded evaluation structure and an unbounded (continuous) internal representation there is often an infinite number of viable expressions that can represent the bounded evaluation metric. That is AND/OR/NOT/XOR are easy to express with a variety of different perceptron formulations. Note this does not imply that all values are possible to represent all solutions, there are just an infinite number of bounded solutions.

There are two different rules are

- Perceptron (thresholded output)
- Gradient descent/delta (not thresholded)

Defining the perceptron rule we begin with the input labels $X$, the output $y$, the evaluated output $\hat{y}$, the learning rate $N$ and the weights $W$.

When doing calculations using the perceptron we have the following expression: $\hat{y} = \sum_i w_i x_i \geq \theta$ we can modify this function to $\hat{y} = \sum_i w_i x_i \geq \theta > 0$. This enables us to calculate the threshold as a weight, improving calculation times. In order to take advantage of this we add a single column of '1's to the input labels $X$ in order to account for this.

We perform several iteration steps of the perceptron rule function, by which we modify the weights. The weight changing function is defined $w_i = w_i + N(y - (\sum_i w_i x_i \geq 0))X_i$. The inequality in the middle assumes a binary yes/no output structure.

If there is a half-plane that separates the data, then the data is linearly separable and then the perceptron rule can find the answer in a finite number of iterations.

Only run the perceptron rule while there is some error.

An algorithm that is not solely focused on linearly separable data would be gradient descent. Gradient works by defining an error metric over the data set. The error function is defined as $E(x) = \frac{1}{2} \sum_{(x,y)\in D}(y - (\sum_i x_i w_i))^2$. We then simply minimize the error by finding the derivative and then solving for the minimum. We finally wind up the equation: $\sum_{(x,y)\in D}(y - (\sum_i x_i w_i)(-x_i))$ The weight varying function for gradient descent is defined as $\delta w_i = N(y - a)x_i$.

Perceptron rule has a guarantee of finite convergence in the case of linearly separability.

Gradient descent is more robust (data doesn't need to be linearly separable) but as a trade-off only converges to a local optimum.

We use the perceptron rule for cases where the data thresholding function is non-differentiable.

The sigmoid function acts like the perceptron function however it is differentiable. The sigmoid function is defined as $\sigma(a) = \frac{1}{1+e^{-a}}$. The derivative of the $\sigma$ function is written $D\sigma(a) = \sigma(a)(1 - \sigma(a))$.

The sigmoid function is useful in allowing gradient descent to be used universally, where previously the perceptron function itself could only be used.

Neural networks are layers of nodes that compute the sigmoid weights of the previous layers. There are hidden layers (not available to IO) that are used for larger representations. The mapping from input to output using a sigmoid ann is entirely differentiable.

Back propagation arises from the computationally beneficial organization of the chain rule. That is data flows forwards while error flows backwards. Also known as error back propagation.

Back propagation occurs so long as the error function is differentiable.

Sigmoid units are analogous to a perceptron but have many local optima.

There are other advanced methods of optimization (learning) that can be used:

- Momentum: pushes the ball so it doesn't necessarily settle in local optimum
- Higher order derivatives: combinations of weights, hamiltonians etc
- Randomized optimization
- Penalties for complexity, prevents overfitting (less nodes, less layers, large numbers)

Large numbers affect network complexity (increase the complexity and chance of overfitting).

- Boolean functions can be represented by a network of threshold-like units.
- Continuous functions can be represented by a single hidden layer.
- Arbitrary functions can be represented by a double hidden layer, stitched together.

There is a danger of overfitting due to the capability of networks to represent complex data. Cross validation can be used to determine the number of hidden layers, number of nodes in each layer and when to cap the size of weights.

The initial weights for a neural network, typically small random values are chosen (different random values for different runs).

Neural networks prefer simple explanations, occam's razor as well as correct values. Occam's razor: entities should not be multiplied unnecessarily.

# 8 Support Vector Machines

Fits that are too close to the data overfit the data. An svm finds a line that fits the data but is least committed to it.

We consider the generalized linear function $y = w^t x + b$. Where $y$ is the labels, $x$ is the input, $w$ and $b$ are the parameters of the splitting hyperplane.

We want the plane that correctly splits the data while being as far away from it as possible.

The equation for the splitting hyperplane is specifically formulated as $w^t x + b = 0$.

The output labels $y \in -1, 1$, and we can then say the lines tangential to the positive and negative sets can be defined as $w^t x + b = 1$ and $w^t x + b = -1$. When then maximize the distance between these two lines. This distance is defined as $\frac{2}{||w||}$

Goal of support vector machines is to maximize the margin such that the margin $m$ is defined as $\frac{w^T}{||w||}(x_1 - x_2)$. However we need to correctly classify everything which is represented by the margin $m$ being constrained such that $m = \frac{2}{||w||}$.

In order to classify everything correctly we have consider the inequality $y_i(w^T x_i + b) \geq 1$ considering $\forall_i$. That inequality is used because correctly predicted non-members multiple double negatives and invert the sign of the predicted value. However maximizing $\frac{2}{||w||}$ is difficult so instead we minimize $\frac{1}{2}||w||^2$. The form is easier to solve as it is a quadratic programming problem. We then formalize the problem in the quadratic normal form $W(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j x_i^T x_j$. This is done under the constraints that $\alpha_i > 0$ and $\sum_i \alpha_i y_i = 0$ and try to maximize the $W(\alpha)$.

Retrieving the initial values we have $w = \sum_i \alpha_i y_i x_i$. Most $\alpha$ are zero. Most data doesn't matter when considering how to split the data. The data that is used to actually define the support vector machine are considered the support vectors.

The points that are close to the decision boundary are important while the ones that are farther from the decision boundary are not considered important. Its an example of instance based learning that gets rid of the unnecessary points. The dot product can be considered a measure of similarity.

Data sets that are linearly married imply that a support vector machine cannot separate the data properly as any line chosen still results in errors. We use a kernel function $\Phi$. The formula $\Phi$

is used because when applying it in support vector machines and apply the dot product we can transform the data so that we can still apply the SVM.

There is some higher dimensional space that is equivalent to the current one, via the kernel. However that kernel function may need to transform an unbounded number of equations.

The general form of the SVM is $W(\alpha) = \sum \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$ where $K(x_i, x_j)$ is the kernel where we inject domain knowledge into the SVM. A common kernel form is $(x^T y + c)^p$. A good kernel function captures domain knowledge properly. The kernel function measures the similarity of two items in a specific domain.

Kernel functions should explain the Mercer condition: it acts as a well behaved distance function.

Margins are useful in determining how generalization and overfitting happen.

# 9 SVMs in practice

SVMs split data of two classes.

Margin is the distance between the line and the points that is maximized by the SVM.

SVM lines that are too close to the data are more susceptible to noise

An svm prioritizes a correct separation of the data over the maximization of the margin.

SVMs are capable of tolerating small/individual outliers.

The $C$ parameter controls the tradeoff between a smooth decision boundary and classifying the training data points correctly. A higher $C$ implies more training points will be correct. That is we fit tightly to the data.

The $\gamma$ defines how far the influence of a single training example reaches. The higher $\gamma$ the closer the decision boundary fits to the data.

SVM work well on complicated domains.

Doesn't work well on large data sets ($O(n^3)$)

Does not handle lots of noise well, overfits to the noise.

# 10 Instance based learning

Instance based learning remembers the data, its very fast to insert new data and its generally simpler.

Distance between training examples is important. There are multiple metrics to measure distance in a traditional sense, let alone for more abstract classifications. Where distance and similarity are analogues for one another.

## 10.1 K-NN

Given the training data $D = x_i, y_i$, a distance metric (domain knowledge) $d(q, x)$ and the number of neighbors $k$ (also considered domain knowledge). Finally we are also given the query point $q$.

We define the nearest neighbor set $NN$ as the set $i : d(q, x_i)\text{min}k$ . Where we then return the mode/plurality of the $NN$ in the case of classification and in the case of regression we take the mean.

## 10.2 Use of KNN

Ties in classification are handled by a designer metric.

Most of the features of the KNN algorithm are defined by the designer which allows the algorithm to be apparently simple in its structure.

Instance based learning is slower to query then learn while other methods are slow to train and quick to query. Lazy vs eager learners.

The choice of distance metric and k drastically affect the performance of the KNN.

KNN assumes near points are similar to one another, the distance function encodes that assumption. There is always one best distance function. KNN by averaging things together, expects functions to behave smoothly. Dependent on the distance function, most KNN metrics assume most features matter equally, there is no explicit feature preferencing.

KNN like most other algorithms suffers from the curse of dimensionality: as feature/dimensions grow the data required to generalize increases exponentially.

The choice of distance function is integral to the performance of KNN.

Weighting in the distance function can be used to help with the curse of dimensionality.

Setting $k = n$ and using a weighted average we wind up using a specific technique called locally weighted regression (which regressed by averaging the local data points more heavily than

others).

Similarity is just another way of capturing domain knowledge.

Instance based method of using linear regressions which allow us to compose several learning methods together. (The KNN averaging function and distance functions can be used to encode other algorithms).

# 11 Naive Bayes

The decision boundary is a separator of two or more classifiers.

Save about 10% of data for testing.

$p(E_2|E_1)$ is the sensitivity and $p(\neg E_2|\neg E_1)$ having it is the specificity.

Prior probability and test evidence results in the posterior probability. $P(E_1|E_2) = P(E_1)P(E_2|E_1)$ and $P(\neg E_1|E_2) = P(\neg E_1)P(E_2|\neg E_1)$.

We then normalize each item by $P(E_1|E_2) + P(\neg E_1|E_2)$

Naive Bayes is naive because it ignores greater context.

# 12 Bayesian Learning

Basics of machine learning are to learn the most probable hypothesis given some data and some domain knowledge. In this case we equate best to likelihood. This can be represented as $\text{argmax}_{h \in H} P(h|D)$ where $h$ is a particular hypothesis and $D$ refers to a set of data.

Formally speaking Bayes Rule is written $P(h|D) = \frac{P(D|h)P(h)}{P(D)}$

We can derive Bayes' Rule from the standard chain rule in probability: $\Pr(a, b) = \Pr(a|b)\Pr(b) = \Pr(b|a)\Pr(a)$

In Bayes' rule $\Pr(D)$ refers to your prior belief about the data. $\Pr(D|h)$ refers to the data given the hypothesis and refers to the actual running the hypothesis. $\Pr(h)$ refers to out prior belief refers to our belief in a certain hypothesis over any others, this generally refers to domain knowledge. This implies better domain knowledge or higher accuracy $\Pr(h)$ and $\Pr(D|h)$ improve our overall probability.

If a test is not useful, then via Bayesian reasoning it makes sense to alter the population you perform the test on. The algorithm for Bayesian Learning we for each $h \in H$, calculate $\Pr(h|D) =$

$\frac{\Pr(D|h)\Pr(h)}{\Pr(D)}$ and then find the maximal probability in the set of outputs and choose that as out $h$. That is $h = \text{argmax}_h \Pr(h|D)$. Its not necessary to find $P(D)$ since we only need to find the argmax.

The Maximum a posteriori or MAP is defined $\Pr(h|D) \approx \Pr(D|h)\Pr(h)$.

It is often difficult to derive any useful domain knowledge $\Pr(h)$ we then reduce the computed hypothesis to the maximum likelihood hypothesis which is defined as $h_{ml} = \text{argmax}_h \Pr(D|h))$ since we assume all hypotheses are equally likely.

Difficult to use the most likely hypothesis model directly since hypothesis space may be infinite. Maximum likelihood can be used as a gold standard.

Given a set of data $< x_i, d_i >$ as a set of noise free examples of a hypothesis $c$, that $c \in H$ and there is a uniform prior aka an uninformed prior. We then define $\Pr(h) = \frac{1}{|H|}$. We also have $\Pr(D|h) = 1$ s.t. $\forall d_i \in D d_i = c_i)$. And finally we have $\Pr(D) = \sum_{h_i \in H} P(D|h_i)\Pr(h_i)$ assuming that all hypotheses are mutually exclusive. Deriving further we only select the hypotheses that are consistent with the data (that is all points agree) since otherwise their probability is 0. So we get $\Pr(D) = \sum_{h_i \in VS_{H,D}} 1 * \frac{1}{|H|} = \frac{|VS|}{|H|}$ Finally we have $\Pr(h|H) = \frac{1}{|VS|}$ This states that all hypotheses in the version space are all equally as usable, however this depends on the correct hypothesis being in the hypothesis space, noise free data and a uniform prior.

A derivation for Bayesian learning with noise follows: Given a set of inputs and labels $< x_i, d_i >$. The function that we are trying to analyze $f$ is defined such that $d_i = f(x_i) + \epsilon_i$. The error $\epsilon$ is drawn from a normal distribution $N$ with a 0 mean and a variance $\sigma^2$ whose relevance is minimal. Each epsilon is independent and identically distributed.

We begin with $h_{ML} = \text{argmax}\Pr(D|h)$ and then via the chain rule have $\text{argmax}\prod_i \Pr(d_i|h)$. Then since each $\epsilon_i$ is distributed over a normal distribution and is iid, then we can express the function as a product of the likelihood of any specific errors occurring in the form of a Gaussian

$$\text{argmax}\prod_i \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(d_i - f(x_i))^2}{2\sigma^2}}$$

Since we are using argmax we can remove any constant expressions so using log is monotonic then we can drop the exponential and the constant term $\frac{1}{\sqrt{2\pi\sigma^2}}$ can be dropped. We then have $\text{argmax}\sum_i \frac{-(d_i - f(x_i))^2}{2\sigma^2}$. Then minimizing instead of maximizing and throwing out constants we get $\text{argmin}sum_i(d_i - h(x_i))^2$ and this function is represented by the sum of squared errors.

Several machine learning algorithms 'pop' out of the Bayesian derivation, and our assumptions. We actually as a result assume Gaussian noise corrupting all label data but not corrupting all input.

A derivation of the minimum description length follows: Given $h_{MAP}$ such that $h_{MAP} = \text{argmax}\Pr(D|h)\Pr(h)$ we then apply $\log_2$ to get $\text{argmax}\lg\Pr(D|h) + \lg\Pr(h)$ and then multiply by $-1$ to get $\text{argmin}\ [-\lg\Pr(D|h) - \lg\Pr(h)]$. Then finally from information theory we have the optimal code for event $w$ with probability $p$ has a length $-\lg p$. We then have $\text{length}(D|h) + \text{length}(h)$. Reduced we want to minimize error + size. This shows that we desire shorter and correct descriptions. In essence it is a Bayesian

representation of Occam's razor.

Sadly the tradeoff between misclassification error and the size of the hypothesis is built in and can't be entirely overcome.

However, Bayesian learning is focused on finding the best $h$ not the best label $d$ as a result we need to actually calculate $d$ which is done by a weighted vote where for $h \in H$ the weight of the vote is $\Pr(h|D)$. This is represented by the equation

$$d_{MAP} = \text{argmax}_d \sum_h \Pr(d|h) \Pr(h|D)$$

This is known as the Bayes Optimal Classifier. There is no way to do any better on average than the BOC.

# 13 Bayesian Inference

Bayesian networks are good at representing joint probabilities over complex spaces.

A joint distribution is a distribution of combined factors with a total probability of one.

We use conditional probability and independence in order to prevent the exponentiation of the factors every time we add parameters.

Conditional independence is defined as: $X$ is conditionally independent of $Y$ given $Z$ if the probability distribution governing $X$ is independent of the value of $Y$ given the value of $Z$ or formally $\forall X, Y, Z \Pr(X = x|Y = y, Z = z) = \Pr(X = x|Z = z)$ or more compactly $\Pr(X|Y, Z) = \Pr(X|Z)$. The independence of two independent variables means their joint probabilities is the product of their margins. It is not possible to necessarily possible to be as strong as independent probability. However, conditional independence can be used more generally.

Belief networks are also know as Bayes networks or graphical models. The nodes correspond to the variables and edges correspond to dependencies.

The more influences on a node the number of nodes in the probability table to calculate the resultant probability table grows exponentially. That is the cardinality of the probability table $|P(E)| = 2^k$ where $k$ is the number of influential nodes on event $E$. This deals with the indegree of the belief network structure.

Belief network shows relationship between probabilities. Are not able to infer causality between. Merely talking about statistical dependence.

Sampling from a joint distribution is done in topological order, however this depends on the graph must be acyclic.

We depend on Bayes belief networks being acyclic so that we can properly compute joint probability distributions over them.

Cyclic graphs make everything more complicated and can result in complicated predicating and constraints on the conditional probability.

We are able to use the joint distribution to compute any joint subgraph. Given a joint distribution where there exist nodes for conditional evens $a \in A$ we we compute the subgraph via the fact that each event $a$ and its probability table $P(a)$ is written such that if that event $a_i$ is conditionally independent predicated on events $a_j, a_k, \ldots a_n$ these can be considered a subset $A_i$ then we write the probability table for $a_i$ as $\Pr(a_i|A_i)$ and then the joint probability distribution of any subgraph $A_s$ can be expressed as $\prod_i \Pr(a_i|A_i)$ where $\forall a_i, A_i$ that $a_i \in A_s$ and $A_i \subset A_s$

The number of boolean variables required to represent the subgraph is normally $2^{|A_s|}$ but using a joint probability distribution where the elements are conditionally independent we gain the benefit of expressing it with fewer values and the minimal expression can be done via $\sum_i 1 + |A_i|$. If all events are independent than the minimal expression possible is $|A_s|$.

Distributions are useful for determining the probability of a certain value and also generating values. If a distribution represents a process, it allows us to approximately inference and to model that process and to simulate it. We are able to visualize the distribution and to get a feeling for the behavior of the distribution. We use approximation since exact is hard and/or approximation is faster. Solving arbitrarily complicated belief networks is NP hard. (Due to the exponential explosion of complicated conditional independent events).

There are some basic probabilistic inference rules such as marginalization where the formula $\Pr(x| = \sum_y \Pr(x, y))$ and the chain rule which is $\Pr(X, Y) = P(X)P(Y|X)$ and then we have Bayes Rule.

If you know your in a specific context there are features that are common and their are probabilities of those features occurring given the context.

The probability of finding the probability of a root node given a number of attributes. Then we have the formula $\Pr(V|a_1, a_2, \ldots a_n) = \prod_i \frac{\Pr(a_i|V)P(V)}{Z}$ where $Z$ is the normalization factor across all of the attributes. This type of formalization allows for the use of attributes to determine the general class of something. This classification formula is MAP class $=$ argmax $\Pr(V) \prod \Pr(a_i|V)$

Inference is cheap with naive Bayes, there is a linear number of parameters. We can estimate parameters with labeled data (count the number of occurrences in that class over the number of items in that class). Naive Bayes connects inference and classification and Naive Bayes is empirically successful.

Naive Bayes assumes conditionally independent features if you know the class. No inner relationships among the attributes are not considered.

Its OK if the probabilities are wrong so long as the answer is in the right direction ('OK in practice validation').

Ordering is preserved, Naive Bayes believes its answers too much. One unseen attribute spoils the entire product. A way to avoid the full veto is to initialize the values to small probabilities

(this is considered overfitting). By smoothing the data we have an inductive bias that everything is possible.

Naive Bayes allows us to link Bayesian Probability and classification. It handles missing attributes well, it allows inference in any direction.

# 14 Ensemble B&B

Using a number of simple rules can help to determine but collectively cannot represent eh whole corpus of rules required to fully represent a concept or classification.

Ensemble learning combines simple rules into a complex rule. Subsets of the data are used to generate these simple rules. These rules are then combined.

Simplest method for subsets is uniformly and randomly picking data. Simplest method is combining is mean/voting.

Ensemble learning gets the benefits of cross validation, that is we don't fit the training data as closely and generalize better, getting rid of some overfitting.

Choosing random subsets to apply learners to is known as bagging or bootstrap aggregation.

Subsets should be chosen iteratively by creating subsets based on those with the highest error for the current/previous learners and then they are weighted for the final computation.

We define error in boosting as $\Pr_D[h(x) \neq c(x)]$.

A learning algorithm is weak when no matter the distribution, it will do better than chance. Formally speaking this implies $\forall D \Pr_D[\cdot] \leq \frac{1}{2} - \epsilon$. This implies the learner always learns something from the data.

However if the distribution space has a potential weighting situation that prevents some hypothesis from satisfying the definition, then we have no weak learner. Small hypothesis spaces with bad hypotheses make having a weak learner difficult.

## 14.1 Boosting Algorithm

Assume a classification circumstance

Given training data $\{(x_i, y_i)\}$ where $y \in \{-1, 1\}$

For $t = 1$ to $T$ we:

- Construct $D_T$

- Find a weak classifier with $h_t(x)$ with a small error $\epsilon_t = \Pr_{D_t}[h_t(x) = y_i]$

Then output $H_{final}$ The $\epsilon$ is bounded such that $\epsilon < \frac{1}{2}$.

To find the distribution the first distribution $D_1(i) = \frac{1}{n}$ At every time step we have distribution $D_{t+1}(i) = \frac{D_t(i) \cdot e^{-\alpha_t y_u h_t(x_i)}}{Z_t}$ where $\alpha_t = \frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$. This utilizes the fact that the algorithm is working for a classifier (if the predicted and actual label have the same sign the probability decreases depending on the normalization constant.

$Z_t$ stands for whatever normalization constant is required to make the distribution work.

Then we get the weighted average $H_{final} = \text{sign}(\sum_t \alpha_t h_t(x))$.

## 14.2 Boosting Continued

A feature of ensemble methods is that $H \leq \{\sum h \in H\}$ whereby we are referring to the complexity of the hypothesis classes and therefore can be more expressive.

Confidence refers to how strongly you believe in a specific hypothesis.

Variance can be used as a stand-in for confidence (low variance = high confidence).

Using a modification to the weighting formula normalizing it, we can obtain a measure of confidence. That is $H_{final}(x) = \text{sign}(\frac{\sum_t \alpha_t h_t(x)}{\sum_t \alpha_t}$. The closer the value is to the maximal $-1$ or 1 the more confident the ensemble method is in that value.

Essentially as iterative steps are done, error stays the same but confidence increases. Boosting increases the margin between the confidence and larger margins counter overfitting.

Weak learners tend to overfit in the circumstances where the underlying learners overfit. Pink noise also causes boosting to overfit. Pink noise is uniform noise.

White noise is Gaussian noise.

There is no direct definition of 'strong learner'.