A Companion of Questionable Quality to *Machine Learning: An Algorithmic Perspective*

R. Dougherty-Bliss

CSC-490, 2016FA

Contents

Companion Introduction **3**

Introduction **3**
- Complexity of Learning Algorithms . . . 4
- Supervised Learning . . . 4
- Examples of Noise . . . 5

Preliminaries **5**
- Distance Formula . . . 5
- The Curse of Dimensionality . . . 5
- Accuracy Metrics . . . 6
- Training, Testing, and Validation Sets . . . 6
  - K-fold Cross-Validation . . . 6
- Accuracy Metrics . . . 7
- The Receiver Operator Characteristic Curve . . . 7
- Naïve Bayes Classifier . . . 7

Neurons and Neural Networks **8**
- Neurons . . . 8
- Perceptrons . . . 8
  - Introduction . . . 8
  - Perceptron Convergence Theorem . . . 8
  - Convergence Time Estimate . . . 9
  - Perceptron Linear Regression . . . 9

MLP in Practice **9**
- When to Stop? . . . 9
- Probably Approximately Correct Learning . . . 10
- Universal Approximation Theorem . . . 10
- 1-of-N . . . 10
- Compression . . . 10
- Recipe for MLP . . . 10

Dimensionality Reduction **11**
- Linear Discriminant Analysis (LDA) . . . 11
- Principal Components Analysis (PCA) . . . 12
- Factor Analysis . . . 12
- Locally Linear Embedding . . . 13

Probabilistic Learning **14**
Gaussian Mixture Models . . . 14
Information Criteria . . . 15
EM Algorithm . . . 15

Support Vector Machines **16**
Perceptrons and their Limitations . . . 16
Enter: Support Vector Machines . . . 16
Finding the Maximum Margin Classifier . . . 17
Questions and Clarifications . . . 17
Kernels . . . 18
Transformations . . . 18
Kernels and the Kernel Trick . . . 18
Kernel Clarification . . . 19
The Karush–Kuhn–Tucker Method . . . 20
The Lagrangian . . . 20
KKT: Extending the Lagrangian with Inequalities . . . 20
SVM Algorithm Outline . . . 21

Optimization and Search **22**
Line Searches and Gradient Descent . . . 22
Conjugate Gradients . . . 23

The Levenberg-Marquardt Algorithm **23**
Search Direction . . . 23
Algorithmic Sketch . . . 24

Genetic Algorithms **25**
General Description . . . 25
The Knapsack Problem . . . 25

Reinforcement Learning **26**
Action Selection . . . 26
Values . . . 27
The Algorithms . . . 27

Learning with Trees **27**
ID3 Example Setup . . . 28

Unsupervised Learning **29**
The K-Means Algorithm . . . 29
The K-Means Algorithm as a Neural Network . . . 29

Random Number Sampling **30**
The Box-Muller Scheme . . . 30
Proposal Distributions . . . 31

Markov Chain Monte Carlo (MCMC) **31**
Markov Chains . . . 31
Markov Chains and Sampling . . . 31

Gaussian Process Regression **32**
Gaussian Processes . . . 33
Performing Regression . . . 33
Algorithm Sketch . . . 34

Bayesian Networks **34**
Computational Examples . . . 34
Score-Based Approach . . . 34
Minimum Description Length (MDL) . . . 34
Independence Tests . . . 35
PC Algorithm . . . 35
Hidden Markov Models . . . 36
Forward Algorithm . . . 36
Baum-Welch (Foward-Backward) Algorithm . . . 36

Companion Introduction

This document is a partial collection of notes to accompany the second edition of Marsland's textbook *Machine Learning: An Algorithmic Perspective*. It was written as a part of an honors course in machine learning I took at Oglethorpe University.

The different sections of this were written with different intentions. Some of them are my notes from the lectures in class, while some of them are notes I prepared for my own "lectures" as a part of the honors class. This gives some variability to each section. Some sections are bullet points recounting the text's material, some offer the viewpoint of a different source, and some could serve as lecture notes. There is a rough chronological ordering to the notes, e.g. Chapter 1 is first, then Chapter 2, etc, but the sections themselves are not currently numbered here.

An Algorithmic Perspective covers a wide array of topics, provides exercises and references for further reading, almost always gives an outline of algorithms and often has implementations available online. There are numerous examples of algorithms being run, and the text *usually* manages to stay away from complicated formalism for introductions.

The presentation in the book could be better. There are enough typos that I worried what formulas were typed correctly in the book. At times the writing is verbose, and after pages of text the reader is left wondering what the point was. At times the opposite is true, and the reader is left to puzzle out how a mess of equations applies to the problem at hand. I did a lot of flipping through the book while I read it; both to get to the point of a long-winded section, and to try and find earlier explanations of mathematical topics.

The mathematics in the book is not pitched at a constant level. As the book continues, the math tends to get more sophisticated, but the author never takes the time to explain it all. That is fair, since this is not a math book. I would say that a good course in linear algebra is a good thing to have to make it through some of the more sophisticated sections. (At the time I wrote these notes, I hadn't had one yet. When explaining the math, I made judgement calls when I had to. Take it with a grain of salt.)

Finally, the book chooses to not use any machine learning libraries. I do not know if this is good or bad. On the one hand, we see how to build the algorithms from scratch, without too many optimizations. On the other hand, in an actual project, I would opt to use libraries like `scikit-learn` over the code in the book.

Hopefully the notes here will help someone else, even if only to frustrate them enough to write an even better companion.

Introduction

The introduction of Section 1.2 is dense, but contains important points. Machine learning generally tries to emulate the following process:

- *Remembering* past examples.
- *Adapting* memory of past examples for the present.

- *Generalizing* knowledge to handle new examples.

The four main types of learning algorithms are summarized in a list in Section 1.3, reproduced here:

- *Supervised learning*, where training inputs have target outputs.
- *Unsupervised learning*, where inputs do not have target outputs.
- *Reinforcement learning*, where an evaluator is told when they are wrong, but not the answer.
- *Evolutionary learning*, where random solutions are generated to maximize some fitness function.

These classifications are slightly arbitrary. Reinforcement learning and the fitness function in evolutionary learning are both very much like supervised learning.

A rough sketch of the overall learning process is given in Section 1.5, reproduced here:

1. Data Collection and Preparation
2. Feature Selection
3. Algorithm Choice
4. Parameter and Model Selection
5. Training
6. Evaluation

No learning takes place until the Training step. The steps before it are entirely focused on finding data and tailoring the learning algorithm choice to the problem at hand.

Section 1.6 ("A Note on Programming") could be important. Random programs aren't reproducible, so the `np.random.seed()` call is necessary for debugging specific cases. Aside from this method of debugging, reference programs are very useful — mostly to see where we went wrong in our implementation.

Complexity of Learning Algorithms

The reason for machine learning existing is because data sets are too large for humans to analyze. The inputs that learning algorithms receive will be much larger than most algorithms, so time complexity will be very important. For a data set of ten-thousand, the difference between $O(n^2)$ and $O(n^3)$ is huge.

Learning algorithms can be roughly split into two components:

- *Learning* (looking at training data); and
- *Application* (applying the trained algorithm to data).

The complexity of a learning algorithm can be measured in both of these parts.

Given n training inputs, the complexity of learning will be *at least* $O(n)$; we have to at least look at each input once. Generally, this process is expected to have more complexity and take longer than application.

Given n inputs, we would like for application to be *no more* complex than, say, $O(n^2)$; any more and we risk taking too long to be useful.

Supervised Learning

The book makes a point to introduce supervised learning here, but there is really only one point to make.

In supervised learning, a sequence of points (x_i, t_i) is given. For each x_i, the element t_i is the *correct* answer to classifying x_i. The goal is for the algorithm to learn from these pairs and be able to correctly match each x_i with t_i.

Usually, (x_i) is a sample of the set of all possible inputs, so the algorithm should also generalize its learning to inputs *not* in (x_i).

The regression example in 1.4.1 is a nice way to reframe what most people know about regression. Given a sequence of points (x_i, y_i), the sequence (x_i) is the sequence of x-coordinates, and (t_i) is the sequence of y-coordinates.

The classification example in 1.4.2 is basically regression but with a discrete range for the function.

Examples of Noise

The *noise* in a data set refers to the random fluctuations that are not relevant for classification or prediction. As a fabricated example, a collection of identical thermometers next to each other may give slightly different readings due to the noise of the environment around them.

For a more concrete example, suppose that we are using supervised learning, and we have the following two training points:

$$((0,0,0),1)$$
$$((0,0,0),0)$$

After training, if the input $(0,0,0)$ is seen, there is no reliable way to determine if the output should be 0 or 1. This is an extreme example of noise, where one input appears to have two different targets associated with it.

As a more realistic example, consider the following training points:

$$((0,0,0),0)$$
$$((1,1,1),1)$$

If we then see the input $(0,0,1)$, it is possible that the single 1 is due to random noise. Hence it is difficult to decide if the output should be 0 or some intermediate value.

Preliminaries

The terminology introduced in 2.1 is very biased towards supervised learning, and neural networks in particular. Input and output vectors ($\vec{x}$ and $\vec{y}$) are pretty general; but weights and activation functions ($\vec{W}$ and $g(\cdot)$) are mostly associated with neural networks, and targets ($\vec{t}$) are only for supervised learning.

Distance Formula

The distance formula is introduced here in the context of neural networks. The rough idea is to treat weights of neurons as coordinates in space, and have the neurons fire if they are "close enough" to an input vector. The distance function can be used generally to measure how far away inputs are from some target.

This concept doesn't seem to be used in the chapter, so it feels a little out of place.

The Curse of Dimensionality

The point of the hypersphere argument is unclear. In summary, as the number of dimensions increases, we will need much more data to train on. This is because the data will, in general, tend to be more spread out.

To explain the hypersphere argument, construct a unit box in an arbitrary number of dimensions. Think of the hypersphere as our target; that is, if we pick a random point from the unit box, we want it to be in the sphere. If we are picking the points randomly, the probability of a point being in the sphere is $V(S)/V(B)$, where $V(S)$ and $V(B)$ are the volumes of the sphere and box, respectively.

As the number of dimensions increases, the volume of the sphere does not grow as quickly as the volume of the cube, so the probability of picking a point in the sphere at random decreases simply because we added more dimensions.

This idea will apply to our learning algorithms. Our classifiers will be less accurate as the number of dimensions in our inputs increases, just because of the greater spread. Thus we will need much more data as the number of dimensions we use increases.

Accuracy Metrics

Accuracy metrics are useful to compare different methods of learning to each other.

Leave out some, crossfold validation (K-Fold Cross Validation): Split data into different chunks, and assign some chunks to be training, some to be validation, and some to be testing. For every possible combination of assignments, train a new model on the created data set, then take the best model. (There was a very pretty picture.)

Training, Testing, and Validation Sets

During learning, we are always given a data set of examples. In supervised learning, we will partition the data into three different sets:

- *Training*: Used to train a classifier.
- *Validation*: Used to test the accuracy of a classifier independent of the the classifier has never been trained on, to test for overfitting.
- *Testing*: Used to test the final accuracy of a classifier against data it has never even seen.

It is important that these sets be disjoint. The point of the validation set is to test both for accuracy and overfitting. If points from the validation set are in the training set, then the classifier may overfit to the validation set, destroying its intended metric. The same point can be made for the testing set.

It is also important that these sets be randomly chosen. If there is some sort of bias, then the classifier could inherit this bias. For example, suppose that a data set is sorted low-to-high for some feature. If we just take the first 60% of the data to be training, 20% to be validation, and 20% to be testing, then the classifier would only be trainined on the lower-end values of this feature.

Discussion questions:

- Why do we have training sets and test sets? (We need to test for generalization and overfitting.)
- What purpose does the validation set serve? (It offers an intermediate overfitting metric before testing.)
- When is the test set used? (Only after all training is complete.)
- Why is overfitting bad? (We might not be able to apply our algorithm to data outside of the training set.)

K-fold Cross-Validation

When we have lots of data, partitioning the data into training, validation, and testing sets is fairly simple. If we do not have lots of data, we can use the method of K-fold cross validation.

Briefly:

- Randomly partition the data into K subsets.
- Choose one subset for validation, another for testing, and combine the rest to be training.
- Train a classifier on these given sets.
- Repeat with a new classifier for all possible random partitions, then take the one with the best testing accuracy.

Accuracy Metrics

Accuracy is simply the ratio of the number of correct classifications to the total number of inputs: the estimated probability that we can correctly classify inputs.

- *Sensitivity* is the ratio correct positives over total positives present: the estimated probability that we can correctly classify inputs given that they are positive; cf. $P(\text{Classified True} \mid \text{True})$. *Specificity* is analogous, but for negatives; cf. $P(\text{Classified False} \mid \text{False})$.
- *Precision* is the ratio of correct positives over all classified positives: the estimated probability that an input was correctly classified positive given that we did classify it positive; cf. $P(\text{True} \mid \text{Classified True})$. Sort of the converse of sensitivity[1].
- The F_1 *metric* is the ratio of correct positives over the sum of correct positives and average number of misclassifications; it is one if and only if there are no misclassifications, and strictly less than one otherwise.

The text mentions the implicit assumption of a balanced data sets for these metrics. This assumption comes into play because many of these metrics will be misleading if a large majority of data is negative or positive. For example, consider a data set that is a snapshot of infections in a population of a rare disease. Most inputs will be negatives, so models that always classify negative will have specificity of one, but will not have actually learned how to detect the disease.

The Receiver Operator Characteristic Curve

The ROC curve is mostly well explained. A curve that looks as smooth as the ones in the book will require either a lot of points, or some kind of regression curve.

Naïve Bayes Classifier

Given the feature vector $\vec{X_j}$, to compute $P(C_i \mid \vec{X_j})$, we need to compute $P(X_j^1 = a_1,\ X_j^2 = a_2,\ \ldots,\ X_j^n = a_n \mid C_i)$. We may not be able to find a point such that $X_j^k = a_k$ holds for every valid k. This is an issue, because then the joint probability is impossible, and so $P(\vec{X_j} = \vec{a} \mid C_i) = 0$.

To make this easier, we make the simplifying assumption that the components $X_j^k = a_k$ are independent from each other. Then, $P(X_j^1 = a_1,\ X_j^2 = a_2,\ \ldots,\ X_j^n = a_n \mid C_i) = P(X_j^1 = a_1 \mid C_i)P(X_j^i = a_2 \mid C_i) \cdots P(X_j^n = a_n \mid a_n)$. This lets us find examples that satisfy each value separately, instead of looking for examples that satisfy *every* value restriction.

[1] This is bunk. – Dr. Patterson

Neurons and Neural Networks

Neurons

- Hebb's Rule postulates that the relationship between neurons is determined by how often they fire simultaneously. This allows us to formulate our model of how neurons learn.
- Given MCP neurons, we can only modify the weights and the threshold. These are updated using the perceptron algorithm.

A positive weight from node A to node B indicates that positive inputs in A generally imply B. A negative weight from A to B indicates that positive inputs in A generally imply not B.

Perceptrons

Introduction

- Perceptrons are networks of MCP neurons, where there is a layer of inputs and a layer of MCP neurons. Every input is connected to every neuron in the network.

Weights are updated by punishing the weights that do poorly (mismatch the target).

- Find nodes where $y_k \neq t_k$, and consider the positive weights.
- If $y_k > t_k$, then positive inputs are contributing too much.
- If $y_k < t_k$, then positive inputs are not contributing enough.

This method only works if we have a positive input. If we have a negative input, then we need to do the opposite. For example, if $y_k > t_k$ and x_k was negative, then the weight did not add enough value to x_k to stop the firing, so the weight must be increased. This rule is succinctly described by

$$\Delta w_{ik} = -(y_k - t_k)x_i,$$

where Δw_{ik} is the amount of change to weight w_{ik}, i.e. $w_{ik}^{k+1} = w_{ik}^{k} + \Delta w_{ik}^{ik}$. The following properties are easy to verify:

- If $y_k > t_k$ (misfire):
  - If $x_k > 0$, then the weight decreases. ($w_k x_k$ added too much.)
  - If $x_k < 0$, then the weight increases. ($w_k x_k$ did not take away enough.)
- If $y_k < t_k$ (should have fired):
  - If $x_k > 0$, then the weight increases. ($w_k x_k$ did not add enough.)
  - If $x_k < 0$, then the weight decreases. ($w_k x_k$ took away too much.)

The only remaining component is the learning rate, η. It is essentially like a step size, i.e. how far we should step in the direction of the "correct" descent.

Perceptron Convergence Theorem

Amazingly, for some datasets, Perceptrons are guaranteed to learn them in finite time. These datasets are called *linearly separable*, and to talk about them, we need to discuss more about what the Perceptron is actually doing.

The weights of perceptrons define a *decision boundary*. If an input falls on one side of the decision boundary, it it classified one way; if it falls on the other side, the other way. This decision boundary is, in general, a hyperplane defined by the equation $\vec{x}\vec{W}^T = 0$, where $\vec{W}$ is the weight vector for a single perceptron.

A dataset is called *linearly separable* if there exists a linear separator that correctly classifies the dataset. More formally, if we can partition the dataset into two disjoint classes C_1 and C_2, then the dataset is linearly

separable iff there exists some weight vector $\vec{W}$ such that $\vec{x}\vec{W}^T \geq 0$ for all $\vec{x} \in C_1$, and $\vec{x}\vec{W}^T < 0$ for all $\vec{x} \in C_2$.

The Perceptron Convergence Theorem states that, if a dataset is linearly separable, then the Perceptron learning algorithm will find a linear separating weight vector in finite time. That is, the Perceptron will be able to correctly classify inputs based on which side of the boundary they fall on in finite time.

Convergence Time Estimate

The Perceptron Convergence theorem states that, given a linearly separable dataset, the perceptron will learn a linear separator within $1/\gamma$ updates, where γ is defined to be the distance from the linear separator that the perceptron *will* converge to and the datapoint closest to it.

As an approximation of γ, we can use d, where d is half of the smallest distance between any two points of opposite classes. Then, we define $T' = 1/d^2$, which is *usually*[2] less than T. That is, we we create a convergence time estimate T' such that $T > T'$.

Perceptron Linear Regression

Essentially, we use calculus to find the "weight" vector $\vec{\beta}$ that minimizes the sum of squared errors

$$\sum_{k=0}^{N}(t_k - y_k)^2 = \sum_{k=0}^{N}(t_k - \sum_{j=0}^{m}\beta_j x_j).$$

Perceptrons try to draw lines *between* data, to classify points. Linear regression tries to draw lines *through* data, to match the points' class values.

MLP in Practice

Recall that slicing on numpy arrays can be done in multiple dimensions. For example, let `x` be a numpy matrix. Then:

- `x[0]` returns the first row.
- `x[0:3, 0]` returns the first element of the first three rows.
- `x[:3, 0]` is the same as above.
- `x[:3, :6]` returns the first six elements of the first three rows.
- `x[:3, -3:]` returns the last three elements of the first three rows.

When to Stop?

The naïve method is to train for T batches, for some constant T. There is no theoretical basis for this. As Dr. Patterson puts it, this method "kinda sucks."

The next, slightly less naïve method is to set an accuracy threshold for the training set. This could end in an infinite loop if that accuracy is never reached.

The more analytic approach is to make use of the validation set. In theory, while the MLP is learning the dataset or function, validation error will be decreasing. As we begin to overfit the data or function, the validation error will begin to increase again. Thus we will try to stop training at a local minimum of the validation error.

[2] He says, using proof by intimidation.

Finding this local minimum can be tricky. The book suggests keeping three previous models during training. If either of the two most recent validation errors has decreased enough from the validation error before themselves, then continue training. We check both so that a minor fluctuation in validation error will not halt training prematurely.

Probably Approximately Correct Learning

The book says that, given W weights, we should have roughly $10W$ datapoints to train on. Dr. Patterson says that, given L inputs, we should have roughly 2^L datapoints.

Neither of these have much theoretical grounding – they are just practical rules of thumb. The subset of machine learning called *Probably Approximately Correct Learning* is more theoretically grounded.

Universal Approximation Theorem

The Universal Approximation Theorem states that any neural network, with any number of of hidden layers and nodes, can be approximated by a single hidden layer MLP with some amount of nodes. This amount may be very large.

(Actually, it's about approximating convex functions. See the formal statement.)

1-of-N

Instead of using one number to encode N classes, 1-of-N classifiers use an N-tuple with components zero and one. As an example, with three classes, class 0 would become $(1, 0, 0)$, class 1 $(0, 1, 0)$, and class 2 $(0, 0, 1)$.

Compression

Neural network compression is very lossy. By cutting down from k inputs to $N < k$ activations in the hidden layer, we will always lose information.

(Can we get a bound on this? For example, if the hidden layers are activating to some finite set of integers, say zero and one, then there would be 2^L possible inputs to remember.)

Recipe for MLP

1. Select inputs and outputs.
2. Normalize inputs to some interval, usually $[0, 1]$.
3. Split dataset into training, validation, and testing.
    - Various splits are possible. Common ones are 50/25/25 and 60/20/20.
4. Decide on NN architecture.
    - How many layers?
    - How many nodes in each layer? (Too many and we lose accuracy, not enough and we might overfit.)
5. Train network until validation

Dimensionality Reduction

The book makes a big deal about the "curse of dimensionality," referring to how computational complexity and the amount of data needed to train a learning algorithm increases with the number of dimensions in a dataset. In this chapter, we will learn ways to fight the curse.

There are three main ways to do this:

- **Feature selection**: Throw out features that aren't useful to learning. We saw an example of this with tree learning. We looked at every feature and decided what would be the best one to examine, adding features until we had a perfect classifier or had no more features.
- **Feature derivation**: Create new features using combinations of old ones, then throw out the old ones. See Figure 6.1 for an example.
- **Clustering**: Group together similar datapoints and see if this provides hints for features that could be removed.

Linear Discriminant Analysis (LDA)

Linear discriminant analysis attempts to cluster data by projecting it onto a hyperplane. The hope is that classes will be separated far enough on the hyperplane so that their distance along the line will be enough to cluster them. This reduces the dimensions of the data down to one, a single scalar.

There are two metrics that we consider in LDA:

- **Within-class scatter**: The total amount of scatter between datapoints inside the same class. The total "spread" of the dataset on an *intra*-class level. Measured as

  $$S_W = \sum_{\text{classes } c} p_c \operatorname{cov}(c, c),$$

  where p_c is the probability of a class occurring, and $\operatorname{cov}(c, c)$ is the covariance matrix of the class c. Note that this is a matrix.
- **Between-class scatter**: The total amount of spread between classes themselves. The total "spread" of the dataset on an *inter*-class level. Measured as

  $$S_B = \sum_{\text{classes } c} (\vec{\mu}_c - \vec{\mu})(\vec{\mu}_c - \vec{\mu})^T,$$

  where $\vec{\mu}$ is the mean of the entire dataset. Note that this is a matrix, as we are dealing with column vectors.

We would like for S_W to be small, and S_B to be big. That is, we would like to maximize the ratio S_B/S_W.

We will be maximizing this ratio by choosing the hyperplane with the "best" spread to project our data onto, described by the unit vector $\vec{w}$. For any datapoint $\vec{x}$, the scalar projection onto our hyperplane is $\vec{w} \cdot \vec{x}$. If we compute the within-class and between-class scatter using the scalar projections, we obtain (through some linear algebra)

$$S'_W = \vec{w}^T S_W \vec{w}$$
$$S'_B = \vec{w}^T S_B \vec{w}.$$

Thus, we want to maximize the ratio $\frac{\vec{w}^T S_W \vec{w}}{\vec{W}^T S_B \vec{w}}$. In general, this is only easy if we have two classes. With two classes, the plane that will maximize this ratio is in the direction of

$$\vec{w} = S_W^{-1}(\mu_1 - \mu_2).$$

Principal Components Analysis (PCA)

Principal Components Analysis attempts to separate unlabeled data by transforming points so that only dimensions that datapoints "vary" along are examined. It is a type of feature selection.

The motivation for the algorithm is very linear algebra heavy. The gist is this:

- We have a (centered) data matrix X with covariance matrix $\text{cov}(X)$.
- Besides centering the data, we now want to rotate it. This will be done by multiplying by a rotation matrix P^T. Our new dataset is $Y = P^T X$.
- We want, for some reason, to choose P^T such that $\text{cov}(Y)$ is diagonalizable. That is,

$$\text{cov}\,(Y) = \begin{bmatrix} \lambda_1 & \cdots & \cdots & \cdots & \\ \vdots & \lambda_2 & \cdots & \cdots & \\ \vdots & \vdots & \lambda_3 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \\ \cdots & & & & \lambda_n \end{bmatrix}.$$

- From definition of covariance, we arrive at $\text{cov}(Y) = P^T \text{cov}(X) P$. Since P is a rotation matrix, its transpose is its inverse. This leads us to

$$P\,\text{cov}(Y) = PP^T\,\text{cov}\,(X)\,P = \text{cov}\,(X)\,P.$$

- If we write P as a list of column vectors $P = [\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n]$, then we have

$$P\,\text{cov}\,(Y) = [\lambda_1\vec{p}_1, \lambda_2\vec{p}_2, \ldots, \lambda_n\vec{p}_n] = \text{cov}\,(X)\,P.$$

- Since the two matrices are equal, so are their columns. Splitting this into a system of equations, we have

$$\vec{\lambda}\vec{p}_k = \text{cov}\,(X)\,\vec{p}_k.$$

- This leads us to the conclusion that the columns of P are eigenvectors of $\text{cov}\,(X)$.
- The covariance is square and symmetric, so its full complement of eigenvalues are orthogonal, and thus form an eigenspace. This means that each eigenvalue roughly corresponds to one dimensions. We want to choose the dimensions that have the larger eigenvalues.

Algorithmic sketch:

1. Begin with centered data matrix X.
2. Compute the covariance matrix C.
3. Find the eigensystem of C, and arrange the eigenvectors in decreasing order.
4. Select a number of dimensions to keep, then take this number of the largest eigenvectors.
5. Use the selected eigenvectors to transform the data, dropping the dimensions that aren't selected.

Factor Analysis

The text's explanation of factor analysis is esepcially brief.

Factor analysis works to create a simple linear model to describe a dataset where the elements of the model are noisy random variables. That is, given a data matrix X, we construct the model

$$X = WF + \epsilon,$$

where F is a random matrix describing a set of "factors" that we think are largely responsible for the data, W is a matrix of "factor loadings" that describe how the factors affect the measurements, and ϵ is a normal random matrix with mean zero and variances ψ_i that represents the "noise" in the data. Our goal is to find a matrix W and set of variances ψ_i that maximize the likelihood of observing X given our model.

As a brief example, say that X are the results of an IQ test. Then, we might have

$$F = \begin{bmatrix} \text{IQ}_1 & \text{Height}_1 \\ \text{IQ}_2 & \text{Height}_2 \end{bmatrix}$$

and

$$W = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix},$$

since the results of the IQ test are dependent only on the subject's IQ.

Factor analysis is an EM algorithm, meaning it is composed of two very simple steps that are impossible to explain. Essentially, we compute the expectation of observing our data given the current model, differentiate it to maximize the expectation, update W and each ψ_i, and repeat until convergence. The steps are outlined on pages 141 and 142, with a partial code listing on page 142.

Locally Linear Embedding

(The text has a lot of confusing notation in this section. I decdided what things should be, but the code listings given should probably be consulted.)

Locally Linear Embedding (LLE) works to reduce the dimensionality of a dataset by transforming higher dimensional vectors into lower dimensional ones that minimize the "reconstruction error" of the lower dimenison. Broadly speaking, the algorithm chooses neighborhoods around points, then places the points into a lower dimensional space around the neighborhood that minimizes some error metric.

The algorithm first chooses a neighborhood around each datapoint. These neighborhoods can be created in many different ways, e.g. in the "topological" sense of all points within a distance d, or in the k-nearest neighbors sense of grabbing the closest k points.

After choosing a neighborhood around a point $\vec{x}_i$, the remaining points are used to "reconstruct" (approximate) $\vec{x}_i$. Each point in the dataset is assigned a weight in this reconstruction, denoted W_{ij}. For every $\vec{x}_j$ not in the neighborhood of $\vec{x}_i$, we set $W_{ij} = 0$. Then, we have

$$\vec{x}_i \approx \sum_{j=1}^{N} W_{ij}\vec{x}_j,$$

where N is the number of datapoints. The dataset is then assigned a reconstruction error ϵ that is the sum-of-squares error in the reconstruction approximations:

$$\epsilon = \sum_{i=1}^{N} \left| \vec{x}_i - \sum_{j=1}^{N} W_{ij}\vec{x}_j \right|^2.$$

There is no need to construct W by hand. There is a method for choosing W such that ϵ is minimized:

- For each point $\vec{x}_i$, construct its neighborhood. Create a list of neighbor points $\vec{z}_j$, $j = 1, 2, \ldots, K$. Let $\vec{d}_j = \vec{z} - \vec{x}_i$ be the difference from each neighbor to $\vec{x}_i$. Form a matrix D of these differences. (D must be $N \times N$, which I am not sure how neighborhoods are guaranteed to be disjoint. For the brave, there is a partial code listing on page 146.)
- Compute the "local covariance," defined as $C = DD^T$. (I believe that this works out to actually be the covariance of D, but it's hard to follow with all the vectors around.)
- Solve $CW = I$ for W, where I is the $N \times N$ identity. That is, $W = C^{-1}$.
- Set $W_{ij} = 0$ when x_j is not in the neighborhood of x_i. (In the topological sense, I feel confident that "in the neighborhood of" is a symmetric relation, but not so sure for K-nearest neighbors.)

- Normalize the elements so that the matrix sums to one, i.e. set $W_{ik} = W_{ij} / \sum W_{ij}$.

This gives us an "ideal" reconstruction matrix W to play with.

Next, we get to the "embedding" part. We will choose an arbitrary lower dimension L, and construct the point $\vec{y}_i$ that corresponds to $\vec{x}_i$ in $\mathbf{R^L}$. In this lower dimensional space, we create a new reconstruction error ϵ_L, defined as

$$\epsilon_L = \sum_{i=1}^{N} \left| \vec{y}_i - \sum_{j=1}^{L} W_{ij} \vec{y}_j \right|^2 .$$

Note the similarity with the previous reconstruction error. To futher this analogy, we want to choose $\vec{y}_i$ such that ϵ_L is minimized. The text punts hard, and says that these vectors are given by the eigenvectors of the *quadratic form matrix*

$$M = (I - W)^T (I - W).$$

The full algorithm strings these steps together, and can be found on page 145.

Probabilistic Learning

A criticism of neural networks is that we cannot "see" what they are doing. Weights and biases may work, but they have no intuitive meaning. Other learning methods work well and are more transparent. Decision trees[3] are one example. In this chapter, we will present another, based on statistical methods.

Highlights:

- EM Algorithm
- Unsupervised learning example
- Nearest neighbor methods
- Statistical methods

Gaussian Mixture Models

Note that the text says, in Chapter 7, that "we *will* [emphasis added] see lots of ways to deal" with unlabeled examples in Chapter 6. We have already seen Chapter 6, in theory.

- M: number of Gaussian distributions.
- α_m: "weight" of distribution m.
- $\sum_{m=1}^{M} \alpha_m = 1$.
- $\phi(x; \mu_m, \sigma_m)$: normal function with specified parameters.

Define

$$P(x_i \in C_k) = \frac{\alpha_k \phi(x_i; \mu_k, \sigma_k)}{\sum_{m=1}^{M} \alpha_m \phi(x_i; \mu_m, \sigma_m)}$$

to be the probability of input x_i belonging to class C_k.

Say that we have two normals G_1 and G_2, where p is the probability of a datapoint beloning to G_1. Suppose that p is also a random variable, and has probability density function π. From the above model, we have the random variable

$$y = pG_1 + (1 - p)G_2.$$

The probability of any particular value of y occuring is

$$P(y) = \pi\phi(y; \mu_1, \sigma_1) + (1 - \pi)\phi(y; \mu_2, \sigma_2).$$

[3]Decision trees, Chapter 12, are referred to in the past tense in this chapter, Chapter 7.

Our goal now is to iterate, trying to move the Gaussian models to fit our data correctly. For example, we will move μ_1 until the curve over μ_1 seems to be a good fit of the data.

Essentially, γ_i is an estimate of $P(x_i \in C_1)$. After calculating every γ_i, we have a rough picture of what the data looks like from our previous guesses. From this, we move the means, standard deviations, and probability π around so that we fit the data better. We repeat this until our parameters converge.

Example:

$$D = \{5, 10, 4, 6, 5, 4, 6, 11, 10, 9, 5, 7, 3\}.$$

From drawing a graph, we conjecture that there are two normals, with

$$\begin{aligned} \mu_1 = 5, \quad \sigma_1 = 2, \quad \pi = \frac{9}{13} \\ \mu_2 = 10, \quad \sigma_2 = 1. \end{aligned}$$

After performing first M-step, we have

$$\begin{aligned} \mu_1 &= 4 \\ \mu_2 &= 11 \\ \bar{y} &= 6.54 \\ \sigma_1 = \sigma_2 &= 6.38 \\ \pi &= 0.4 \end{aligned}$$

Information Criteria

We are familiar with the validation set. This set can be used to determine when to stop training, i.e. stop when the validation error is at a local minimum. However, there are other ways to determine when to stop training. In particular, one way to do this is by measuring *information criteria.*

Given models, we have two information criteria. The *Akaike Information Criterium*

$$\text{AIC} = \ln \mathcal{L} - k,$$

and the *Bayesian Information Criterium*

$$\text{BIC} = 2 \ln \mathcal{L} - k \ln N.$$

Here, k is the number of parameters in the model, N is the number of datapoints, and $\mathcal{L}$ is "best likelihood of the model." The book is very quiet about what this is or how to calculate it.

EM Algorithm

1. **Expectation**: Compute the expected liklihood of the model.
2. **Maximization**: Update the model to maximize the expected likihood.

Support Vector Machines

Perceptrons and their Limitations

In Chapter 3, we encountered a simple type of neural network which could compute many functions: the **Perceptron**.

As a quick summary of the material in the Perceptron section, recall:

- The perceptron defines a *decision boundary*, $\vec{x}\vec{W}^T = 0$
  - In 2D, this is a line between classes.
  - In 3D, this is a plane.
  - In higher dimensions, we call this a hyperplane.
- The Perceptron Convergence Theorem guarantees finite-time convergence for linearly separable data sets.
  - If we can draw a line, or plane, or hyperplane between classes in input space, then the perceptron will find a line that can do this.
  - This will happen in a finite amount of time.

Now, this theorem only applies to linearly separable datasets. If a dataset is not linearly separable, then there might be issues.

- Perceptrons might not work if the input space is not linearly separable.
  - For example, take the 2D XOR function.
  - Cannot draw a line between the XOR data, so it's not linearly separable.
  - A perceptron will waffle between incorrect inputs, and never converge to a correct answer.
  - This can be fixed by adding extra dimensions.

Enter: Support Vector Machines

(This was written for section 8.1, but also from a Stanford webpage on SVMs.)

Support Vector Machines (SVMs) provide two important things:

1. An objective way to choose the *best* decision boundary; and
2. A set of methods to transform any dataset into one that is linearly separable. (Example: 3D XOR function.)

That is, using SVMs, we can always linearly separate data by transforming it.

Also, after a brief training period, SVMs only need to remember a select few training points to classify new points.

- The *margin* of a linear separator is the largest radius M such that no data points lie within M units of the hyperplane defined by the separator.
- If there exists a linear separator with a margin larger than any other, it is the *maximum margin classifier*. This is considered to be the optimal linear separator.
- Points that are exactly M units away from a linear classifier are called *support vectors*. (Measured by $|\vec{w} \cdot \vec{x} + b| = M$.)
- The margin of a linear classifier with weights $\vec{w}$ is $1/|\vec{w}|$. (The derivation is rather long, and very confusing in our textbook. See the Stanford webpage for a better derivation.)

- Given a point $\vec{x}$ with target $t = \pm 1$, a measure of the goodness of the classification of $\vec{x}$ is the *functional distance* $t(\vec{w} \cdot \vec{x} + b)$.
  - General measure of how well classified a point is; the larger the value is, the better it was classified.
  - Positive for correct classifications, and negative for incorrect classifications.

Finding the Maximum Margin Classifier

We need to

1. Find the largest margin possible; and
2. Make sure that classifications are still "good."

Maximizing the margin is the same as minimizing $\frac{1}{2}|\vec{w}|^2$. The $\frac{1}{2}$ factor is for convenience, and the square does not effect anything, since the square root function is increasing for nonnegative inputs anyway.

For the "good" requirement, we arbitrarily require that the functional distance is greater than one. That is, that $t(\vec{w} \cdot \vec{x} + b) \geq 1$. Equality is obtained only with support vectors.

This gives us a constrained optimization problem:

Let $\vec{x_i}$ and t_i, $i = 1, 2, \ldots, n$ be sequences of input vectors and targets, respectively, where $t_i = \pm 1$. Then,

$$\text{minimize } \frac{1}{2}|\vec{w}|^2 \text{ under the constraint } t(\vec{w} \cdot \vec{x} + b) \geq 1, i = 1, 2, \ldots, n.$$

This problem can be solved using the Karush–Kuhn–Tucker (KKT) method, which is a generalization of the method of Lagrange multipliers that allows for inequalities. Once the problem has been set up, we can place it into a solver that will do the heavy lifting for us. We will not get too much into this right now[4], but here are the high points:

- We seek a $\vec{w}^*$ and b^* such that $\vec{w}^* \cdot \vec{x} + b* = 0$ is a maximum margin classifier.
- The solver will provide us with a vector of Lagrange multipliers $\vec{\lambda}$ such that

  $$\vec{w}^* = \sum_{k=1}^{n} \lambda_k t_k \vec{x_k}$$

  and

  $$b^* = \frac{1}{N} \sum_{\text{support vectors } s} \left(t_j - \sum_{k=1}^{n} \lambda_k t_k \vec{x_k} \cdot \vec{x_s} \right),$$

  where N is the number of support vectors.

This method will allow us to find an optimal minimum classifier for any linearly separable dataset. The next step for SVMs is handling datasets that are not linearly separable.

Questions and Clarifications

- In the definition of a margin, M is defined to be the largest radius such that no datapoints lie within M units of the hyperplane. In Figure 8.2, the margin bounding region is drawn with a solid line, indicating that no points lie exactly M units away. In other words, for every datapoint $\vec{x}$, we have $|\vec{w} \cdot \vec{x} + b| > M$.

  Almost immediately, support vectors are defined to be the points $\vec{x}$ such that $\vec{w} \cdot \vec{x} + b = M$. So, let's all just agree that the boundary line in Figure 8.2 should be dotted, and that every datapoint $\vec{x}$ satisfies $|\vec{w} \cdot \vec{x} + b| \geq M$.

[4] Because I don't understand it.

- After Equation 8.11, the text mentions that the SVM's optimal linear classifier has the property that prediction relies on computing the dot product of the input vector and *only* the support vectors. However, the relevant term is

$$\vec{z} \cdot \sum_{k=1}^{n} \lambda_k t_k \vec{x_k},$$

which is certainly over all training vectors $\vec{x_k}$, and not just the support vectors. What are they talking about?

Kernels

As previously mentioned, the point of a linear classifier to draw a hyperplane between linearly separable datasets. This was not always possible when datasets were not linearly separable, as in the case of the XOR function. To fix this, we will introduce kernels, which are computationally simple ways to make any dataset linearly separable.

Transformations

Let X be our input space; usually this is some $\mathbf{R^n}$. To transform our datapoints $\vec{x} \in X$, we will define a *feature mapping* $\phi \colon X \to V$ such that $\phi(\vec{x})$ is the transformation of the input $\vec{x}$, where V is some other input space, usually of a higher dimension.

Assuming that this mapping creates a linear separable dataset, we can apply the KKT method to find our optimal linear classifier in this new input space, giving us the optimal parameters

$$\vec{w}^* = \sum_{k=1}^{n} \lambda_k t_k \phi(\vec{x_k})$$

$$b^* = \frac{1}{N} \sum_{\text{support vectors } s} \left(t_j - \sum_{k=1}^{n} \lambda_k t_k \phi(\vec{x_k}) \cdot \phi(\vec{x_s}) \right)$$

To make predictions of a new input $\vec{z}$, we need to map z with ϕ, then compute

$$\vec{w}^* \cdot \phi(\vec{z}) + b^*,$$

using these vectors from the new input space V. The vectors of V may have a very large dimension, and this may take a very long time. To get around this computation, we will introduce *kernels*.

Kernels and the Kernel Trick

Let $K \colon X^2 \to \mathbf{R}$ be a symmetric mapping. Then, K is a *kernel function* iff it is positive definite (p.d.), as defined below. That is, the function K is a kernel iff

- (Positive definite) For all $< c_0, c_1, \ldots, c_n > \in \mathbf{R^n}$ and $\vec{x_i}\vec{x_j} \in X$,

$$\sum_{\substack{1 \le j \le n \\ 1 \le i \le n}} c_i c_j K(\vec{x_i}, \vec{x_j}) \ge 0;$$

- (Symmetric) $K(\vec{x}, \vec{y}) = K(\vec{y}, \vec{x})$ for all $\vec{x}, \vec{y} \in X$.

The key point of this definition is an application of Mercer's Theorem[5]: *if there exists a positive definite kernel on an input space X, then there exists a feature mapping $\phi\colon X \to V$ such that $K(\vec{x}, \vec{y}) = \phi(\vec{x}) \cdot \phi(\vec{y})$.* That is, if we have a kernel, then we are always implicitly computing the dot product of vectors in a higher dimension, without ever working in that higher dimension. Using kernels to avoid this higher-dimensional computation is the *kernel trick*.

Practically, the kernel trick means that by replacing all dot products that the SVM computes with the kernel computation, we will be implicitly using higher dimensions. To force linear separability, we punt into a higher dimension, then the kernel trick punts us right back to where we started.

With the kernel trick in mind, we do not need to find a feature mapping ϕ anymore; we only need to find positive definite kernels that are easy to compute. Luckily, there are a few standard kernels and functions that they come from:

- The polynomial kernel of degree d: $K(\vec{x}, \vec{y}) = (1 + \vec{x} \cdot \vec{y})^d$.
- The sigmoid kernel with parameters k and δ: $K(\vec{x}, \vec{y}) = \tanh(k\vec{x} \cdot \vec{y} - \delta)$.
- The radial basis function with parameter σ: $K(\vec{x}, \vec{y}) = \exp(-|\vec{x} - \vec{y}|^2/2\sigma)$. (The text incorrectly gives the vector factor as $(\vec{x} - \vec{y})^2$, which is not a scalar.)

Kernel Clarification

Question: How does using a kernel actually avoid the computation in the higher dimension? The formally doesn't readily explain this. Answer: Because of vector stuff.

Recall that the optimized weight vector, $\vec{w}^*$, is given by the equation

$$\vec{w}^* = \sum_{k=1}^{n} \lambda_k t_k \vec{x_k},$$

where t_k is the target for $\vec{x_k}$, and $\vec{\lambda}$ is given by `cvxopt` after employing KKT. If we use a mapping $\phi\colon X \to V$, then the classification computation for an input $\vec{z}$ becomes

$$\left(\sum_{k=1}^{n} \lambda_k t_k \phi(\vec{x_k}) \right) \cdot \phi(\vec{z}) + b^*.$$

Of course, the inner product is distributive over vector addition, so this becomes

$$\sum_{k=1}^{n} \lambda_k t_k \phi(\vec{x_k}) \cdot \phi(\vec{z}) + b^*.$$

At this point, knowing a kernel, we can replace the inner product $\phi(\vec{x_k}) \cdot \phi(\vec{z})$ with $K(\vec{x}, \vec{z})$, where K is a kernel of ϕ.

Equipped with the kernel, the only thing left to compute is compute b^*. This computation only requires computing dot products $\phi(\vec{x_k}) \cdot \phi(\vec{x_j})$, which can be replaced with kernel computations.

The book talks about another K, the Gram matrix, or the kernel of distances. This is a confusing name and symbol; as far as I can tell it is not the actual kernel. The book is remarkably silent on how this matrix works.

[5]Mercer's Theorem is a result in functional analysis. We will not talk about it or functional analysis ever again.

The Karush–Kuhn–Tucker Method

If you were sick of the math *before*...

The Karush–Kuhn–Tucker method is used in support vector machines (SVMs) to find the optimal linear classifier. It does this by minimizing $\frac{1}{2}|\vec{w}|^2$ subject to $t_i y_i \geq 1$ for $i = 1, 2, \ldots, n$. The goal of KKT is to handle this inequality.

Before introducing KKT, we will describe the Lagrangian to see where KKT differs.

The Lagrangian

We wish to minimize the function $f(\vec{x})$ subject to the list of constraints $g_k(\vec{x}) = 0$, $k = 1, 2, \ldots, n$. There are two possibilities:

1. The gradient of f is parallel to the gradient of each g_k, so that moving along $g_k(\vec{x}) = 0$ will not decrease f; or
2. The gradient of f is zero, so that f is at a minimum.

The minima of f will *possibly* occur when one of these two conditions do. These conditions can be succinctly summarized by defining the Lagrangian

$$\mathcal{L}(\vec{x}, \vec{\lambda}) = f(\vec{x}) + \sum_{k=1}^{n} \lambda_k g_k(\vec{x}),$$

where $\vec{\lambda}$ is a vector of multiples meant to describe all possibilities of parallel vectors. At critical points of $\mathcal{L}$, i.e. when $\nabla\mathcal{L} = \vec{0}$, we have

$$\frac{\partial \mathcal{L}}{\partial \lambda_k} = g_k(\vec{x}) = 0,$$

so that the constraints are satisfied, and

$$\nabla_{\vec{x}} f(\vec{x}) + \sum_{k=1}^{n} \lambda_k \nabla_{\vec{x}} g_k(\vec{x}) = \vec{0},$$

a technical condition that ensures that $\vec{x}$ is a local minimum[6]. This gives us enough equations to completely determine $\vec{x}$ and $\vec{\lambda}$, if such a solutions exists.

In summary, *if a point is a minimum for the Lagrangian, it is a minimum for f subject to the given equality constraints.* Thus we solve the system given by $\nabla\mathcal{L} = \vec{0}$.

Note that this method only handles equality constraints, i.e. $g(\vec{x}) = 0$. KKT will allow us to handle inequalities as well.

KKT: Extending the Lagrangian with Inequalities

For KKT, we keep our equality restraints $g_k(\vec{x}) = 0$, but now add the list of inequality restraints

$$h_j(\vec{x}) \leq 0, \quad j = 1, 2, \ldots, m.$$

Just as we introduced the vector $\vec{\lambda}$ for equality constraints, we will introduce the vector $\vec{\mu}$ for inequality restraints. Our Lagrangian becomes

$$\mathcal{L}(\vec{x}, \vec{\lambda}, \vec{\mu}) = f(\vec{x}) + \sum_{k=1}^{n} \lambda_k g_k(\vec{x}) + \sum_{k=1}^{m} \mu_k h_k(\vec{x}).$$

[6] See http://math.stackexchange.com/a/453421/261157. The rough sketch is that, for a point $\vec{x}$ to be a local minimum for f constrained by each g_k, then $\nabla f(\vec{x})$ must belong to the vector space spanned by each ∇g_k. Or, $\nabla f(\vec{x}) = \sum_{k=1}^{n} \lambda_k \nabla g_k(\vec{x})$, as stated.

The procedure for this extended Lagrangian is identical to the original one for $\vec{x}$. We find the critical points with respect to $\vec{x}$, giving

$$\nabla_{\vec{x}} f(\vec{x}) + \sum_{k=1}^{n} \lambda_k \nabla_{\vec{x}} g_k(\vec{x}) + \sum_{k=1}^{m} \mu_k \nabla_{\vec{x}} h_k(\vec{x}) = \vec{0}. \tag{1}$$

This ensures that $\vec{x}$ is a local minimum.

Next, we find the critical points with respect to $\vec{\lambda}$, but in a slightly different way. For some reason[7], the solution $\vec{x}$ depends on $\vec{\lambda}$, so we may not treat functions of $\vec{x}$ as constant with respect to $\vec{\lambda}$. Practically, this means that we solve

$$\nabla_{\vec{\lambda}} f(\vec{x}) + \sum_{k=1}^{n} \nabla_{\vec{\lambda}} (\lambda_k g_k(\vec{x})) + \sum_{k=1}^{m} \nabla_{\vec{\lambda}} (\mu_k h(\vec{x})) = \vec{0}. \tag{2}$$

Finally, the inequality restraints are completely different. Instead of solving $\nabla_{\vec{\mu}} \mathcal{L} = 0$, for technical reasons, we have the following requirements:

$$\mu_j h_j(\vec{x}) = 0, \quad j = 1, 2, \ldots, m \tag{3}$$

$$\mu_j \geq 0. \tag{4}$$

Solving the system given by (1), (2), (3), and (4) will give us possible minima.

SVM Algorithm Outline

After discussing the theoretical aspect of SVMs, we turn our attention to the algorithm itself and its implementation. The implementation is provided by the book, so we will mostly sketch the outline and discuss the software used.

The Python package `cvxopt` (convex optimization) is required for the book's implementation. The free and open source package package numerically solves convex optimization problems. More details can be found at the cvxopt homepage.

The algorithm is fairly straightforward, math aside. The authors choose to support three different kernels: the linear, polynomial, and radial basis function (RBF) kernels. There is some slight initialization to handle this fact.

1. Compute the Gram matrix $K = XX^T$, where X is the column vector containing every datapoint.

- Linear kernel: return K.
- Polynomial kernel of degree d: return $\frac{K^d}{\sigma}$, where σ is a parameter.
- RBF: return $K_{ij} = \exp(-|x_i - x_j|^2/\sigma^2)$.

2. Solve for $\vec{\lambda}$ using `cvxopt`.

We will use `cvxopt`'s quadratic solver, `cvxopt.solvers.qp()`. Its signature is `cvxopt.solvers.qp(P, q, G, h, A, b)`, and it minimizes

$$\frac{1}{2}\vec{x} \cdot P\vec{x}$$

subject to

$$G\vec{x} \leq \vec{h}, \quad A\vec{x} = \vec{b}.$$

The text derives the appropriate values for these variables in terms of the Gram matrix K and $\vec{\lambda}$.

[7] Which I am not clear on at all.

3. Compute b^*.

This completes the set up of the SVM. It is now ready to classify new datapoints.

Optimization and Search

Line Searches and Gradient Descent

In many types of learning, we want to minimize some error function f. There are many ways to do this, but the simplest is called a *line search*. In this method, we have some initial guess for the minima, $\vec{x}_0$, and we follow a line in some preset direction for some preset distance to the next guess. That is, we pick our next guess with the equation

$$\vec{x}_{n+1} = \vec{x}_n + \eta_n \vec{p}_n,$$

where $\vec{p}_n$ is the direction of the line, and η_n is the distance to follow along the line.

With this equation, we are free to pick the direction $\vec{p}_n$ and the distance η_n. Different methods of choosing these will give us different search techniques. For example, setting $\vec{p}_n = -\nabla f(\vec{x}_n)$ gives us gradient descent.

The choice of $\vec{p}_n = -\nabla f(\vec{x}_n)$ is usually due to the fact that $-\nabla f(\vec{x}_n)$ is the direction of greatest decrease for f. However, we can also derive this direction by considering different approximations for f.

These approximations are extensions of taylor expansions of single-variable functions. Recall that a single variable function $g(x)$ may be approximated as

$$g(x) \approx g(x_0) + g'(x_0)(x - x_0) + g''(x_0)(x - x_0)^2 + \cdots .$$

For functions of multiple variables, we will replace g' and g'' with more general "derivatives" called the gradient and Hessian, respectively.

First, we have the linear approximation

$$f(\vec{x} + \vec{p}) \approx f(\vec{x}_n) + \nabla f(\vec{x}_n) \cdot \vec{p} + o(|\vec{p}|).$$

If we minimize this with respect to $\vec{p}$, we end up with $\vec{p} = -\nabla f(\vec{x}_n)$, which recovers gradient descent.

Next, we have the quadratic approximation

$$f(\vec{x} + \vec{p}) \approx f(\vec{x}_n) + \nabla f(\vec{x}_n) \cdot \vec{p} + \frac{1}{2}\vec{p}^T H(f(\vec{x_n}))\vec{p} + o(|\vec{p}|^2),$$

where $H(f(\vec{x}))$ is the Hessian matrix of second derivatives,

$$H(f(\vec{x})) = \begin{bmatrix} f_{x_1x_1}(\vec{x}) & f_{x_1x_2}(\vec{x}) & \cdots & f_{x_1x_n}(\vec{x}) \\ f_{x_2x_1}(\vec{x}) & f_{x_2x_2}(\vec{x}) & \cdots & f_{x_2x_n}(\vec{x}) \\ \vdots & \vdots & \vdots & \vdots \\ f_{x_nx_1}(\vec{x}) & f_{x_nx_2}(\vec{x}) & \cdots & f_{x_nx_n}(\vec{x}) \end{bmatrix}.$$

If we minimize this equation with respect to $\vec{p}$, then we obtain

$$\vec{p} = -(H(f(\vec{x}_n)))^{-1}\nabla f(\vec{x}_n).$$

This direction is called the *Newton direction*.

The computational complexity of the Newton direction is fairly high; we need to compute the inverse of the Hessian matrix. However, according to the text, the payoff is that our step size η is always set to one.

Conjugate Gradients

The goal of conjugate gradients is to spend a little more time thinking about what directions and how far along them to step to minimize the number of steps taken. In fact, the goal is, in n dimensions, to take exactly n steps to reach the minimum of a function.

Except for when the error function is linear, this goal is almost never reached, but the method of conjugate gradients will still improve on line searches.

For the direction $\vec{p}_n$, conjugate gradients makes two choices. For the first step, we follow gradient descent and set $\vec{p}_0 = -\nabla f(\vec{x}_0)$. After this, we apply what is called a *Gram–Schmidt process* to discover the directions. In n dimensions, we create the sequence of n coordinate axis vectors $\vec{u}_k$. Then, the direction is given by

$$\vec{p}_n = \vec{u}_n + \sum_{k=1}^{n-1} \beta_k \vec{p}_k,$$

where

$$\beta_k = \frac{|\nabla f(\vec{x}_k)|^2}{|\nabla f(\vec{x}_{k-1})|^2}.$$

For the step size α_n, conjugate gradients uses Newton–Raphson iteration to derive the optimal

$$\alpha_n = \frac{\nabla f(\vec{x}_n) \cdot \vec{p}}{\vec{p}^T H(f(\vec{x}_n))\vec{p}}.$$

From here, the algorithm on page 200 is fairly straightforward. It is essentially the line search, but making the choices described above.

The Levenberg-Marquardt Algorithm

(The text does a terrible job of explaining where equations and final statement of problems come from. In particular, the jump to Equations (9.16) and (9.17) is abrupt. Wolfram's MathWorld was a great help here.)

The Levenberg-Marquardt Algorithm (LMA) is a line-search trust-region algorithm used to minimize linear *and* non-linear least-square problems. That is, it seeks to minimize functions of the form

$$f(\vec{x}) = \frac{1}{2}\sum_{k=1}^{n} r_k(\vec{x})^2 = \frac{1}{2}|\vec{r}(\vec{x})|^2.$$

As a *line search* algorithm, it has a particular way of determining the direction to search in, involving Jacobians and linear least-square methods. Overall, it is a *trust region* algorithm because it makes assumptions about the function being minimized within a particular region, called a *trust region*.

Search Direction

Recall that a line search algorithm works by choosing an initial guess $\vec{x}_0$, then defining each later point by the equation

$$\vec{x}_{n+1} = \vec{x}_n + \alpha_n \vec{p}_n,$$

where α_n is the search length and $\vec{p}_n$ is the search direction.

LMA sets $\alpha_n = 1$ and then chooses $\vec{p}$ in a way that blends two other methods: gradient descent and the Gauss-Newton method. Gradient descent chooses the direction $\vec{d}_n = -\nabla f(x_n)$, which ends up being defined by

$$\nabla f(x_n)_j = \sum_{k=1}^{n} r_k(\vec{x}) \frac{\partial r_k}{\partial x_j}.$$

If we let J denote the Jacobian of $\vec{r}$, then this gradient works out to be

$$\nabla f(\vec{x}) = J^T \vec{r}.$$

Jumping straight to Jacobians, the Gauss-Newton method direction is

$$\vec{g} = (J^T J)^{-1} J^T \vec{r}.$$

(For technical reasons, i.e. J may not be square, this inverse does not always simplify nicely.) Finally, LMA chooses the direction

$$\vec{l} = -(J^T J + \lambda I)^{-1} J^T \vec{r},$$

where the parameter λ is a "dampening" parameter and I is the identity matrix. According to smart people, if λ is large, then this direction is approximately the same as gradient descent; if it is small, then the direction is approximately the same as Gauss-Newton.

The trust region as a region where we assume that the function f being minimized is roughly quadratic. This is an approximating assumption LMA makes that throws away terms of the taylor expansion of f. The size of the trust region is roughly inversely proportional to the value λ. The "trust region growing" means that λ decreases, or that we are more comfortable choosing the more aggressive Gauss-Newton direction. The "trust region shrinking" means that λ increases, or that we are more cautious and prefer the standard gradient descent direction.

Algorithmic Sketch

The only head-scratching part in this is evaluating the quadratic trust region assumption. If a function is quadratic, then we can closely predict its change. If this prediction is off, then we shrink our trust region. If it is acceptable, then we change nothing. If it is good, then we increase our trust region.

(This is a good-faith effort to combine the strange text algorithm with its online implementation.)

- Choose an initial guess $\vec{x}_0$ for the minimum of $f(\vec{x})$.
- While $J^T \vec{r} > \epsilon > 0$ and a maximum number of iterations is not exceeeded:
  - repeat until a new point is found
    - Solve $\vec{l} = -(J^T J + \lambda I)^{-1} J^T \vec{r}$ using linear least-squares methods.
    - Set $\vec{x}_{n+1} = \vec{x}_n + \vec{l}$.
    - Evaluate how well our trust region assumption is working.
      - actual $= |f(\vec{x}_{n+1}) - f(\vec{x}_n)|$
      - predicted $= \nabla f(\vec{x}_n) \cdot (\vec{x}_{n+1} - \vec{x})$ (note the probable typo on page 196)
      - set $\rho =$ actual/predicted
    - If $0 < \rho < 0.25$:
      - accept new step
    - If $0.25 < \rho$:
      - accept new step
      - increase trust region (decrease λ)
    - Else ($\rho \leq 0$ (?)[8]):
      - reject new step
      - reduce trust region (increase λ)

[8] This doesn't make much sense. This is negative only if "predicted" is negative, which doesn't mean anything about the relative magnitude of errors. It is difficult to parse the author's implementation, since they apply `np.linalg.norm`, which I believe makes everything positive, then makes the same positive/negative check.

Genetic Algorithms

A genetic algorithm applies concepts from evolution to search a solution space in both exploitative and exploratory ways.

General Description

A genetic algorithm requires: - An *alphabet* to construct *strings* from. - A *fitness function* to describe how good a string is. - An initial *population* of test strings. - A way to *generate and replace* new offspring.

Strings:

- Some way to encode information about a solution.
- Should *not* include any information aside from the solution, i.e. no information about optimal solutions.
- Keep as small as possible.

Fitness function:

- Takes a string and outputs a number called the *fitness* of that string.
- Higher fitness means more fit.
- Nonnegative function.

Population:

- Large list of possible solutions as strings.

Parent Selection:

When looking at a population, we need to pick some number to act as parents for the next generation. In general, we would like to select the most fit population members, but also to allow some exploration. See page 215.

The Knapsack Problem

Suppose that we have a knapsack with a finite carrying capacity, and we want to place objects of different weights into it. In general, there is too much weight to carry everything. How do we place items into the knapsack to maximize the weight that we take?

This problem is simple, but is very computationally complex. It is NP-complete.

Here are a few algorithms to convince us that the problem is hard:

- Exhaustive algorithm
  - Checks every possible solution.
  - Guaranteed to find the optimal solution, given enough time.
  - For n items, there are 2^n ways to pack them (not accounting for invalid combinations), so this is $O(2^n)$.
- Greedy algorithms
  - Grab heaviest/lightest object at each step.
  - Not guaranteed to find the optimal solution, and usually won't.
  - For n items, there are $O(n)$ steps.

Between these, we can choose exponential time for an accurate solution, or linear time for a poor solution. We hope to use genetic algorithms to find acceptable solutions while keeping fairly low time complexity.

Suppose that we have L items, a max weight of M, and that the weight of item k is $w(k)$.

We will use binary strings to encode solutions, where a 1 in position k means we are taking item k, and a 0 there means we are not taking it.

The fitness function we will use is

$$f(k) = \begin{cases} w(k), & \text{if } w(k) \leq M, \\ w(k) - 2(w(k) - M), & \text{otherwise.} \end{cases}$$

From here on out, we run the GA as described in the previous section.

Reinforcement Learning

For every reinforcement problem, we must have the following things:

- **State space**:
  - set of states that an agent may be in
  - information that an agent has access to
  - environment information, if it is changing
- **Action space**:
  - set of *all* possible actions
  - the state space may encode all *possible* actions from a given state

(We would, in general, like to minimize the size of these spaces.)

- **Reward function**:
  - set by the environment
  - returns an *actual* reward for every action taken
- **Discounting**:
  - subjective reward function that attempts to predict future rewards from an action
  - Choose $\gamma \in [0, 1)$, a measure of how much we believe our prediction.
  - Our estimated reward at time t is weighted by γ^t.

Action Selection

Every agent must have a way to select an action at each state. To assist in this, $Q_{s,t}(a)$ is defined as the average reward for choosing action a at state s after choosing it t times in the past. We would like for $Q_{s,t}(a)$ to converge to the *actual* reward for the action.

We can think of $Q_{s,t}(a)$ as a three dimensional table; or as the continual replacement of a two dimensional table as t increases.

There are various ways to choose the next action:

- **Greedy**: choose the action a that maximizes $Q_{s,t}(a)$.
- **ϵ-Greedy**: generally choose the greedy solution, but with ϵ probability, uniformly choose a random action.
- **Softmax**: Choose an action relative to the softmax probabilities; i.e. set

  $$P(Q_{s,t}(a)) = \frac{\exp(Q_{s,t}(a)/\tau)}{\sum_k \exp(Q_{s,t}(k)/\tau)}$$

  and choose action a relative to this probability[9].

[9] That is, pick a random number r. If r is less $P(Q_{s,t}(1)$, then choose action 1. If r is larger than it, then examine the second action. And so on until the last action is reached.

RL problems can be split into two classes: episodic and continual. An *episodic* problem has a definite goal to be reached. A *continual* problem has no definite goal, and goes on indefinitely.

Clearing up the example:

- The agent does *not* know the overall topography.
- Absorbing states have no actions to choose from.

Values

The explanation of values is a little abstruse. Roughly, values are the reward that an agent expects to get from an action at a certain state. That is, they are a "subjective reward." The goal of reinforcement learning is to find a policy of choosing actions that will maximize these subjective values.

This policy will be created by initializing all values to small, random numbers, then exploring the state space. As the agent explores the state space, it uses the rewards it obtains to update the values it believes each action and state should have. With any luck, the values will converge to the actual rewards.

There are two different value functions:

- **State value function**:
$$V(s) = E[r_t \mid s_t] = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+r} \mid s_t = s].$$
  - Let current policy set action; average over all actions.
- Action-value function:
$$Q(s,a) = E[r_t \mid s_t, a_t] = E[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a]$$
  - Let current policy select action; take the value from only those actions.

The optimal value function Q^* is given by the authors as

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}).$$

This doesn't seem to build up to much. Later, they give an update formula

$$Q(s,a) \leftarrow Q(s,a) + \mu(r + \gamma Q(s', a') - Q(s,a)).$$

The Algorithms

Q-Learning and Sarsa do roughly the same thing. The only difference is that Sarsa uses the "current policy" to make *all* of its decisions, where Q-Learning uses ϵ-greedy for some decisions.

Learning with Trees

This section corresponds to Chapter 12. Some of the code in this chapter lacks a frusterating amount of documentation.

- ID3 is greedy; it grabs the feature with the greatest gain at each step. When there are no features left, it grabs the most common class and hopes that it works.

– It's worth noting that the book requires computing the entropy of the entire dataset. We don't have to! Gain is given in Equation 12.2. When trying to maximize this, Entropy(S) is constant, so the feature that has the smallest $-\sum_{f \in values(F)}$ term will maximize gain.

To see this, let $\sum_F$ and $\sum_{F'}$ be the sigma terms for two different features. If

$$\sum_F < \sum_{F'},$$

then applying the decreasing function $f(x) = \text{Entropy } S - x$ to this inequality gives

$$\text{Entropy } S - \sum_F > \text{Entropy } S - \sum_{F'},$$

or

$$\text{Gain}\,(S, F) > \text{Gain}\,(S, F')\,.$$

- The entropy of a set with classes $C_1,\ C_2,\ \ldots,\ C_n$ is calculated with the probability p_k being the probability of the class C_k occurring in the data set. Equation 12.3 shows this with two classes: true and false.
- `calc_entropy` does not actually compute the entropy that these sections talk about; it only *helps* compute it.
- `calc_info_gain` does not actually compute the gain; it computes the sigma term from Equation 12.2, and `make_tree` later computes the gain. It also generalizes to an arbitrary number of classes without telling us. Here's my best guess at documentation:

```
def calc_info_gain(data, classes, feature):
    """Compute the sigma term from Equation 12.2 from choosing the given feature.

    data: List of vectors [feature_1, feature_2, ..., feature_n].
    feature: Integer that indexes the feature to be used from the dataset.
    classes: List of the same length of `data`, where `classes[k]` is the class
             of the kth datapoint.

    The possible values will be found by looking directly at the data.
    """
```

ID3 Example Setup

Here is the setup for an ID3 example (this was the majority of a class period):

Let sex denote our class, with Male and Female being values. We have the attributes Height (Tall, Medium, Short) and Weight (Heavy, Median, Light).

Consider the following dataset:

Height	Weight	Sex
T	H	M
S	L	F
M	M	M
M	M	M
S	H	M
S	L	F
T	L	F
T	L	M
M	L	F

From the note above, calculate the sigma term in Equation 12.2 for each Height and Weight, then split on the feature that has the smallest.

Unsupervised Learning

Chapter 14 introduces us to the concept of *unsupervised learning*, or learning where an algorithm must learn to classify data without being given pre-classified examples.

The biggest change from supervised to unsupervised is the lack of targets to train against. However, because there are no targets, we also can't use any of our old error functions. They rely on targets or other external information that we don't have on hand anymore. Thus, we need error functions that can operate with only a set of datapoints.

As an example of how such an algorithm would work, the text introduces the *K-means algorithm.*

The K-Means Algorithm

The K-means algorithm is roughly the unsupervised analog of the K-nearest neighbors algorithm. For K-nearest neighbors, an input was classified by choosing the most common class out of the K-nearest examples. This idea requires that we know the class of the examples; i.e., it is a supervised algorithm. For K-means, we won't have those targets to pick from.

The K-means algorithm operates on the same assumption as K-nearest neighbors: datapoints that are close to each other are likely in the same class, and classes are usually far enough apart to be separated. The K-means algorithm assumes that the data is split into K classes which are separated enough to be clustered together, then tries to find the center of each cluster iteratively.

The K-means algorithm needs two things: a distance measure, or metric, and a way to compute the mean. Usually, we use the Euclidean metric, but there are others.

A solid outline of the K-means algorithm is given on page 283. One important thing to note is that the learning process looks at every datapoint before making any updates. This will change when we introduce neural networks to solve the problem.

A final point is that the K-means algorithm doesn't know anything about class labels. It can only cluster data, not say what that data is. For example, suppose that we have a dataset of flowers with the classes "rose" and "lily." We are given a set of flowers without labels, so we try K-means to cluster the examples. This gives us two clusters, but we don't know which cluster corresponds to roses, and which to lilies. Assuming that the first cluster corresponds to the first class, say "rose," could be incorrect. There is not much we can do to fix this problem.

The K-Means Algorithm as a Neural Network

We can express an *on-line* version of the K-means algorithm as a neural network. On-line indicates that we will make updates to the network after seeing each datapoint, not after seeing all of them. There are various reason that we might want to do this. Perhaps we are being fed data one point at a time and need to make predictions faster than we can receive data.

The neural network is a single-layer perceptron, with one input neuron for each feature in the dataset. There are K output neurons representing the K clusters, whose weights are the locations of the center of each cluster. The activation function used is something like the distance between two points (more on that later). To choose which output fires, hard-max is used. That is, the neuron that a point is closest to fires and gets updated. An outline of the algorithm is given on page 289.

There are a few technical notes about the book's implementation: normalization and the activation function. The activation function used is $g(\vec{x}) = \vec{x} \cdot \vec{w}$, where $\vec{w}$ is the weight vector. As the book claims, this *effectively* computes the distance between $\vec{w}$ and $\vec{x}$ under the assumption that $\vec{w}$ and $\vec{x}$ are unit vectors. What they mean by this is that, if $\vec{x} \cdot \vec{w}$ is at a maximum, then, $|\vec{w} - \vec{x}|$ is at a minimum. So we pick the output neuron that maximizes $\vec{x} \cdot \vec{x}$ using hard-max, and we know that $\vec{x}$ is closest to this neuron.

To see why this is true, we will examine $|\vec{w} - \vec{x}|^2$. When this is minimized, then $|\vec{w} - \vec{x}|$ will also be minimized. Now,

$$\begin{aligned} |\vec{w} - \vec{x}|^2 &= (\vec{w} - \vec{x}) \cdot (\vec{w} - \vec{x}) \\ &= \vec{w} \cdot \vec{w} - 2\vec{x} \cdot \vec{w} + \vec{x} \cdot \vec{x} \\ &= |\vec{w}|^2 - 2\vec{x} \cdot \vec{w} + |\vec{x}|^2. \end{aligned}$$

Since $\vec{x}$ and $\vec{w}$ are unit vectors, $|\vec{w}|^2 = |\vec{x}|^2 = 1$, so

$$|\vec{w} - \vec{x}|^2 = 2 - 2\vec{x} \cdot \vec{w},$$

or

$$\vec{x} \cdot \vec{w} = 1 - \frac{|\vec{w} - \vec{x}|^2}{2}.$$

Thus, when $\vec{x} \cdot \vec{w}$ is maximized, so is

$$1 - \frac{|\vec{w} - \vec{x}|^2}{2},$$

meaning that

$$\frac{|\vec{w} - \vec{x}|^2}{2}$$

is at a minimum, and so $|\vec{w} - \vec{x}|^2$ is as well.

Random Number Sampling

Sampling from a probability distribution is a key part of any algorithm with stochastic behavior. Thus, how we get those samples is important.

The Box-Muller Scheme

Often, we want to sample from a standard normal distribution. If we can sample from a uniform distribution on $[0, 1]$, we can generate pairs of numbers from independent standard normal distributions. This is called the Box-Muller Scheme, and it's pretty simple:

Given two uniformly distributed random variables U_1 and U_2 on $[0, 1]$, let $\theta = 2\pi U_1$ and $r = \sqrt{-2\ln(U_2)}$. Then, $x = r\cos\theta$ and $y = r\sin\theta$ are Gaussian variables with mean zero and standard deviation one.

The proof of this requires a little bit of multivariate probability theory. The gist is that, for two independent, standard Gaussian variables X and Y, their product is distributed with density

$$f(x, y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} = \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}}.$$

Next, convert to polar, i.e., set $r^2 = x^2 + y^2$ and $\theta = \arctan(y/x)$. If we can sample r and θ from a uniform distribution that maintains this relationship, then we can work in reverse to recover independent Gaussians.

It works out that θ is uniformly distributed on $[0, 2\pi]$, and r is on $[0, 1]$. To sample θ, we can pick $\theta = 2\pi U_1$. To sample r, we need to solve

$$P(r \leq R) = 1 - e^{-\frac{r^2}{2}} = 1 - U_2,$$

which works out to be $r = \sqrt{-2\ln(U_2)}$.

Proposal Distributions

There is an alternate method to perform the Box-Muller Scheme, which uses the idea of sampling and rejection.

Let U_1 and U_2 be uniformly distributed random variables on $[-1, 1]$, and let $w = \sqrt{U_1^2 + U_2^2}$, so that

$$w^2 = U_1^2 + U_2^2.$$

If $w^2 < 1$, i.e. the point (U_1, U_2) lies in the unit circle, then

$$x = U_1\sqrt{\frac{-2\ln(w^2)}{w^2}}$$

and

$$y = U_2\sqrt{\frac{-2\ln(w^2)}{w^2}}$$

are independent, standard Gaussian variables.

This new method can be quicker, but requires that we might have to reject some points. (What's the probability that we'll have to do that?) The idea of "sample-and-reject" can be used to sample from arbitrary probability distributions.

Markov Chain Monte Carlo (MCMC)

Previously, we considered ways to approximate sampling from a distribution that is difficult to sample from. This included the Box-Muller Scheme (using a uniform distribution to sample from a standard normal) and the rejection sampling distribution (using an easier "proposal distribution" to approximate samples). In this section, we look at using Markov chains to accomplish the same thing.

Markov Chains

A Markov Chain is a sequence of states where the probability of being in the next state depends only on the current state. For examples, think of the standard "random walk" or "Brownian motion" in probability.

The text does not make use of a lot of Markov chain theory, but it does place a lot of technical restrictions on the chains. In particular, it requires that the chains are strongly ergodic. This means that they are irreducible, positive recurrent, and aperiodic. This doesn't ever make a big appearance in the section, but it is quite easy to create a non-ergodic Markov chain. If ergodicity is actually important, we should be careful when just playing with Markov Chains.

Markov Chains and Sampling

Some Markov chains have *limiting distributions*. These are analogous to stable equilibrium points in differential equations. For a limiting distribution, no matter what state we begin in, after a large number of steps, the

probability of our end position will be described by a constant distribution. Formally, if $p_{ij}^{(n)}$ is the probability of being in state i after n steps from state j, then

$$\lim_{n\to\infty} p_{ij}^{(n)} = \pi_i,$$

where π_i is a constant. Because we require that our Markov chains be strongly ergodic, each of them has a unique limiting distribution. (Take my word for it.)

Markov chains are very simple to sample from. Start at any state and take a large number of random steps. The end state is our sampled value. If we can create an ergodic Markov chain whose limiting distribution is the distribution we want to sample from, then we can approximate samples from that distribution. For example, if our Markov chain had a Poisson limiting distribution, then the end state after a large number of random walks would approximately be a random value from a Poisson distribution[10].

To generalize the idea of sampling, we have a proposal distribution $q(x_i \mid x_i)$. That is, a distribution whose next sample depends only on the current sample. This is sort of a "generalized" Markov chain. We start with an initial guess and then we take another sample from this proposal distribution. We take this new sample only if it is roughly "more likely" than the current sample. This is done by computing an "acceptance ratio" and accepting the point with probability equal to this acceptance ratio.

The algorithm runs as follows:

Metropolis-Hastings Algorithm:

1. Decide to sample n values and pick an initial sample x_0.
2. Set $k = 0$.
3. Loop until $k = n$:
   i. Sample a new value x^* from the proposal distribution.
   ii. Compute the acceptance ratio
   $$\alpha = \min\left(1, \frac{p(x^*)q(x^* \mid x_k)}{p(x_k)q(x_k \mid x^*)}\right).$$
   - Note that α increases if q says that moving to x^* from x_k is more likely than moving the opposite direction the current sample. It also increases if p says that x^* is more likely than x_k in general.

   iii. Pick a uniform number u from $[0, 1]$.
   iv. If $u < \alpha$, then set $x_{k+1} = x^*$ and $k \leftarrow k + 1$. Otherwise, repeat.
4. Celebrate with your n samples.

Gaussian Process Regression

When given a dataset to perform regression on, we must choose what model to use. Is the data linear? Exponential? Or maybe something really weird, like sinusoidal? Generally, to pick the best model, we go through a trial-and-error process. The goal behind Gaussian Processes is to generalize this trial-and-error process by selecting different types of models from a probability distribution.

[10]It will not do to simply select a state at random. This would be sampling from a uniform distribution. The walks have to actually be performed, since that's what the limiting distribution requires.

Gaussian Processes

A stochastic process is a collection of random variables. For example, the function $f(x) = \exp(-ax)$, where a is sampled from a standard normal distribution, can be thought of as a stochastic process. For each sampled value of a, we obtain a new distribution $f(x)$. (This is shown in Figure 18.3.)

A *Gaussian process* (GP) is a stochastic process where each variable has a Gaussian distribution, and joining any finite number of them results in another Gaussian distribution. For example, the stochastic process $f(x)$ above is a Gaussian process[11].

A regular Gaussian distribution is determined by a mean and covariance matrix. Likewise, a Gaussian process is determined by mean and covariance *functions*. That is, a function that describes the mean and covariance matrix of each random variable. It turns out that any function that is a *kernel* from SVMs will do as a covariance function. A common covariance function is the RBF kernel,

$$k(\vec{x}, \vec{y}) = \sigma_f^2 \exp\left(-\frac{|x-y|^2}{2l^2}\right),$$

where σ_f and l are positive parameters.

Performing Regression

In the context of regression, we make the assumption that sampling is a Gaussian Process. That is, each datapoint comes from a normal distribution and any collection of datapoints is also from a normal distribution. This will allow us to predict the value of a testpoint using the mean of a created normal distribution.

First, for any set of examples, we subtract off the mean so that everything has mean zero. This means that we can ignore the mean function. This gives us a set of points $(\vec{x_k}, t_k)$. Next, we choose some covariance function $k(\vec{x}, \vec{y})$. From this, we build the covariance matrix for the normal distribution of the dataset. It is defined by

$$K_{ij} = k(\vec{x}_i, \vec{x}_j).$$

At this point, we are ready to perform regression on a test point $\vec{x^*}$. Because we assumed that sampling is a Gaussian process, the test point $\vec{x^*}$ with the dataset are normally distributed. This new distribution has the covariance matrix (obtained by appending a row and column to K)

$$K_{\vec{x}^*} = \begin{bmatrix} K & \vec{k}^* \\ \vec{k}^{*T} & k^{**} \end{bmatrix},$$

where K is the covariance matrix of the dataset, $\vec{k}^*$ is the vector of covariances $k(\vec{x}^*, \vec{x})$, and $k^{**} = k(\vec{x}^*, \vec{x}^*)$.

It works out that the probability that $\vec{x^*}$ should have value y^* given the dataset, i.e.

$$P(y^* \mid (\vec{x}_1, t_1), (\vec{x}_2, t_2), \ldots, (\vec{x}_n, t_n))$$

follows the normal distribution

$$N(\mu = \vec{k}^* K^{-1} \vec{t}, \sigma^2 = k^{**} - \vec{k}^* K^{-1} \vec{t}),$$

where $\vec{t} = [t_1, t_2, \ldots, t_n]$. To predict the value that $\vec{x}^*$ should have, we simply take the mean of this distribution.

[11] I think.

Algorithm Sketch

- Pick a covariance function $k(\vec{x}, \vec{y})$, and create the dataset $(\vec{x}_k, t_k)$ with mean zero.
- Compute the covariance matrix $K_{ij} = k(\vec{x}_i, \vec{x}_j)$.
- For each testpoint $\vec{x}^*$:
  - Compute the "covariance vector" $\vec{k^*}$.
  - Compute the covariance $k^{**} = k(\vec{x}^*, \vec{x}^*)$.
  - Predict $y^* = \vec{k}^{*T} K^{-1} \vec{t}$.

Bayesian Networks

Computational Examples

Dataset:

A	B	C
T	F	T
T	T	F
T	F	T
F	T	T
F	T	T
F	F	T
F	F	F
F	F	T

$P(A = T) = \frac{3}{8}$

$P(B = T) = \frac{3}{8}$

$P(A = T \mid B = T) = \frac{1}{3}$

$P(B = T \mid A = T) = \frac{1}{3}$

The variables A and B are dependent since $P(A = T \mid B = T) \neq P(A = T)$. That is, knowing that B occurs affects the probability of A occuring.

The goal is, given a dataset, to create a Bayesian network describing the dependency relations.

Score-Based Approach

This is very similar to genetic algorithms.

1. Generate and/or modify Bayesian networks.
2. Evaluate quality of networks using some "score" metric.
3. Repeat steps 1 and 2 with variations on the "best" network.

Minimum Description Length (MDL)

This is one possible way to score a network.

Suppose that we have M datapoints on a graph G with N attributes. Then, the MDL is defined as

$$L(G) = \frac{1}{M} \log \prod_{k=1}^{M} P(D_k \mid G) = \frac{1}{M} \sum_{j=1}^{N} \sum_{k=1}^{M} \log(P(X_j = d_{kj} \mid C(X_j)),$$

where X_j is the jth variable, d_{kj} is the jth attribute of the kth datapoint $C(X_j)$ is the conditioning set of X_j, or the set of parents of X_j.

For example (because we desperately need one), consider the dataset from above, with the graph G described by $A \to C$ and $B \to C$. Then,

$$\begin{aligned} L(G) &= \frac{1}{8} \sum_{k=1}^{8} \sum_{j=1}^{3} \log P(X_k \mid C(X_k)) \\ &= \frac{1}{8} [3 \log P(A = T) + 5 \log P(A = F) + \dots] \end{aligned}$$

Independence Tests

Formally, two variables X and Y are independent iff $P(X \mid Y) = P(X)$. When sampling values of X and Y, we may never be sure if they are *actually* independent, since we will never have the full picture. Patterson introduced us to the χ^2 (chi-squared) test for independence.

Null hypothesis (H_0): the variables X and Y are independent given Z.

χ^2 statistic: square of actual minus expected, divided by expected. (?)

Example: Suppose that 100 people took a course. Of those, 50 passed the course. Of the original 100, 50 passed the first exam, and only 25 of those that passed the exam passed the course. Is passing the course independent of the first exam?

However (and I quote), of 50 passing the first exam, 35 passed the course. Are they independent? (What?)

$$\chi^2 = \frac{(35 - 25)^2}{25} = \frac{100}{25} = 4.$$

Roughly, this means that this is unlikely.

PC Algorithm

1. Begin with a complete, undirected graph.
2. (Remove direct indepencies.) For all pairs of variables (X, Y), if X and Y are independent under some independence test, remove the $X \to Y$ edge.
3. (Remove indirect independencies.) For all pairs (X, Y), and for all Z that are adjacent to X *or* Y, if X and Y are independent given Z under some independence test, remove the $X \to Z$ edge.
4. Repeat step 3 for all sets Z of size 2, 3, ..., until out of sets. That is, check if X and Y are independent given Z_1 and Z_2 for all adjacent Z_1 and Z_2, and so on, for all sizes of the set Z.
5. We now have an undirected skeleton. If (X, Y) are *both* adjacent to Z, then check if X and Y are independent given Z. If they are dependent, then orient edges as $X \to Z \leftarrow Y$.
6. Repeat step 5 until all pairs are tested.
7. If $A \to B$, $B - - - C$, the pair (A, C) is not adjacent, and $C \not\to B$, then set $B \to C$.

8. If $A - - - B$ and there's a *directed* path from A to B, then set $A \to B$.
9. Repeat steps 7, 8 until there is no change. Orient remaining edges randomly.

Hidden Markov Models

Forward Algorithm

Goal: Estimate $P(\text{state})$ given a set of observations and the current model.

Using the Law of Total Probability,

$$P(O) = \sum_{r=1}^{R} P(O \mid \Omega_r) P(\Omega_r),$$

where Ω_r is a possible sequence of states. Since Ω_r represents a sequence of states, we will assume that

$$P(\Omega_r) = \prod_{t=1}^{T} a_{\Omega_{r,t}\Omega_{r,t+1}},$$

where T is the number of states in each Ω_r and a_{ij} are the transition probabilities of the Markov chain. We will also assume that

$$P(O \mid \Omega_r) = \prod_{t=1}^{T} b_{\Omega_{r,t}}(O_k),$$

where $b_i(o)$ is the probability of emitting observation o given that we were in state i. Thus,

$$P(O) = \sum_{r=1}^{R} \prod_{t=1}^{T} b_{\Omega_{r,t}}(O_t) a_{\Omega_{r,t}\Omega_{r,t+1}}.$$

Runtime: $O(TN^T)$.

Forward Trellis: R and T are usually *really* big, and that means that $O(TN^T)$ is hard. The "forward trellis" avoids this by focusing only the the observations seen and the most likely steps. This gets us down to $O(TN^2)$.

Baum-Welch (Foward-Backward) Algorithm

Goal: Learn transition and observation probabilities.

Issues:

- Unsupervised learning.
- NP-Complete (almost certainly exponential)

Variables:

- $\beta_i(t)$ = probability that we are in state i at time t.
- π_i = probability of state i in the initial distribution.
- a_{ij} = probability of moving from state i to state j.
- $b_i(o_k)$ = probability of observing o_k after being in state i.

See page 340 for actual definitions.

Algorithm:

- Initialize π to uniform distribution and a_{ij}, $b_i(o_k)$ to be random probabilities.
- While the algorithm has not converged:
    1. E-step: calculate α and β. For each o_t, i, j, compute $\xi_{i,j,k}$.
    2. M-step: