2011

A Method for Automatically Generating Network Structure in Bayesian Classifiers

Lionel R. Barrow
Bard College

A Method for Automatically Generating Network Structure in Bayesian Classifiers

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Lionel Barrow

Annandale-on-Hudson, New York
May, 2011

Abstract

The Naive Bayesian algorithm for classification has been a staple in machine learning for decades. Simple and efficient, the algorithm makes unrealistic independence assumptions about the data; yet it performs very well, often nearly matching the performance of far more complex modern algorithms. Only recently have researchers understood the theoretical reasons for this unreasonably good performance. In 2004, Professor Harry Zhang of the University of New Brunswick articulated the notion of a dependence-derivative factor, which more defines precisely how much Naive Bayes is harmed by certain violations of its independence assumption. In this project, I present a way to use Zhang's dependence derivatives to create classifiers similar to Naive Bayes, but with a network struture more resilient of these violations.

Contents

Dedication

Dedicated to my parents, Mary and John, my brother Jack, my sister Janet and my two dogs, Starr and Pepper.

Acknowledgments

I'd like to thank my project advisor, Cliona Golden, for her tireless support though all the craziness, initial lack of direction, procrastination, and babies. Sven Anderson, for first getting me interested in AI with his fantastic course last spring. Vasant Honavar and Rafael Jordan of Iowa State University, who instructed me in machine learning during my time at ISU's excellent REU in Bioinformatics. All my math and CS professors through the years, including (chronological order) Greg, Mary, Lauren, Jules, Sven (again), Maria and Jim, Becky, Ethan, Bob, and my academic advisor, Sam, who I regret never taking a class with. I also want to thank all the other professors who helped me during my time here, particularly Bill Mullen, Susan Rogers, and Leon Botstein. I'd like to thank my family for all their enthusiasm, especially Dad, who kept trying to explain the project to people in all the wrong contexts, and of course, last but never least, my friends. Thanks for the support, understanding, stress breaks, snacks, late-night food runs, beer, coffee, and fun.

1
Introduction

1.1 Classification

Classification is the familiar problem of examining something and deciding what it is. A typical classification problem might involve being given a description of an object – say a piece of fruit – and being asked to determine what type of fruit is being described. While humans are able to classify objects intuitively and without much thought, developing algorithms to enable machine classification is an ongoing problem in the field of machine learning, where classification is the dominant problem in the subfield of supervised learning.

Formally, a **classifier** is a function f that maps an **example** E to a **class** c. An example is usually represented as an ordered n-tuple of **attributes** $(x_1, x_2, ..., x_n)$. Typically, we create a classifier according to an algorithm, which examines a number of examples and creates a classifier function based on the data. In most cases, the classifiers produced by a given algorithm running on the same data will be equivalent, meaning that for identical inputs, they will give identical outputs; because of this, it is common to refer to a classifier function and the algorithm that created it interchangeably.

When managed well, classifiers can quickly and quietly reduce the number of inconsequential decisions in our lives. Probably the most well-known example of a classifier is an email spam filter, which simply determines if an incoming email matches the known profile of an unwanted advertisement. Spam filters are an illustrative example of the development of classification technology; though it took a long time for spam filters to consistently "get it right," modern algorithms work so well that it is easy to forget that they are there. Classifiers can also have far more dramatic applications, however, than just sorting email. The diagnosis of disease can be treated as a classification problem, as can the assignment of a credit score.

Classification can be broken into two separate arenas: binary classification and multi-class classification. In a binary classification problem, the classifier attempts to distinguish between only two classes; this reduction in complexity typically makes it much easier to articulate the intuition behind how a classification algorithm works. For the most part, we will discuss classifiers as if they are binary – distinguishing only between a positive class (+), and a negative class (−). Most of the ideas presented can be quickly generalized to the multiple-class case. If the generalization is non-obvious, we may comment on how it works out.

A note on terminology: I will refer to attributes X_i as random variables that take on specific values x_i when in an example E. Properly speaking, an example is therefore the intersection of the events $X_1 = x_1$, $X_2 = x_2$, and so on, up until $X_n = x_n$. In a given classification problem, all examples in a data set always have the same number of attributes, which I refer to as n.

To make sure the reader understands what classification data look like, I have provided an example below. A group of researchers at the University of Waikato created a classification environment in Java called Weka, which is widely used. Weka reads data from

attribute-relation file format (ARFF) files. The following is an example of an ARFF file [5]:.

```
% 1. Title: Iris Plants Database
%
% 2. Sources:
%      (a) Creator: R.A. Fisher
%      (b) Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
%      (c) Date: July, 1988
%
@RELATION iris

@ATTRIBUTE sepallength  NUMERIC
@ATTRIBUTE sepalwidth   NUMERIC
@ATTRIBUTE petallength  NUMERIC
@ATTRIBUTE petalwidth   NUMERIC
@ATTRIBUTE class        {Iris-setosa,Iris-versicolor,Iris-virginica}

@DATA
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-versicolor
4.6,3.1,1.5,0.2,Iris-versicolor
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
```

```
5.0,3.4,1.5,0.2,Iris-virginica
4.4,2.9,1.4,0.2,Iris-virginica
4.9,3.1,1.5,0.1,Iris-setosa
```

When the classifier trains, it will examine each line of data as a separate instance. So, in this case, the first line of data is a Iris-setosa plant with a sepal length of 5.1, sepal width of 3.5, petal length of 1.4, and petal width of 0.2. I will refer to this as the example $E = (5.1, 3.5, 1.4, .2)$ with class value Iris-setosa. The **attributes** are the descriptions like sepal length, sepal width, and so on, while the **attribute values** are numbers: 5.1, 3.5, etc.

1.2 Bayesian Classifiers

In this project, we focus on Bayesian classifiers, which are named after their reliance on Bayes' Rule. Before we enter the discussion, however, we have to deal with a slightly aggravating problem of terminology in this field. We are about to enter a discussion of the probabilities of various terms, some of which are single values – or the intersection of single values – from a continuous range. Strictly speaking, the probability of such terms is 0 – what we actually mean is the value of a probability density function (for continuous variables) or probability mass function (for discrete variables) at a specific point. However, in the eight or so papers and two textbooks we examined that discussed Bayesian classifiers, all of them used the word "probability" to describe terms like those above. Some even did so while using a little p – which signifies a density function – in their notation. In an effort be both expedient and conscious of the terminology, we are going to *consciously* accept this convention. We will continue to speak of probabilities, but will use a small p instead of a big P to remind both ourselves and the reader that in most cases these are in truth density functions.

Definition 1.2.1. A **Bayesian Classifier** f_b is a function from the space of possible examples to the space of possible classes of the form

$$f_b(E) = \text{argmax}_c \; p(c|E).$$

In a binary classification problem we can give a simpler definition of a Bayesian classifier, which is equivalent.

Definition 1.2.2. A **binary Bayesian classifier** is a function f_b from the example space to the space of classes $\{+, -\}$ of the form:

$$f_b(E) = \begin{cases} + & \text{if } \frac{p(+|E)}{p(-|E)} \geq 1, \\ - & \text{if } \frac{p(+|E)}{p(-|E)} < 1. \end{cases}$$

This definition is equivalent to the more general one above because $\frac{p(+|E)}{p(-|E)} \geq 1$ if and only if $p(+|E) \geq p(-|E)$; it will be used in a theorem later on. For now, we will focus on the first definition. By Bayes' Rule, the function in Definition 1.2.1 can be rewritten as

$$f_b(E) = \text{argmax}_c \; \frac{p(E|c)p(c)}{p(E)}.$$

We can deal with each of these three terms individually.

With that said, $p(c)$ is the probability of the class c, sometimes referred to as the **prior probability** or **prior** of c. This term is not problematic; it can be estimated from data or simply given to the classifier from an external source. A common technique is simply to treat the prior for a class as that class's frequency in the dataset – so if in a dataset with 200 examples, 150 were of class +, we might assume $p(+) = .75$.

$p(E)$ is the probability of the example E. Intuitively, we should think of this as the probability that the example E occurs in any class at all. $p(E)$ could be calculated by marginalizing the class value – that is, using the Law of Total Probability to compute $p(E|c)$ for each c, and adding the sum of these probabilities weighted by each $p(c)$. However, this computation is unnecessary for the purposes of the Bayesian classifier, since $p(E)$ itself

is independent of the class. $p(E)$ will be the same for every class examined, so it will not affect the value of c returned by f_b – so we can ignore it.

$p(E|c)$ is the term that is most difficult to compute in a Bayesian classifier, and it is this difficulty that makes Bayesian classification interesting. The differences between the various Bayesian classifiers are essentially differences over how to compute $p(E|c)$. The most direct method, which produces a classifier referred to as the **full Bayesian classifier**, is to attempt to compute the probability via the multiplication rule of conditional probability.

The definition of conditional probability states that for events $X_1, X_2, ..., X_n$,

$$p(X_1|X_2X_3...X_n) = \frac{p(X_1X_2...X_n)}{p(X_2X_3...X_n)}.$$

Rearranging this gives us

$$p(X_1X_2...X_n) = p(X_1|X_2X_3...X_n)p(X_2X_3...X_n).$$

Observe that the same identity can then be applied to $p(X_2X_3...X_n)$, and so on, until the equation becomes

$$p(X_1X_2...X_n) = \prod_{i=1}^{n} p(X_i| \cap_{k=i+1}^{n} X_k)),$$

where the last few terms of this product will be $p(X_{n-1}|X_n)$ and $p(X_n)$.

Recall that an example E consists of a tuple of attribute values $(x_1, x_2, ..., x_n)$. These values can be treated as instances of random variables in an unknown multivariate probability distribution; or, to be more precise, each class value c has its own probability distribution for each of the attribute variables. This means that each specific value an attribute takes is a probabilistic event. Coming back to the above product, if we let X_n refer to the value of the class, c, of an example, and all the other X_i-s refer to the event, "attribute X_i takes the value x_i", then the product becomes

$$p(x_1x_2...x_nc) = p(x_1|x_2...x_nc)p(x_2|x_3...x_nc), ..., p(x_{n-1}|x_nc)p(x_n|c)p(c).$$

Dividing by $p(c)$ yields

$$\frac{p(x_1 x_2 ... x_n c)}{p(c)} = p(x_1|x_2 ... x_n c)p(x_2|x_3 ... x_n c), ..., p(x_{n-1}|x_n c)p(x_n|c) = p(E|c).$$

We have rewritten $p(E|c)$, but it is still in a very complicated form. We can deal with the last term, $p(x_n|c)$, without difficulty, but the others require care.

Any probability of the form $p(x_n|c)$ is referred to as the **likelihood** of the value x_n given c. (To reiterate, we are here treating X_n as a random variable that has, in the example E, taken the value x_n.) If X_n can take on a continuous range of values, the most common way to compute $p(x_n|c)$ would be to assume that values of the attribute X_n in the class c are normally distributed, and then compute the likelihood of this particular value of x_n. If X_n takes a discrete range of values, then $p(x_n|c)$ is best calculated as the frequency with which $X_n = x_n$ in the class c.

Computing the second-simplest term, $p(x_{n-1}|x_n, c)$, is where things get difficult. Because our data are only samples of the true distributions of attributes, we cannot compute an expression like $p(x_{n-1}|x_n, c)$ without making simplifying assumptions about the relationship between the variables. The Naive Bayes classifier, which we discuss in detail in the next section, assumes the attributes are independent. In this paper, we will make the extremely simple assumption that we can model the dependence of one attribute variable on others linearly. This will take a moment to explain fully; keep in mind that our overall goal is to be able to compute the value of expressions like $p(x_{n-1}|x_n, c)$.

Imagine, just for a moment, that we had a function f that takes as input values of X_{n-1} and X_n, and returns the value X_{n-1} would have had if it were independent of the value of X_n. That is, if we assume that the value of X_{n-1} is partially determined by the value of X_n, this function f returns the value of the part of X_{n-1} that is not determined by X_{n-1}. If we had f, we could express $p(x_{n-1}|x_n, c)$ as $p(f(x_{n-1})|c)$, since $f(x_{n-1})$ and x_n are independent.

That said, without some assumptions about the way X_{n-1} and X_n relate, this hypothetical function is really fairly silly. What if X_{n-1} and X_n have some fantastical, extraordinarily complicated, chaotic relationship, and it really doesn't make sense to ask what one would be without the other? In the general case, a function like f is impossible. However, we need not work in the general case. We will *assume* that the relationship between our attributes is strictly linear. That is, if we have an example, $E = (x_a, x_b)$, and we want to evaluate $p(x_a|x_b, c)$, we will assume that the x_a we observe in the example is the sum of some independent part of x_a, which we will call $x_a^{independent}$, and the value of x_b, possibly multiplied by some constant ρ_b. That is, $x_a^{dependent} = x_a^{independent} + \rho_b x_b$.

Now, to translate this model into practice, we need to figure out what ρ_b should be. After we do that, we can find the function f described above with just a little algebra. We propose that ρ should be the covariance between $x_a^{dependent}$ and x_b. (Recall that for two random variables X and Y, the covariance between them, $\text{cov}(X, Y) = E[XY] - E[X]E[Y]$.) There are three motivations for this choice. First, we want ρ to be something that is easily computable. Ultimately, if we are to implement these classifiers on computers, we want them to be as fast as possible, and covariances are relatively easy to compute. Second, we want our model to be easily understandable – to be transparent – and clearly capture some sort of relationship between x_a and x_b. The covariance accomplishes this. And third, if we assume that x_b is taken from a standard normal distribution, this choice of ρ_b is algebraically consistent. If

$$x_a^{dependent} = x_a^{independent} + \rho_b x_b,$$

then

$$cov(x_a^{dependent}, x_b) = E[x_a^{dependent} x_b] - E[x_a^{dependent}]E[x_b]$$

Since x_b is from the standard normal distribution, $E[x_b] = 0$, and the equation simplifies to

$$cov(x_a^{dependent}, x_b) = E[x_a^{dependent} x_b].$$

Now, $x_a^{dependent} x_b = x_a^{independent} x_b + \rho_b x_b^2$. Expected values can be split across sums, so we simplify to

$$cov(x_a^{dependent}, x_b) = E[x_a^{independent} x_b] + E[\rho_b x_b^2].$$

Since $x_a^{independent}$ and x_b are independent by assumption, $E[x_a^{independent} x_b] = E[x_a^{independent}]E[x_b] = 0$. So we are left with

$$cov(x_a^{dependent}, x_b) = E[\rho_b x_b^2] = \rho_b E[x_b^2].$$

Again, since x_b is from the standard normal distribution, $E[x_b^2] = 1$ and we are left with $cov(x_a^{dependent}, x_b) = \rho_b$. This little bit of algebra shows that if we assume $\rho_b = cov(x_a^{dependent}, x_b)$, our model won't automatically be self-contradictory (or at least not in this way). However, notice that x_b coming from a standard normal distribution was critical to this result. Since this is not usually the case, we need to normalize x_b every time it appears in our model, in order to force this condition. So

$$x_a^{dependent} = x_a^{independent} + cov(x_a^{dependent}, x_b) x_b$$

becomes

$$x_a^{dependent} = x_a^{independent} + cov\left(x_a^{dependent}, \frac{x_b - \mu_b}{\sigma_b}\right) \frac{x_b - \mu_b}{\sigma_b}$$

, where μ_b is the mean of x_b and σ_b is the standard deviation of x_b.

With that all done, we now can define $f\colon (x_a^{dependent}, x_b) \longrightarrow x_a^{independent}$ as

$$f(x_a^{dependent}, x_b) = x_a^{dependent} - cov\left(x_a^{dependent}, \frac{x_b - \mu_b}{\sigma_b}\right) \frac{x_b - \mu_b}{\sigma_b}.$$

This simplification generalizes: when we want to evaluate expressions like $p(x_a|x_b, x_c, c)$, we can extend the model to include x_c as just another term in the sum:

$$x_a^{dependent} = x_a^{independent} + cov\left(x_a^{dependent}, \frac{x_b - \mu_b}{\sigma_b}\right) \frac{x_b - \mu_b}{\sigma_b} + cov\left(x_a^{dependent}, \frac{x_c - \mu_c}{\sigma_c}\right) \frac{x_c - \mu_c}{\sigma_c},$$

and so on. This allows us to quickly compute expressions of the form $p(x_{n-1}|x_n, c)$.

We chose this approach for calculating these probabilities because it is simple, not because it is the most accurate. Assuming that the relationships between attributes are simply linear sums is probably incorrect, and relying on a covariance matrix generated from sample data, particularly small amounts of sample data (recall that when conditioning on the class, we restrict the amount of data we look at), is unwise, since sample covariance matrices are not robust. Modern implementations of full Bayesian classifiers use techniques like Principal Component Analysis or Factor Analysis to generate a sense of which attributes are best modeled this way, or whether more complicated models should be used. However, this approach will work for the purpose of building a fast, easy-to-understand tool for evaluating these complicated probabilities

1.3 Naive Bayes

Notice that the complications in the full Bayesian classifier arise from potential dependencies among the attributes of the example. The probability computations can be dramatically simplified by assuming that these dependencies do not exist – that is, by assuming that the attributes are independent of each other. The classifier making this assumption is referred to as the **Naive Bayesian classifier**; the name comes from the classifier naively assuming that the attributes are not conspiring together behind its back.

Definition 1.3.1. The **Naive Bayesian classifier** f_{nb} is a function from the example space to the class space of the form $f_{nb} = \text{argmax}_c\, p(c|E)$, where the constituent attributes of E are assumed to be independent.

As with the full Bayesian classifier, we ignore the denominator of the fraction that $p(c|E)$ yields and assume $p(c)$ is known or easily accessed. The remaining term, $p(E|c)$ is

simplified by the assumption of independence as

$$p(E|c) = p(x_1 x_2 ... x_n|c) = \prod_{i=1}^{n} p(x_i|c).$$

Each of the terms of this product is easily computed; indeed, the entire algorithm is easy to implement and much faster both than other Bayesian algorithms and non-Bayesian classification algorithms: since it requires only one loop over the training data, a well-written Naive Bayes classifier takes $\mathbf{O}(dn)$ time to construct and $\mathbf{O}(1)$ time to evaluate a single example, where d is the number of examples in the training data, and n is the number of attributes.

Naive Bayes is a powerful classification algorithm that comes close to matching the performance of far more complicated modern algorithms in empirical tests [3]. At first glance, this performance is somewhat puzzling, as the algorithm's assumption of independence among attributes is rarely realistic. Simple tests on highly dependent data-sets reveal that the presence of dependence alone is not necessarily a problem for Naive Bayes. So while the independence of attributes is a sufficient condition for good performance for Naive Bayes, it is not a necessary condition. Explaining Naive Bayes' performance, and characterizing specific conditions under which it performs well – finding the minimal necessary condition on the data – has been an on-and-off problem in machine learning theory for some time.

1.4 Bayesian Classifiers as Graphs

So far I have presented a view of classifiers that mostly engages with the mathematical underpinnings of what a Bayesian classifier does. In practice, however, it is very common for people to adapt the basics of what we have already discussed to create classifiers designed to deal with a specific problem. The easiest way to describe the way such classifiers work is by examining the topology of the graphs they define.

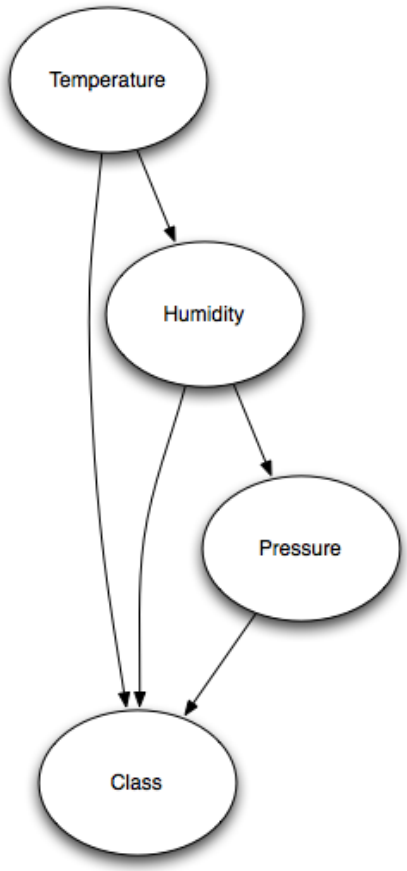
Definition 1.4.1. A **Bayesian graph or Bayesian network** is a graph representing the model used by the classifier to examine an example. It obeys the following two rules:

1. It is an acyclic digraph, usually with multiple sources and exactly one sink. This sink represents the probability that an example is of a given class.

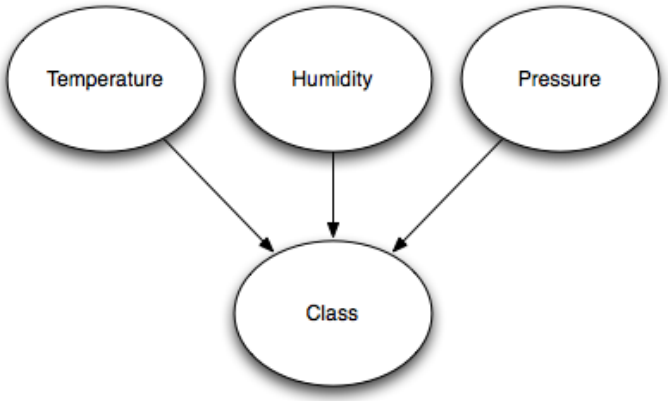
2. Each vertex represents the probability of some event occurring; edges indicate that one event is dependent on another. If an edge goes from vertex A to vertex B, this indicates that the probability of B is affected by the probability of A.

Multiple algorithms for constructing Bayesian networks exist, and they are not limited to classification problems. The algorithm we will use is simple: we simply place down one vertex for each attribute and one for the class value. We create one edge connecting each attribute vertex to the class value, and one edge connecting each vertex to all other vertices that are not independent of it. This is a "bare-bones" Bayesian network – more complicated networks have layers of vertices between the source vertices (the attributes) and the sink (the class), which can represent the relationship between different events with sophistication. However, as such network grow, they can quickly become somewhat opaque to human examiners – they acquire the "black box" characteristic of neural networks, where it becomes difficult to understand what the network is doing internally.

For completeness, we will give a brief example of a Bayesian network. Imagine that we were trying to build a classifier that would predict whether or not it would rain in New York City, given a description of the weather conditions the previous day. Our attributes could be the various measurements one can make about the weather: temperature, humidity, and pressure. When run on the data, a full Bayesian classifier would generate a graph like the one on the left below, while a Naive classifier would generate a graph like that on the right.

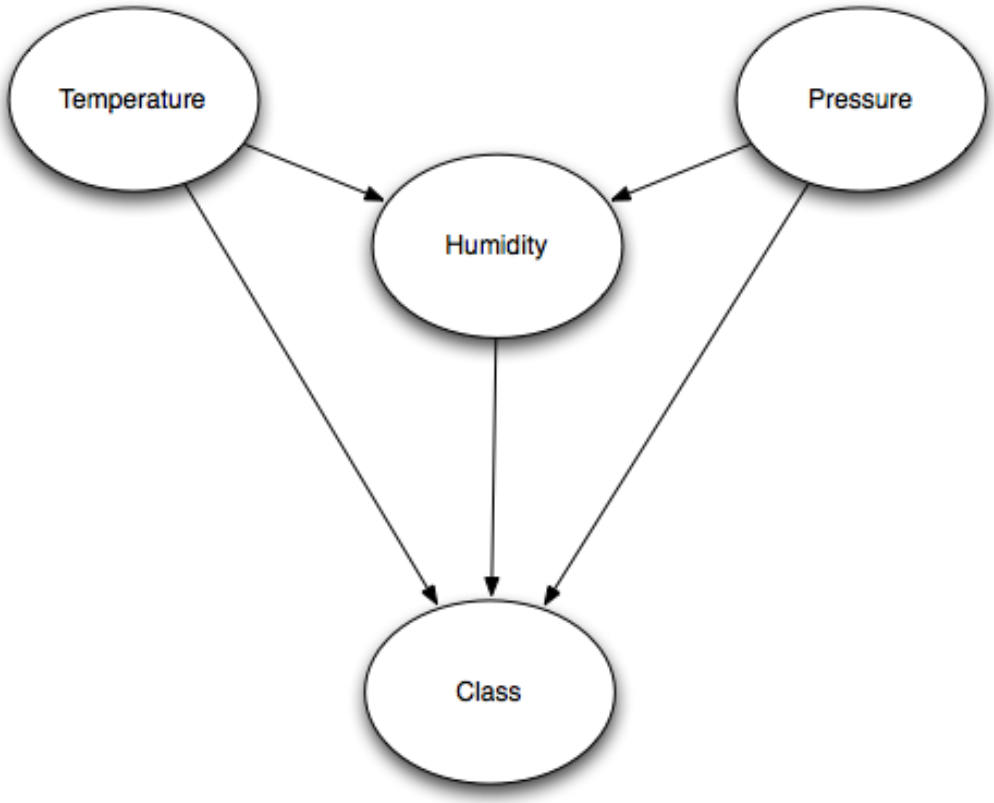


Full Bayes



Naive Bayes

Since we have a fairly good idea of the relationship between these variables, however, we can dramatically improve the performance of the classifier by altering the structure of its graph to match that of reality. Pretend for a moment that humidity is influenced by temperature and pressure, neither of which influence each other, and that all three influence whether or not it will rain. A Bayesian graph reflecting this model of the weather would look like this.



Because this graph reflects an accurate model of the system under consideration, its classifier will have all manner of advantages over the other two we have seen. It will perform better, need less data to become reliable, and assimilate additional data faster and more effectively than the other two. In applied uses of Bayesian graphs, any information at all about the structure of the relationship between attributes is immensely helpful.

Unfortunately, in many situations we do not have as intricate an *a priori* understanding of the relationship between our variables as we do with the weather. That is why general algorithms such as full and Naive Bayes are used: they can be deployed on any data, at any time, and do a relatively good job. Still, researchers are always searching for ways to improve the performance of generalized algorithms, and that is ultimately what this project is all about. By the end of the next chapter, we will have mathematically defined a way in which we can create hybrid classifiers that combine the assumptions of full and

Naive Bayes to improve performance; in a way, however, these hybrids are best viewed in terms of network structure. We will elaborate on this idea in Section 2.2.

1.5 Dependence derivatives on Naive Bayes

In 2004, Harry Zhang of the University of New Brunswick published a paper titled "The Optimality of Naive Bayes", which dramatically improved on existing conditions for Naive Bayes' optimality. Zhang defined the concept of a **dependence-derivative ratio** on an example, and showed that, when this ratio is close to 1, the dependence between attributes is not greatly detrimental to the performance of Naive Bayes. (Zhang did this using standard and reasonably intuitive definitions for terms like "good" and "performance", which we will specify later.)

As described, Bayesian classifiers determine which class is most likely for a given example; the classes are each amorphous multivariate probability distributions, the example is a tiny spot on each of them, and the classifier determines which spot is most likely. When thinking of dependence-derivatives, however, it is better to visualize the classifier as a generalized see-saw, where the classes are the ends and the example is a weight placed near the fulcrum. The relative likelihood of each of the attribute values pushes the weight a little to the left or the right, and in the end the sum of these determines the way the entire apparatus falls. This metaphor illustrates that even small differences in the way algorithms estimate the likelihood of attributes can greatly impact classification, and it also makes it easy to think of particular values of the examples attributes as "pushing" the classifiers' estimate in the direction of one class or another. If, for a particular example, all but one attributes contain little distinguishing information (that is, none particularly moves the weight away from the fulcrum), a single attribute strongly associated with a class can "push" the classifier a great deal in the direction of that class.

With this metaphor in mind, consider the difference between how the naive and full Bayesian classifiers interpret attribute values. To the naive classifier, each attribute value stands alone, and should be evaluated only in its own context; so a value strongly indicating a particular class will always push the classifier in the direction of that class. The full Bayesian classifier is more sophisticated: it also determines if, given the other attribute values that come in the example, a particular attribute value should be expected to point in one direction or another simply because of its dependence on the other attributes. A value strongly indicating a particular class only "pushes" the full Bayesian classifier if it cannot be explained away by dependence between attributes.

The key observation here is that the full Bayesian classifier at most considers attribute values as having same impact as Naive Bayes does; the impact of an attribute value in the full classifier can never exceed the impact in Naive Bayes. Imagine now how this makes a difference when classification is calculated in terms of the see-saw metaphor. Naive Bayes will see an interesting attribute value and immediately push the weight towards the class value the attribute indicates. The full Bayesian classifier will push the weight, but then stop, reconsider, and move the weight back towards where it was in proportion to the extent the attribute value was explained by its dependence on other attributes. The dependence derivative ratio for an attribute and an example is, essentially, the distance between the full Bayesian classifier and the Naive classifier at the end of this process. (In fact, in Zhang's paper, he defines the ratio arbitrarily and then proves it captures this idea.) This distance captures the extent to which the dependence between attributes is affecting Naive Bayes' estimate of that attribute value.

The true utility of this idea becomes clear when we think of all the attributes pushing the classifiers, rather than a single one. Each attribute pushes the naive and full Bayesian classifiers, and, each time, the distance between them changes according to the extent to which that attribute is dependent on the others. But sometimes the distance between

the classifiers is increased, and sometimes it is decreased! In the end, the total effect of dependence on Naive Bayes is the total distance between the naive and full classifiers. Notice that since the classifiers are only sometimes pushed apart, and sometimes pushed together, that dependence is not always detrimental to Naive Bayes' performance. It is only when dependence of one attribute on others is not balanced out by further dependence of other attributes in the other direction that Naive Bayes' performance degrades. The **global dependence factor**, the product of the **dependence-derivative ratios** for each attribute of an example, captures the extent to which classification is affected by dependence.

Definition 1.5.1. For an attribute x_i of an example E, the **dependence derivative** of a class c, $dd_c(x_i, E)$ is

$$dd_c(x_i, E) = \frac{p(x_i|x_1x_2..., x_{i-1}x_{i+1}...x_nc)}{p(x_i|c)}.$$

That is, a dependence derivative is the probability of an attribute value given the attributes parents and the class divided by the probability of the value given just the class.

Dependence-derivative ratios are defined in terms of two classes, as follows.

Definition 1.5.2. The **dependence-derivative ratio** for an attribute x_i between classes $+, -$ is the dependence derivative of one class divided by the dependence derivative of the other. It is denoted $ddr(x_i, E)$ and given as

$$ddr(x, E) = \frac{dd_+(x_i, E)}{dd_-(x_i, E)}.$$

Finally, we define a global dependence factor, which is the product of the dependence-derivative ratios for each attribute.

Definition 1.5.3. The **global dependence factor** of an example E with n attributes is the combined product of the dependence-derivative ratio of an example. It it is given as

$$gdf(E) = \prod_{i=1}^{n} ddr(x_i, E).$$

Notice several things immediately:

1. If a given x_i is independent of other attributes, then $p(x_i|x_1x_2...x_{i-1}x_{i+1}...x_nc) = p(x_i|c)$, so the dependence derivatives of both the positive and negative classes will be 1, so the overall ratio will also be 1.

2. If $ddr(x_i, E) = 1$, then Naive Bayes and the full Bayesian classifier will be pushed by x_i the same amount in their calculations, and x_i will have no effect on the final distance between full and Naive Bayes.

3. If $gdf(E) = 1$, then either $ddr(x_i, E) = 1$ for all x_i, or $ddr(x_i, E) > 1$ for some x_i and $ddr(x_j, E) < 1$ for others. This is the "balancing" effect we discussed earlier.

4. If $gdf(E) = 1$, then Naive Bayes and the full Bayesian classifier will have identical performance on the example. This is a corollary of the following theorem.

The centerpiece of Zhang's paper is a theorem [3], adapted below, which, in words, says that dependence derivatives tell us everything we need to know about the difference between Naive Bayes and a full Bayesian classifier. To prove it we briefly introduce the idea of a condition function.

Definition 1.5.4. Let f be a binary Bayesian classifier. Then, from definition 1.2.2, f returns + if some other function $C(E) \geq 1$, and − if $C(E) < 1$ for some $C\colon E \longrightarrow [0, 1]$. C is the condition function of f.

The condition function of Naive Bayes is

$$C_{nb} = \frac{p(+)\prod_{i=1}^{n} p(x_i|+)}{p(-)\prod_{i=1}^{n} p(x_i|-)},$$

and the condition function of full Bayes is

$$C_b = \frac{p(+)}{p(-)} \prod_{i=1}^{n} \frac{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n, +)}{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n, -)}.$$

Zhang's theorem states that the two functions differ only by $gdf(E)$.

Theorem 1.5.5. *Let f_b be a full Bayesian classifier in a binary classification problem, let f_{nb} be its corresponding Naive Bayes classifier, and let E be an example, $E = (x_1, x_2, ..., x_n)$. Let C_b be the decision function of f_b and let C_{nb} be the decision function of f_{nb}. Then*

$$f_b(E) = f_{nb}(E)gdf(E).$$

Proof. From definition 1.2.2 and 1.5.4,

$$C_b(x_1, x_2, ..., x_n) = \frac{p(+)}{p(-)} \prod_{i=1}^{n} \frac{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, +)}{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, -)}$$

$$= \frac{p(+)}{p(-)} \prod_{i=1}^{n} \frac{p(x_i|+)}{p(x_i|-)} \prod_{i=1}^{n} \frac{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, +)p(x_i|-)}{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, -)p(x_i|+)}$$

Observe the first two terms are just the definition of $C_{nb}(E)$:

$$C_b(E) = C_{nb}(E) \prod_{i=1}^{n} \frac{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, +)p(x_i|-)}{p(x_i|x_1, x_2, ..., x_{i-1}, x_{i+1}, ...x_n, -)p(x_i|+)}$$

$$= C_{nb}(E) \prod_{i=1}^{n} \frac{dd_+(x_i, E)}{dd_-(x_i, E)} = C_{nb}(E) \prod_{i=1}^{n} ddr(x, E) = (C_{nb}(E))(gdf(E)).$$

□

The output of a binary classifier is controlled by the output of its decision function. Zhang's theorem proves that the decision functions of full Bayes and Naive Bayes differ only by a product of $gdf(E)$ – hence why the last bullet point above is a corollary of this theorem. But that isn't all. Zhang further proved that, under a distance measure known as **zero-one loss**, the distance between f_b and f_{nb} decreases as $gdf(E)$ approaches 1. [3] Zero-one loss is not a complicated metric: for a given set of data, the zero-one loss distance between two classifiers is simply the number of examples for which they predict different classes. By proving that this metric decreases as $gdf(E)$ approaches 1, Zhang provided us with a way to modify Naive Bayes to bring its performance closer to that of a full Bayes network.

2
Between Naive and Full Bayes

2.1 Evaluating Classifiers

In this section we will define a series of modifications to Naive Bayes, with the aim of using Zhang's dependence derivatives to improve its performance. Before we do so, however, we need some terminology with which to evaluate classifiers.

Data in the real world is extraordinarily messy, and for this reason it is unrealistic to expect a classifier to accurately predict the class of every object in its training set (let alone the test set). Therefore, when discussing classifiers, we steer clear of terminology that reflects a sense of perfectibility in classifier performance. The particular word in question we have in mind is "optimal". Strictly speaking, the "optimal" classifier is the one that performs best on test data. Because a classifier does not encounter the test data until it is, well, tested, the optimal classifier is therefore the one that strikes the best balance between optimizing performance on the training data and avoiding the problem of over-training, in which the model implicit in the classifier becomes overly fitted to the training data and loses its effectiveness on the test data. Notice, however, that this point of optimal training varies with the degree to which the training data truly represents the test data. A

classifier that fits itself to the training set closely will perform well on well-represented test data and badly on badly-represented test data. A classifier that does not fit the training data particularly closely will be less affected by a decrease in the quality of the test data's representation, but will similarly benefit less from an increase in the quality of training data. As a consequence of this fact, it is common when comparing classifiers to assume that the test data and training data are drawn from the same source – typically one sets aside a small portion of the data as for testing, and trains on the rest – and that this partitioning has been randomized and repeated a number of times (a procedure called folded cross-validation). This is essentially as close to truly representative training data as can be constructed; however this method of comparison has as a downside that it does not give a researcher an iron-clad assessment of the optimal classifier in other situations. It is practical enough, however, as in many common uses of classifiers, this same method of folded cross-validation can be used "in the field".

With that extended aside, we will now briefly define some terminology for assessing classifier performance. In the aftermath of a test run, all classifiers will produce an error matrix, which details exactly how many objects of each class were classified as each – so a two-class problem with classes positive and negative will yield entries for false positives (FP), false negatives (FN), true positives (TP), and true negatives (TN). The error matrix looks like

$$\begin{pmatrix} \text{True Positive, False Positive} \\ \text{False Negative, True Negative} \end{pmatrix}.$$

To jump-start the reader's intuition about this matrix, we provide the following off-the-cuff assertions about how it relates to the performance of classifiers.

1. The classifier is performing perfectly if and only if the matrix is diagonal.

2. If the classifier decides everything in the test data is in the positive class, the matrix will only be non-zero in the first row.

3. If the classifier decides everything in the test case is in the negative class, the matrix will only be non-zero in the second row.

4. If the test case contains only examples from the positive class, the matrix will only be non-zero in the first column.

5. If the test case contains only examples from the negative class, the matrix will only be non-zero in the second column.

With that out of the way, we define the following statistics on the classifier in these terms:

1. The **accuracy** $A = \frac{TP+TN}{TP+TN+FP+FN}$. Accuracy is not a particularly popular measure for assessing classifiers, as it is heavily influenced by the distribution of the classes in the test data. (Imagine, for example, if we were testing for a disease that occurs in only 1% of the population. We could achieve 99% accuracy just by declaring everyone negative.)

2. The **precision** $P = \frac{TP}{TP+FP}$. (Observe that $TP+FP$ is simply the number of objects in the test set.) This is also sometimes called the **true positive rate**.

3. The **recall** or **sensitivity** $R = \frac{TP}{TP+FN}$. This is the rate at which the classifier gets instances of the positive class right.

4. The **specificity** $S = \frac{TN}{TN+FP}$. This is the rate at which examples labeled as positive really are positive.

These terms are set up in terms of the positive class; examining the same statistics in terms of the negative class requires only replacing TNs with TPs, and so on.

All of these terms are attempts to compress information about the 2×2 error matrix into a few scales from 0 to 1, and hence there are all sorts of algebraic relationships between them, most of which are not very helpful. To understand these statistics, think

of what they represent in terms of the classifier's performance. For example, specificity and sensitivity are counterparts; a classifier is said to be sensitive if it correctly classifies most objects in the positive class, while it is specific if it only calls positive those objects that are in fact positive. An ideal classifier would have high values for both sensitivity and specificity; however, in some situations it is beneficial to emphasize one over the other.

Imagine the following scenario: after an industrial accident, a very large number of people are suddenly at risk for developing an extraordinarily rare, but potentially life-threatening disease. The presence of the disease can be easily determined with a simple blood test, but so many people are at risk that it is not possible to test everyone. However, scientists soon determine that people with certain medical characteristics are much more likely to develop this disease than others. A binary classifier is employed to find people whose medical records indicate they might be one of these high-risk patients. In this situation, the cost of a false positive (declaring someone is high risk when they are not) is far less than a false negative (declaring someone is low risk when they are not); therefore, a high-sensitivity classifier would be much more useful than a high-specificity or balanced classifier.

Modifying Bayesian classifiers to produce higher specificity or sensitivity is easy: we simply vary the threshold over which the data must cross to fall into one class or another. In the above example, we want to bias the classifier towards the positive class and away from the negative class. To achieve this we take the standard equation for a Bayesian classifier,

$$f_b(E) = \text{argmax}_c \; p(c|E),$$

and add another term, b_c, which is 0 for the negative class and, say .25 for the positive class. The new formula is:

$$f_b(E) = \text{argmax}_c \; (\; p(c|E) + b_c).$$

This classifier will skew towards the positive class, resulting in higher rates of true positives and false positives, and lower rates of true and false negatives. Glancing at the formulas for sensitivity and specificity will show that this will indeed increase sensitivity at the cost of specificity.

This example shows how neither sensitivity, specificity or precision completely characterize a classifier's performance. However, people have found it convenient to try to compress these statistics into a single number, the **F-score** or **F1** of a classifier.

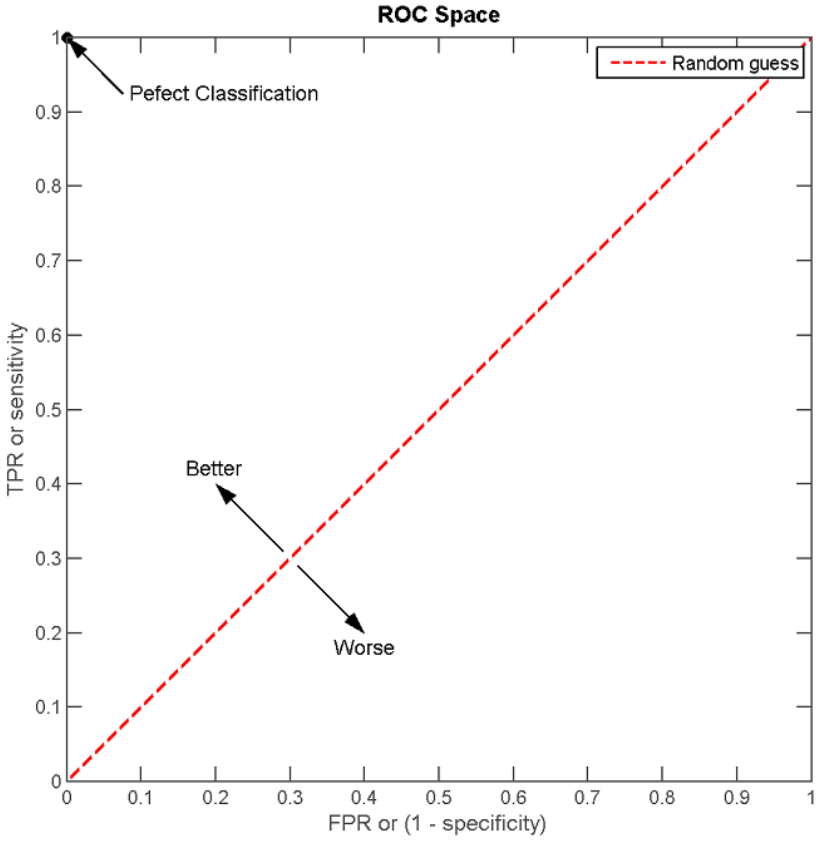
Definition 2.1.1. For a classifier with recall R and precision P, the F-score is defined as

$$F1 = 2\frac{RP}{R+P}.$$

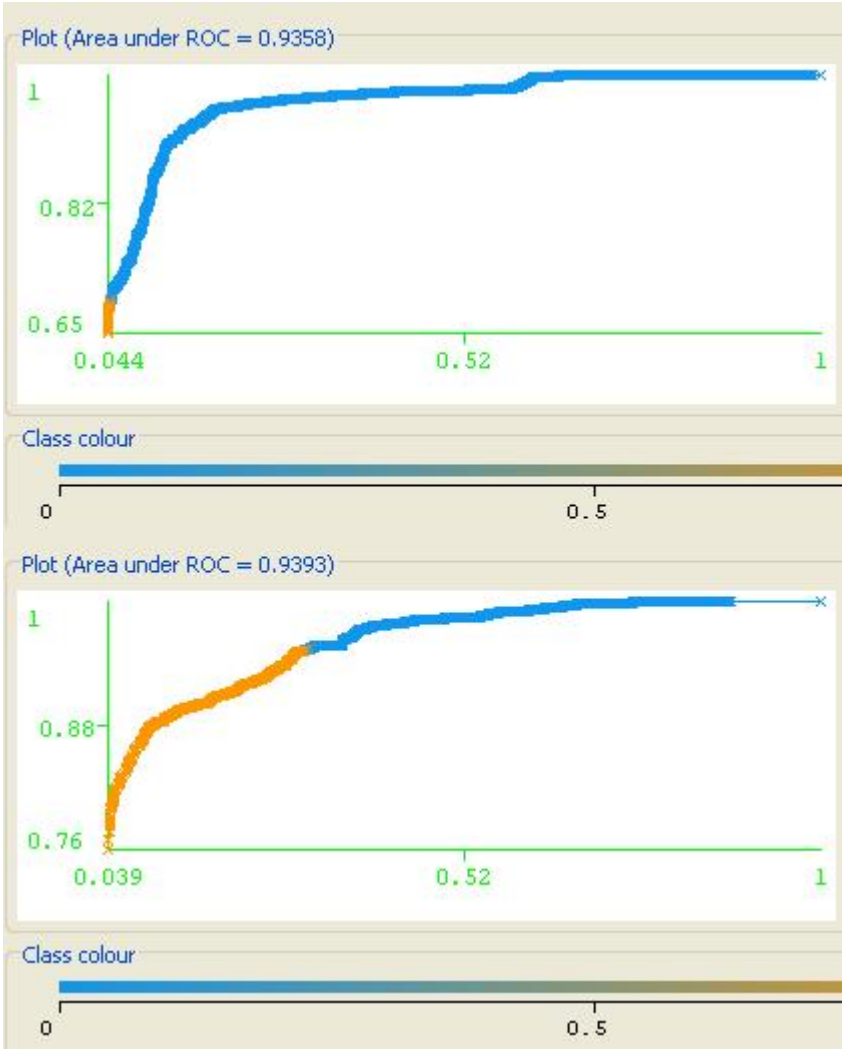
The F-score, like the statistics it is built from, is between 0 and 1. (Observe that if $0 \leq R, P \leq 1$, $0 \leq 2RP \leq 2$ and $0 \leq R + P \leq 2$, so $0 \leq 2\frac{RP}{R+P} \leq 1$.) It gives a reasonable first-glance estimate of how well a classifier is performing. The weakness of the F-score, however, is that it does not tell us anything about how the classifier performs at other decision thresholds – that is, if we modify a classifier to increase recall at the cost of precision, the F-score is likely to change. For example, imagine a classifier with a recall of .7 and a precision of .8. The classifier's F-score is $\frac{2*.56}{1.5} = .746$. If we bias the classifier in the direction of the positive class, it might gain .1 recall but lose .1 precision. It would then have an F-score of $\frac{2*.54}{1.5} = .72$. This is not a trivial problem – there is no particular reason why this bias term should be held at 0 all the time, and the F-score can change a reasonable amount as the bias changes. So, for completeness's sake, it behooves us to define a term that captures the overall performance of a classifier as the bias is changed.

For that, we have to turn to the receiver operator characteristic curve (ROC curve). The **ROC curve** is formed by plotting a classifier's false-positive rate on the horizontal axis and sensitivity on the vertical axis as the decision threshold is varied. In this plot, a perfect classifier is on the upper left at $(0, 1)$. A classifier that randomly guesses classes at

their frequency in the test set will move along the line $y = x$ from $(0, 0)$ to $(1, 1)$ – that is, if the test case contains 80 positives and 20 negatives, a classifier that guesses positive 80% of the time will move along this line. This line of "frequency guessing" is important because it is in a sense the worst a classifier can do – if a classifier ever performs below this line, we can improve performance by inverting the classification. The image below, from Wikipedia [6], represents this visually.



Below are two ROC curves from the same text classification dataset [4]. The first is for the positive class, the second is for the negative class. From the better shape of the first curve and the higher overall area under the curve, we can tell that the classifier is better at dealing with positive datapoints than negatives.



Ideally, we would like to examine an ROC curve by hand, to make note of places where the curve changes dramatically or where the classifier performs best. However, for quick, at-a-glance comparisons, one can usually just look at the area under the ROC curve.

We mention ROC curves not because they are going to be used extensively in the project, but because we believe they give the reader a more complete understanding of the problems involved in understanding the performance of classifiers. Ultimately, all these metrics try to compress a $N \times N$ error matrix – where N is the number of classes in the problem –

into something more easily "glanced at" by humans. This cannot be done without some loss of information – a fact we should keep in mind.

2.2 Modifying Naive Bayes' algorithm

With the ability to assess classifier performance, a sense of how Bayesian classifiers fit into the general scheme of statistical estimators, and Zhang's dependence-derivative ratios, we are finally in a position to propose and evaluate modifications of the classic Bayesian classifiers. The general idea is to use dependence-derivatives to find efficient ways of creating a classifier that contains elements of both the naive and full algorithms. By now, it should be clear that the naive and full classifiers are opposite ends of a scale from minimally fitting the data to fitting the data as closely as possible. We will create algorithms that will occupy the interior of this scale. To do that, we first have to define a measure of where on this scale a classifier lies.

We see two ways in which we can define this scale; both involve setting aside a set of attributes to be treated differently from others. In the strong version, we assume that these attributes are independent of the others; in the weaker version, we assume they are merely unaffected by them (and are free to affect the others).

Definition 2.2.1. Let f be a Bayesian classifier operating on a dataset with n attributes. The **strong naivete** of the classifier is the number of attributes that the classifier assumes are independent of the rest.

Definition 2.2.2. Let f be a Bayesian classifier operating on a dataset with n attributes. The **weak naivete** of the classifier is the number of attributes whose vertices in the digraph of the classifier are sources. This is **not** the number of attributes that are assumed to be independent of the rest. Our assumption is weaker than that: we allow for there to be

dependence between our attributes, but impose a condition on the direction the classifier assumes influence is moving.

The naivete of any classifier with n attributes varies from 0 to n. Appropriately, the Naive Bayesian classifier has naivete n, while the full Bayesian net has naivete 0. Notice that in either extreme case, there is no difference between the strong and weak versions of naivete. However, simply stating that a classifier has a certain naivete value does not fully determine it, since we do not know which attributes are naive.

Zhang's dependence derivatives provide a way out of this ambiguity. We are, in effect, dividing the classifier's attributes into one set that is to be treated as in the Naive algorithm, and another than we will treat as in the full network. We know that the Naive classifier works best on attributes with a global dependence factor close to 1. Therefore, for a k-naive classifier, we can select the k attributes for which their dependence factor is closest to 1.

There is a problem, however. Zhang's dependence-derivative functions are defined in terms of a single data point – recall that the global dependence factor is defined as

$$gdf(E) = \prod_{i=1}^{n} ddr(x_i, E).$$

One way to get around this problem is to recalculate which attributes to use for each data point. This would be extraordinarily computationally intensive, but the resultant classifier would be closest to the ideal of a meld between the Naive and full Bayesian classifiers.

In addition to its computation impracticality, this approach has another flaw: because it is constantly using the dependence-derivative ratios of the attributes at certain values, it is effectively constantly accessing the covariance matrix of the data. This is undesirable, because the covariance matrix of sample data is biased and heavily susceptible to outliers. In other words, the sample covariance matrix needs a large amount of data before it becomes reliably close to approximating the real covariance matrix. To reduce the effect this

will have on our algorithm, we can average the dependence-derivatives for each attribute, and use these averages as the terms in a modified global dependence factor we call the "weight" of a set of attributes.

Definition 2.2.3. If $S = \{x_1, x_2, ...x_k\}$ is a set of k attributes in a binary classification problem with D examples (data points), let the **weight** of S, $w\colon S \longrightarrow \mathbb{R}$, be defined as

$$w(S) = \prod_{x_i \in S} \sum_{d=1}^{D} \frac{ddr(x_i, E_d)}{D}$$

This is the product of the average dependence derivative ratio for each attribute. If it is close to 1, then Zhang's theorem implies that a Naive Bayesian classifier examining only these attributes will perform similarly to a full Bayesian classifier if run on this data.

This weight function immediately gives us an idea of which combinations of attributes are interesting subsets: those that have the smallest weight of subsets of the same size.

Definition 2.2.4. Let $X = \{X_1, X_2, ..., X_n\}$ be the set of all attributes in a classification problem, and let S be a subset of X with size k. We say that S is **maximally balanced** if $w(S)$ is smallest for S when considered against all other subsets of X with size k.

The notion of weight also gives us an idea of which of strongly or weakly Naive classifiers will perform better in ideal conditions. In the strong case, we assume that for an example E, an attribute x_s is independent of the others. This means that for all $x_i \in E$, $p(x_s x_i) = p(x_s)p(x_i)$. This immediately implies that $ddr(x_s, E) = 1$. It also, however, implies that for any x_i,

$$p(x_i|x_1, ..., x_{i-1}, x_{i+1}, ..., x_s, ..., x_n, c) = p(x_i|x_1, ..., x_{i-1}, x_{i+1}, ..., x_{s-1}, x_{s+1}, ..., x_n, c).$$

Since that term is used in the calculation of x_i's dependence derivative, we can see that assuming an attribute is strongly naive also has an effect on the dependence derivatives of the other attributes. Weak naivete does not have this problem, and so it is preferable to strong naivete.

Finding the maximally balanced subset of size k is potentially a computationally difficult task. The most obvious way go about it is to evaluate the average dependence derivative ratio of each attribute, and then find the subset of attributes for which the product of these weights is closest to 1. Since in a set with n elements, there are $\binom{n}{k}$ subsets of size k, the number of subsets to consider can vary wildly with k, and can become quite large. If we wish to find the maximally balanced set over all possible k, then we have 2^n subsets to consider, since our search space is the power set of the n attributes. Since it is not uncommon to have 30 to 40 attributes in real-world classification problems, this is a serious problem.

That said, we will not try to find a more efficient method for finding the maximally balanced subset of attributes. This is because this computational problem, finding the subset of attributes for which the product of its elements' dependence-derivative ratios is closet to 1, is equivalent to a classic NP-complete problem, which computer scientists have been attempting to efficiently solve for decades: the subset-sum problem. The subset-sum problem is stated as follows: given a set S of integers, determine if the elements of any non-empty subset of S sum to 0.

Theorem 2.2.5. *Let A be a set of rational numbers. If an algorithm can determine if a subset of A multiply to 1, then there is a set of rational numbers B for which that algorithm can solve the subset-sum problem.*

Proof. We need a one-to-one map $f : \mathbb{Q} \to \mathbb{Q}$ such that $f(1) = 0$ and $f(ab) = f(a) + f(b)$. $f(x) = \ln(x)$ has these properties. If an algorithm can find a subset $S \subseteq A$ such that $\prod_{s_i \in S} s_i = 1$, it has determined if the set $B = \{\ln(a) | a \in A\}$ has a subset that sums to 0, since $\sum_{s_i \in S} \ln(s_i) = 0$. □

I mention this only as an aside; in any case, we can now talk about a whole range of classifiers that use these maximally-balanced subsets to bridge the gap between the Naive and full Bayesian classifiers.

Definition 2.2.6. Suppose f is a Bayesian classifier operating on a dataset with n attributes and naivete k. The classifier is said to be the **maximally-balanced classifier of naivete** k if the k naive attributes are the maximally-balanced attribute set of size k.

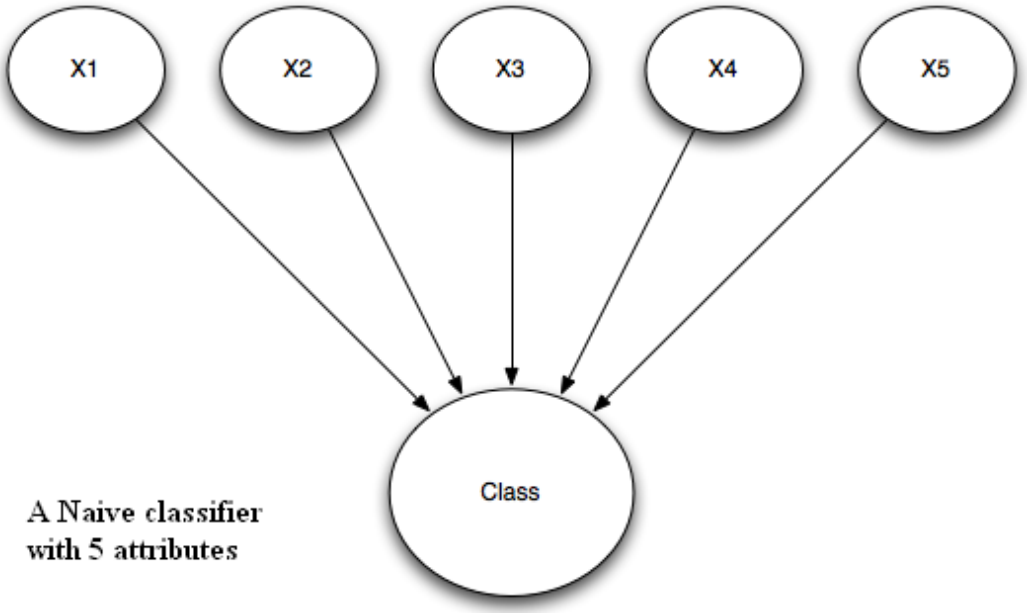
Putting it all together, we can now detail the algorithm used to create k-maximally balanced classifiers of either strong or weak naivete. For a binary classification problem involving D points of training data and n attributes in the set $X = \{X_1, X_2, ..., X_n\}$, the k-maximally balanced classifier can be found in the following way.

1. For each attribute, determine the average of the dependence derivative ratios over all data points.

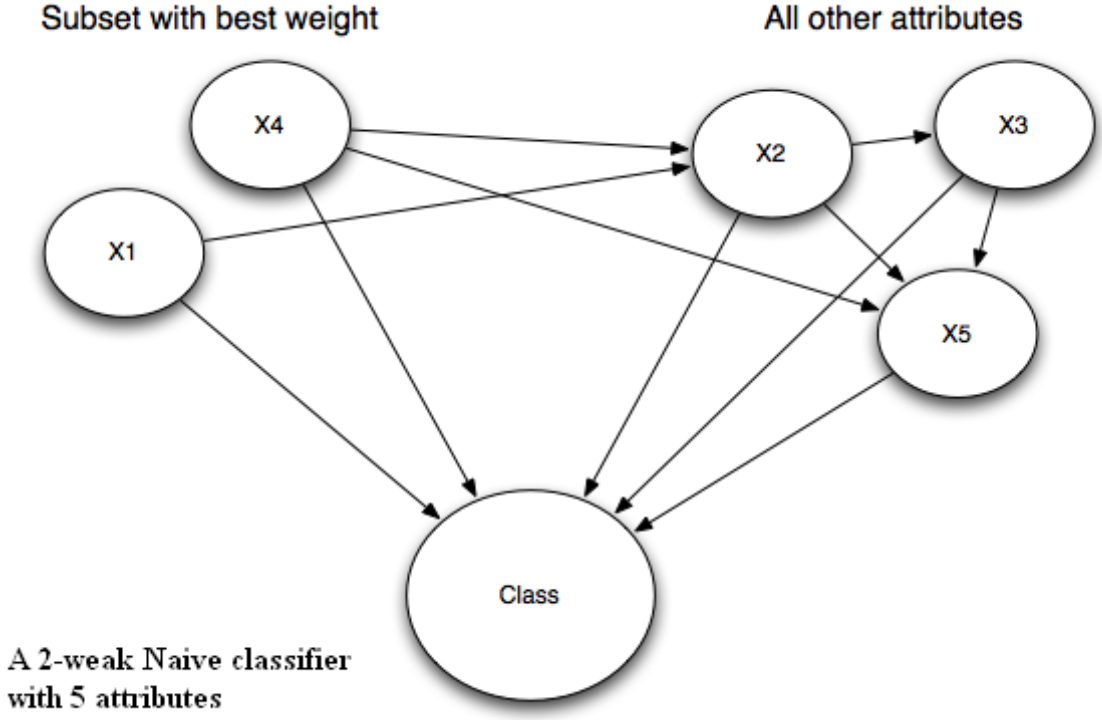
2. Using these averages, find the set of attributes S for which $w(S)$ is minimized.

3. Compute f_b using the additional assumption that the attributes S are independent (in the strong naivete case) or unaffected by the elements of $X - S$ (in the weak naivete case).

We can also look at this algorithm in terms of the structure of the graph associated with the classifier. Each classifier starts with something like this, where each attribute is connected with the class and nothing else.



Our algorithm detects which combination of attributes has the smallest weight, and leaves those as independent from all others (in the strong naive case) or as independent from each other (in the weak naive case). All other attributes are joined together.



We now have a firm understanding of how to create k-Naive classifiers and why they should work. Over the winter break and the second semester we wrote Python programs that implement these algorithms as described. In the next chapter we will describe the programs and display the results of a program of tests we ran to verify the claims made above.

3
Testing K-Naive Classifiers

3.1 Creating k-Naive Classifiers

In this section we explain the structure of the implementation of the classifiers in this project. However, because a detailed understanding of the implementation of the programs is not necessary to understand the results, we will be brief. Final copies of all the code written for the project are included in the appendix.

The classifiers are implemented in an object-oriented framework using numpy, a scientific computing package for Python. A super-class, Bayes Classifier, contains a set of default methods for initializing a classifier, storing the training data and statistics relevant to it (such as means, variances and covariances), and for performing a few basic calculations such as evaluating the likelihood of a value in the standard normal distribution.

Three classes inherit from Bayes Classifier: NaiveBayes, FullBayes, and KStrongNaiveBayes (the last class also implements k-weak Naive Bayes). Each class performs additional initialization functions, such as calculating an optimal naivete, and provides its own methods for evaluating an example. The classes also provide support for both standard and logarithmic calculations of probabilities – a few months in, it became clear that logarithmic

calculations were necessary to avoid machine-precision errors. As we noted in the previous chapter, Naive Bayes and a full Bayesian network can be created by using k-Naive Bayes classifiers for extreme values of k. We wrote these algorithms their own classes to make debugging easier.

A fourth class, TestHarness, controls the classifiers and runs tests on them. On initialization, it creates instance of a classifier as an instance variable and repeatedly calls that classifier's evaluate function to get data. TestHarness is the user-oriented class of the project, and can run cross-validated tests, test on a given set of test data, return values for a classifier's sensitivity, specificity, F-score, and so on. All the charts below were created by a script that repeatedly calls TestHarness called Tester and a script called Data that generates artificial datasets.

3.2 Attribute Balance

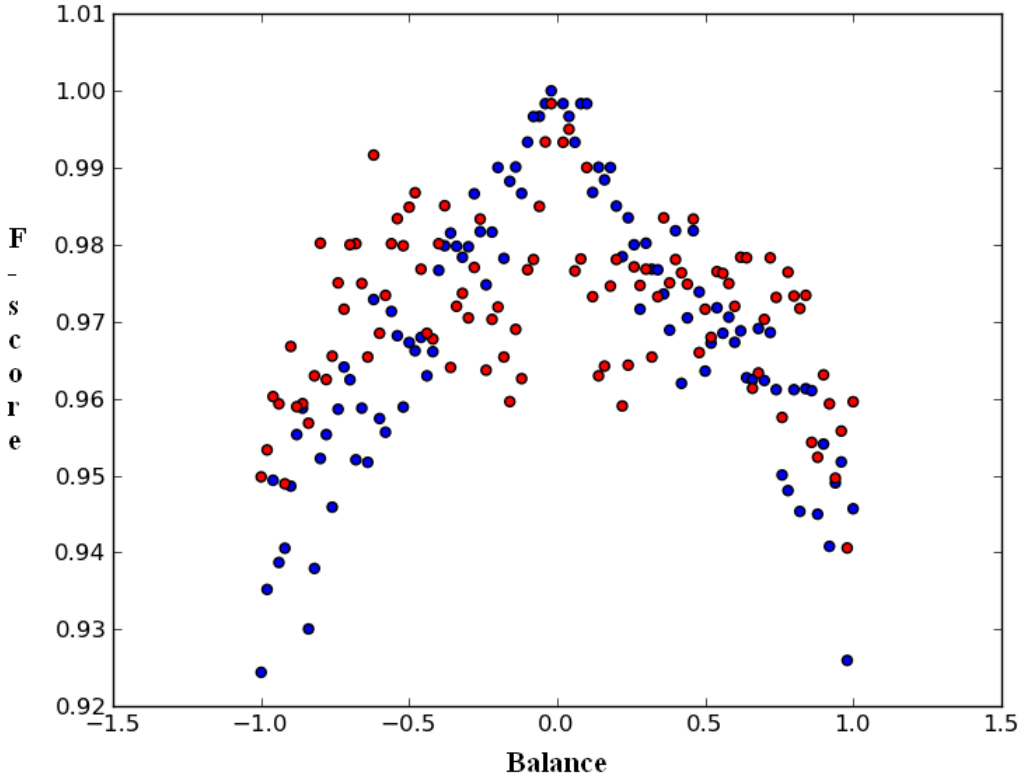
In this chapter, we empirically verify the claims we have made about Bayesian classifiers, with an eye towards demonstrating the potential of k-maximally-balanced classifiers over the standard Naive Bayes. Our default strategy for testing a claim will be to create an artificial dataset with an interesting property, such as a high level of covariance among attributes, and observe the change in classifier performance as we vary this property. If the observed changes in classifier performance are consistent with what we would expect given the discussion of Bayesian classifiers in the previous chapter, that will be evidence that we got things right.

The first parameter to examine this way is attribute balance. **Balance** is a blanket term we use to describe the sum of the weight of all attributes in the training set; because the weight of an attribute can vary widely across data sets, balance does not denote exactly the sum of the weight, but rather roughly the direction and magnitude of the total weight. A balance of -1 indicates that there is a large amount of covariance in the direction of

the negative class; a balance of 1 indicates a large amount of covariance in the direction of the positive class. A balance of 0 indicates little covariance in either direction.

Unless otherwise specified, these scatter-plots have between 50 and 100 points per classifier represented. The simulated data contain random elements; to reduce the impact of random variations, each point on the scatter-plot represents the output of a 10-fold cross-validated test run on a training set of between 150 and 300 4-attribute examples generated using the parameter on the X-axis. This means each part of the X-axis corresponds with a unique set of training data. The Y-axis of the point is usually the F-score of the classifier; the X-axis can vary. For easy comparison, multiple classifiers are often trained on the same data set for a particular point. This means that if, for example, three classifiers are represented on a graph, each X-value should have exactly three points above it (so, if you moved a vertical line from the left hand of the graph to the right, you would always hit points three at a time).

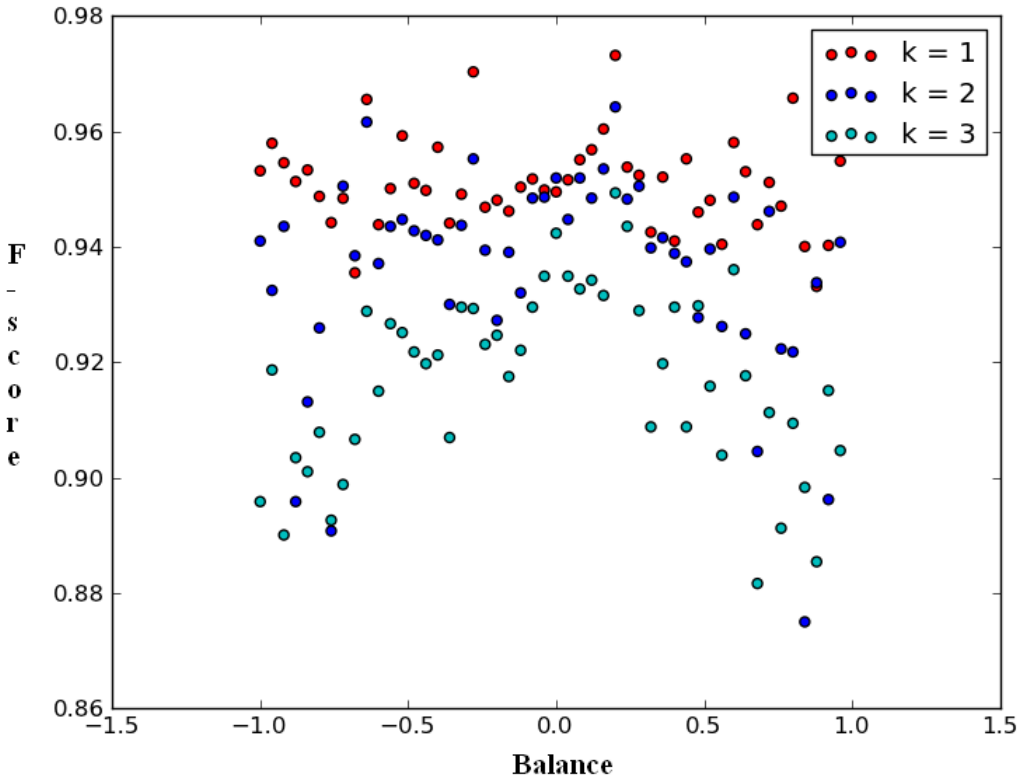
Here is an easy example of one of these charts: empirical evidence that Naive Bayes responds as described to attribute covariance. The chart below shows the performance, as measured by F-score of a full Bayesian classifier (red) and a Naive Bayesian classifier (blue).The X-axis is the balance used to create the data set at that particular point. We expect Naive Bayes to do well when the absolute value of the balance is low, and perform badly when it is high. The results are shown below.



This scatter-plot matches our expectations well. Notice that while the full Bayesian network is relatively unaffected by changes in the balance, Naive Bayes actually performs better than full Bayes under near-zero balance conditions. This is expected; when the data are in balance, Naive Bayes is effectively makes fewer assumptions about the data than full Bayes.

Note that the actual value of the Y-axis displayed in these charts is mostly irrelevant, as it is primarily a product of the relative difficulty of that particular classification problem. Some data sets have nice, clearly-defined classes with few outliers to confuse the algorithms; others are messy. Because we include both sorts of problems in these scatter-plots, readers should mostly be interested in the *relative* performance of the different classifiers as the independent variable changes, rather than what the actual F-score is.

In the previous chapter, we argued that a optimally-balanced k-naive classifier would be able to deal with unbalanced attribute covariance better than a Naive classifier. This is a claim that merits empirical testing. In the previous section we showed that, as expected, Naive Bayes performs very well when attribute covariance is balanced and not as well when it is not. The chart below is run on similar data to the first chart, but with 1- (red), 2- (blue), and 3- (cyan) weak Naive Bayes as the classifiers. As the naivete of the classifier increases (that is, as the color goes from red to blue to cyan), we expect to see it change from something like the red points in the image above to a shape similar to the blue points above.



Notice the rough upside-down V of the cyan (3-Naive) color, which is similar the Naive (blue) classifier in the previous chart; this matches our expectations well. Particularly observant readers will notice that the scale of Y-axis is different in the second image than

in the first; this is because, unfortunately, the data used to generate these two charts are not identical – the exact data that generated the first chart was lost. Nonetheless, we can still draw conclusions from this pair of charts. First, the optimal level of naivete for most parts of these two data sets is 0. Only at one point in the first chart did Naive Bayes actually outperform full Bayes. This is intentional – this data set was designed so that there would be no good solution for the weight problem. Because of this, we expect performance to improve as naivete is reduced. This is indeed true of the second chart: in addition to exhibiting the expected upside-down V, we also see that the low-naivete classifier (in red) clearly outperformed the other two.

3.3 Attribute Weight

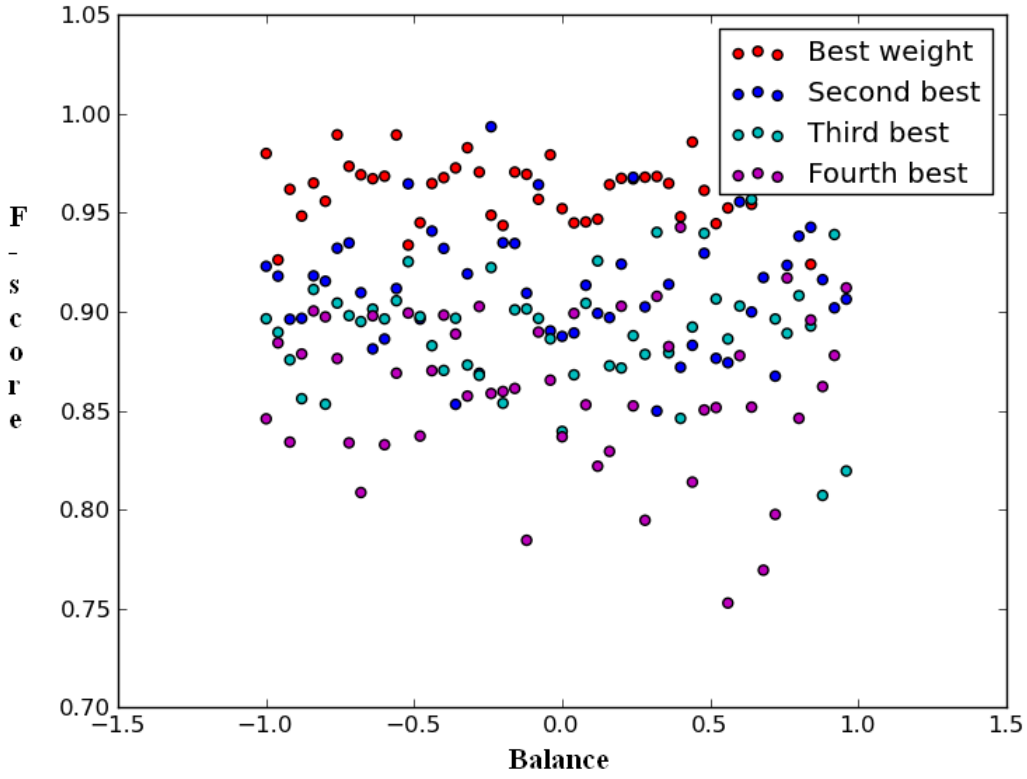
We have just presented evidence that the classifiers are indeed performing as we expect with respect to attribute balance. However, most of the claims we made in Chapter 2 used the much more well-defined idea of weight. Recall from Definition 2.2.3 that the weight of a set S of k attributes is

$$w(S) = \prod_{x_i \in S} \sum_{d=1}^{D} \frac{ddr(x_i, E_d)}{D},$$

where D is the number of examples in the data set.

Zhang's theorem shows that as the product of the dependence derivatives on an example moves towards 1, the performance of Naive Bayes on the example strictly improves. Since weight is the product of the average of dependence derivatives over an entire data set, we expect to see a performance improve as the weight moves toward 1. However because we are averaging, we do not expect a strict relationship between weight and performance – rather, *on average*, the k-Naive classifier constructed on the k attributes where weight is closest to 1 should outperform all other combinations of k attributes.

This next chart illustrates this; using the same data as the previous chart, where attribute balance is the independent variable, we created four versions of the 1-weak Naive classifier, and then forced each version to pick a particular attribute as its naive set. (There were four attributes in each data set.) Then, we sorted each collection of four classifiers based on the distance of their independent attribute's weight from 1, and grouped them together: all the classifiers with weight closest to 1 are represented below in red, the second closest to 1 are blue, third closest are cyan, and fourth closest are purple. We expect that overall, red should tend to have the best performance, followed by blue, cyan, and purple. We expect this to be true regardless of the balance on the x-axis.



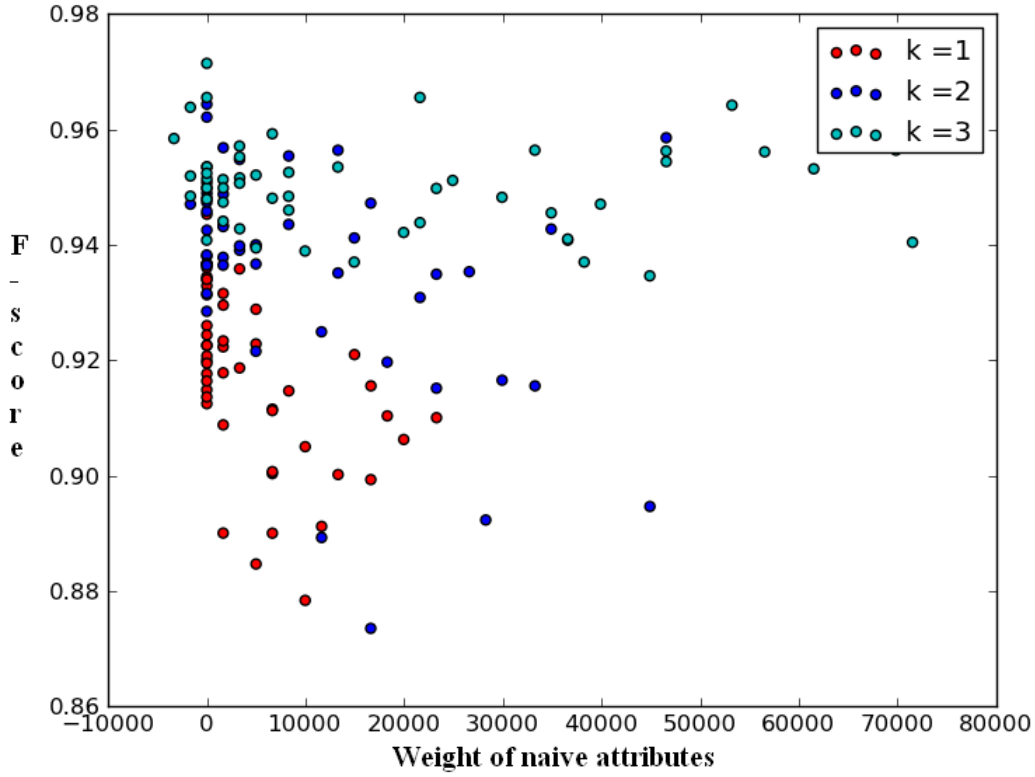
This matches well with our expectations: the average score of the red is clearly higher than that of the blue, which is higher than cyan, which is in turn higher than the purple. It is not true, however, that this ordering is strictly followed for every x-value on the chart: at

around $x = -.25$, for example, the second-best weighted classifier outperforms the best. This is because when we averaged dependence derivatives to create the weight function, we lost the precision of Zhang's theorem.

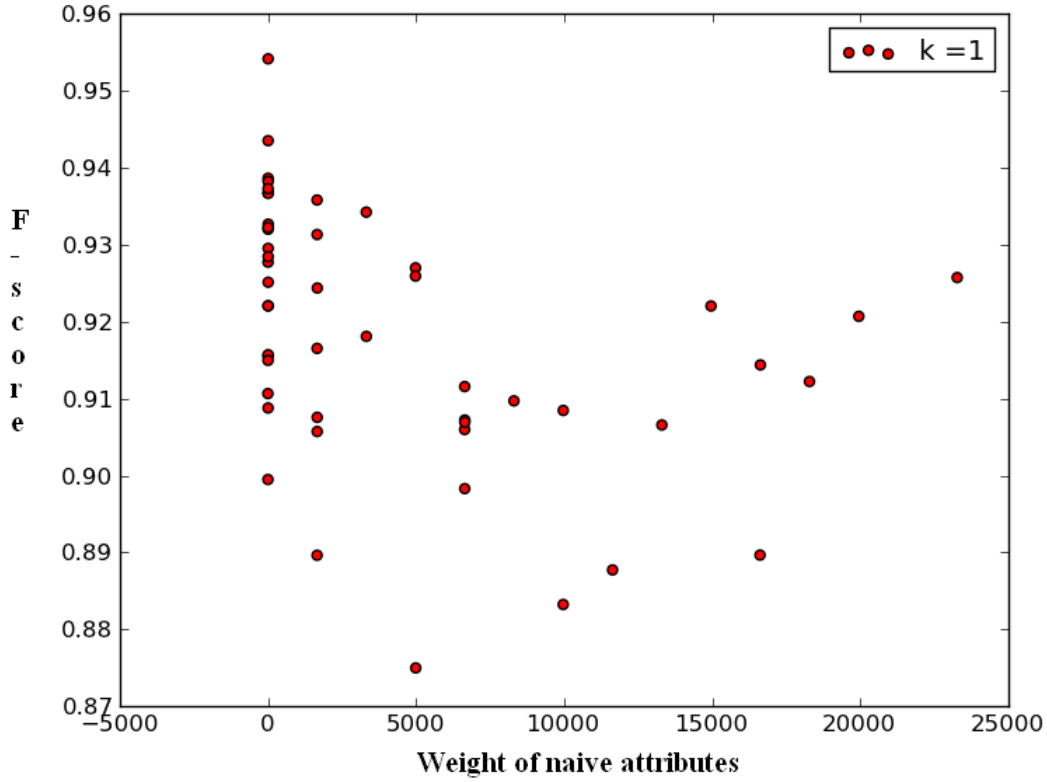
To be truly convincing about the relationship between weight and performance, we should examine a series of similar scatter-plots that at examine all 6 combinations of 2 attributes, and then all 4 combinations of 3 attributes. But these scatter-plots get cluttered very quickly, and we have other charts to display, so in the interest of brevity we will not.

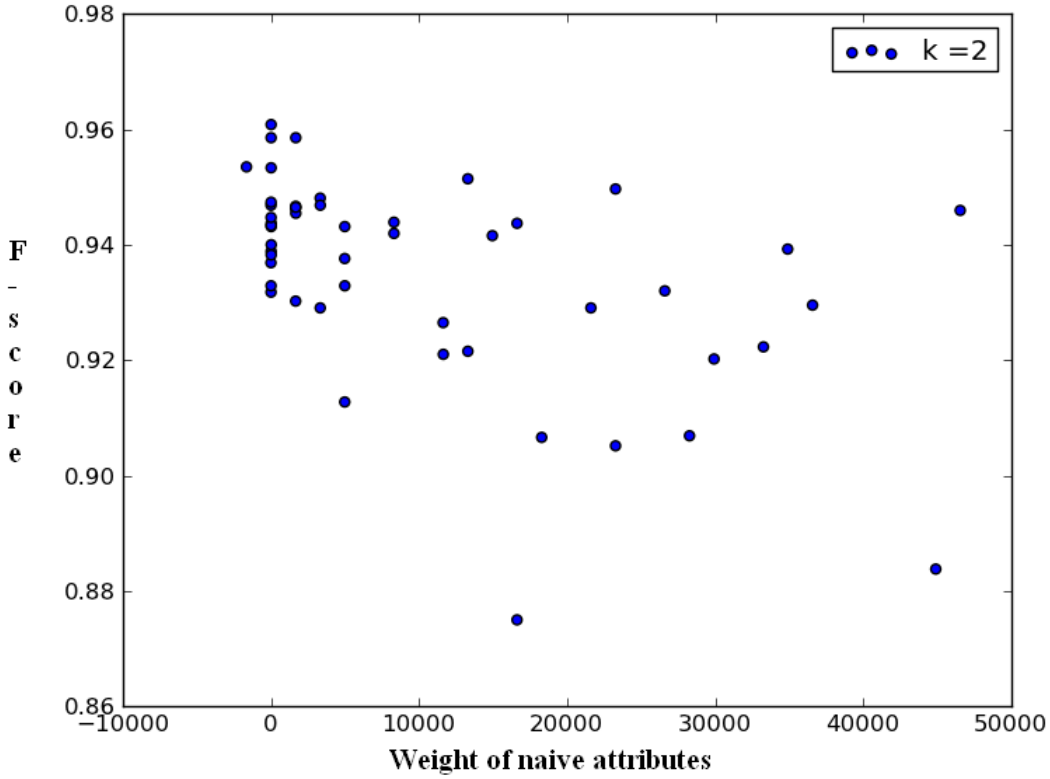
While we have just demonstrated that, in a particular data set, the collection of attributes with weight closest to 1 is usually the best set to use for a k-Naive classifier, is it in general true – across data sets – that better attribute weight corresponds with better performance? We have no particular reason to believe that this is the case; in the previous chapter, we only claimed that sets of attributes with weight closest to 1 would optimize performance for a particular data set. That does not at all imply that across different data sets – with completely different classes, completely different attribute ranges, and so on – that better attribute weight would correspond with better performance.

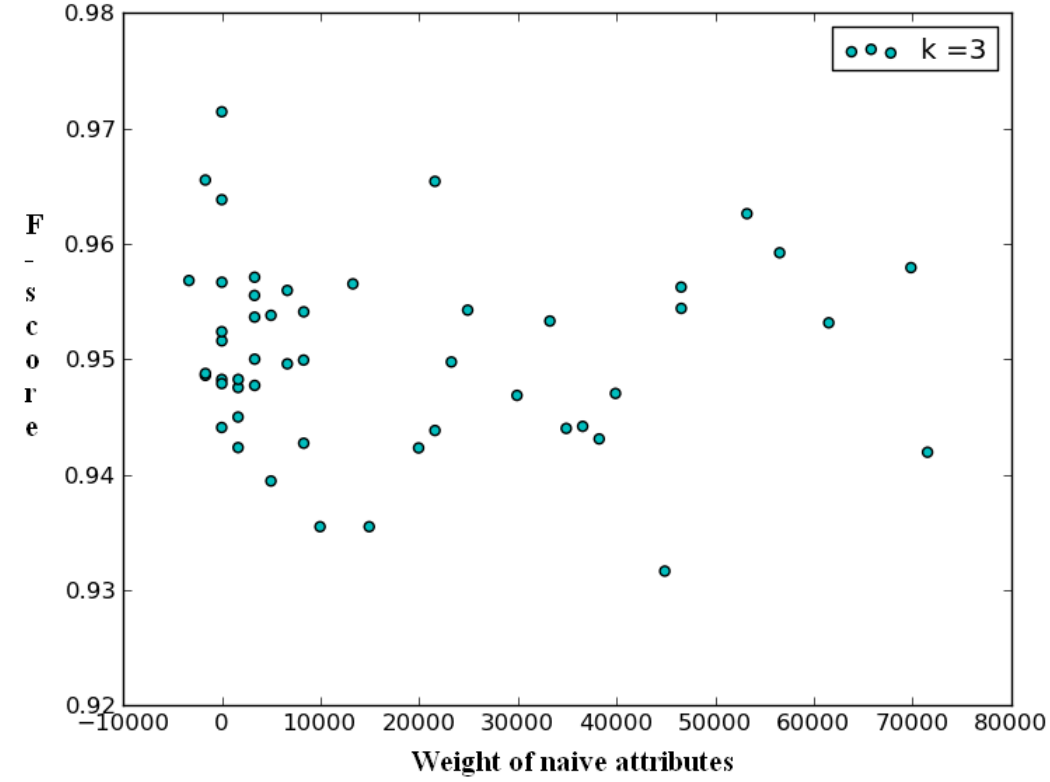
We do not make a habit of checking for patterns where there probably are none; nonetheless, this merits an examination, which the next few scatter-plots provide. On the X-axis we have the *log* of the weight of naive set, and on the Y-axis we have the performance of the classifier. The data being used here is drawn from a wide variety of simulated problems – the only restriction really all have four attributes each. We are still using the same color scheme for k-Naive classifiers (1 is red, 2 is blue, 3 is cyan).



There doesn't seem to be anything here. Let's examine each classifier on its own.



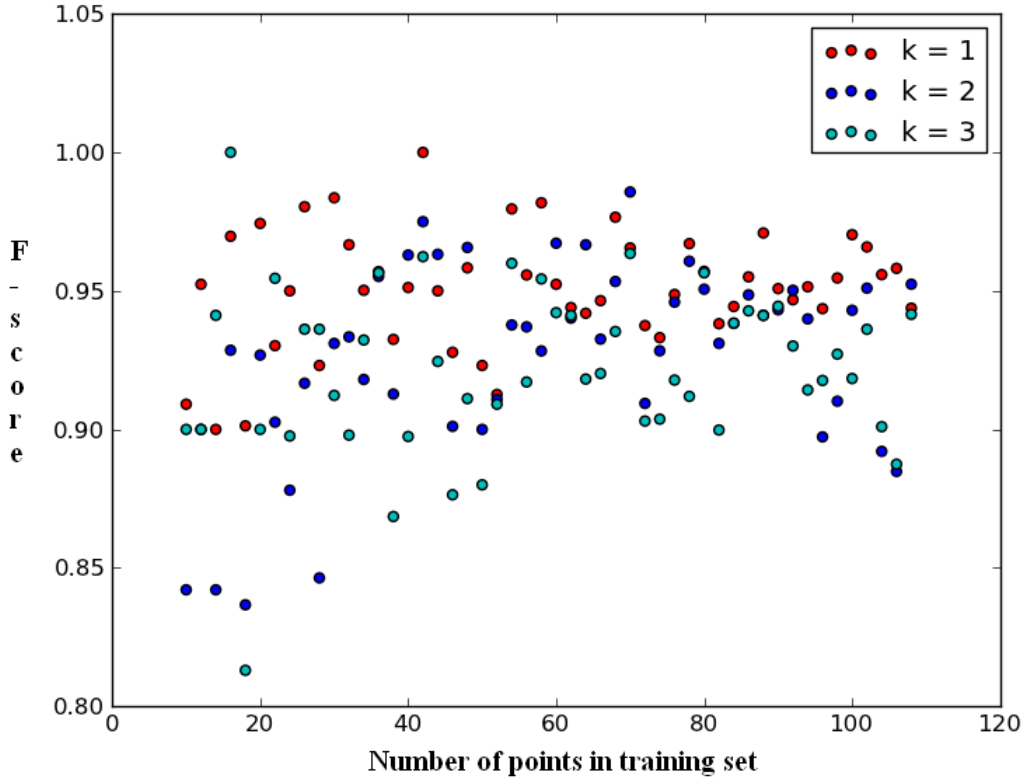

k =2
F-score
0.98
0.96
0.94
0.92
0.90
0.88
0.86
−10000
0
10000
20000
30000
40000
50000
Weight of naive attributes

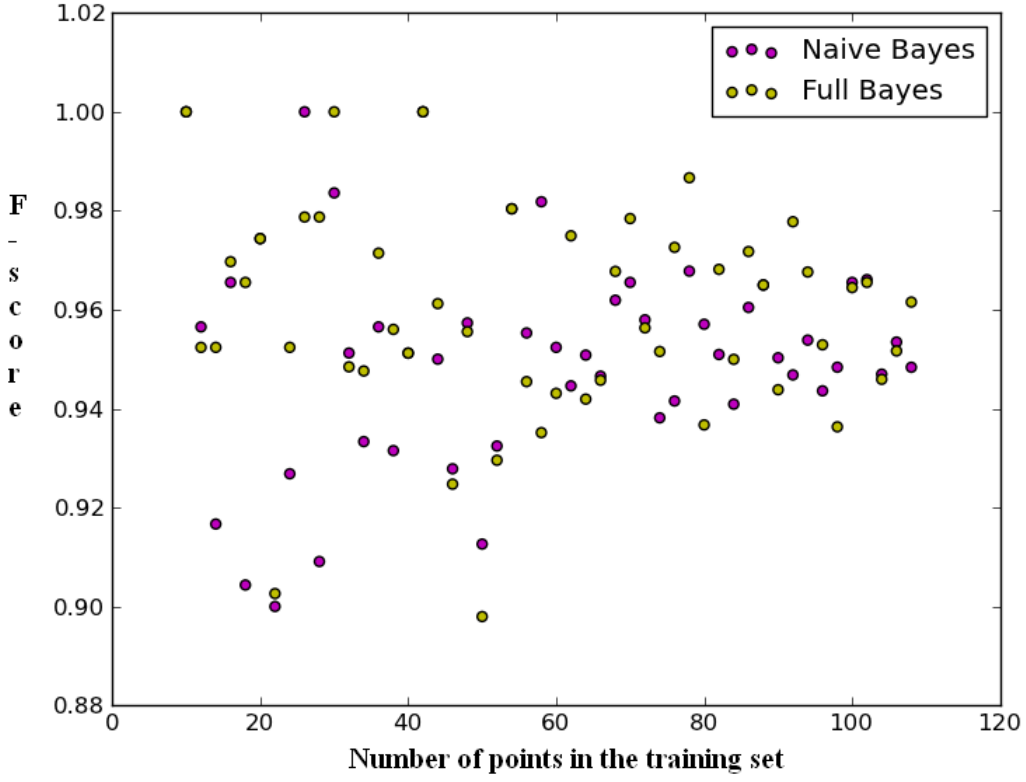

k =3
F-score
0.98
0.97
0.96
0.95
0.94
0.93
0.92
−10000
0
10000
20000
30000
40000
50000
60000
70000
80000
Weight of naive attributes

While there does not appear to be a strong relationship between attribute balance and classifier performance regardless of the data set used, we do note that in each case, the best classifiers were the ones running on data with weight very close to 1; and in each case, the worst performing classifiers had relatively large balances. There does appear to be a consistent signal here – it is just very weak. This may be due to luck or due to the set of classification problems used to generate this chart being unrepresentative; this is an area in which future work can be done.

3.4 Data Sparsity

The last question we examine in this chapter concerns the rate at which Bayesian classifiers improve their performance as the amount of data available increases. One of the greatest strengths of Naive Bayes is that it requires a minimal amount of data to begin performing relatively well. However, because k-Naive classifiers make extensive use of the sample covariance matrix, which is easily biased and requires lots of data to be reliable, we expect that they will not share this ability to withstand a dearth of data. The next scatter-plot is the same 4-attribute balance problem that we saw in Section 3.2, except instead of changing the balance along the X-axis, we hold balance to a constant -.5 and walk the number of examples in the training data from 10 to 110. As usual, the 1-naive classifier is in red, 2-naive is in blue, and 3-naive is in cyan. For comparison, we also include the same plot with full Bayes and Naive Bayes.


k = 1
k = 2
k = 3
F-score
Number of points in training set


Naive Bayes
Full Bayes
F-score
Number of points in the training set

The only noticeable pattern is the tendency for the classifiers' performance to become less variable as the amount of data increases, but this is expected of virtually any statistic. We believe that further testing will eventually reveal the extra reliance of k-Naive classifiers on an abundance of data.

3.5 Conclusion

As a final demonstration of k-Naive Bayesian classifiers, we downloaded a set of breast cancer diagnostic data that is sometimes used as a benchmark in machine learning [7]. It contains 569 examples describing the numerical characteristic of tumors, some of which are malignant, some of which are not. Our task is to separate the malignant tumors from the benign ones. The authors of the data set provided 10 main attributes, which we list here, giving the their explanation of how they were calculated when possible.

1. Radius (mean of distances from center to points on the perimeter)
2. Texture (standard deviation of gray-scale values)
3. Perimeter
4. Area
5. Smoothness (local variation in radius length)
6. Compactness (perimeter^2 / area -1)
7. Concavity (severity of concave portions on the contour)
8. Concave points (number of concave points on the contour)
9. Symmetry
10. Fractal dimension (using "coastline approximation" - 1)

We ran three classifiers on this dataset: Naive Bayes, full Bayes, and the optimally-balanced k-weak Naive Bayes for this set (that is, we allowed the classifier to examine all 2^10 possible combinations of attributes and pick the one with the best weight). Here are the error matrices of the classifiers.

Naive Bayes: $\begin{pmatrix} 342, 14 \\ 56, 148 \end{pmatrix}$ – full Bayes: $\begin{pmatrix} 232, 120 \\ 10, 198 \end{pmatrix}$ – k-weak Naive Bayes: $\begin{pmatrix} 333, 14 \\ 24, 185 \end{pmatrix}$

And their associated F-scores:

Naive Bayes: .878 – full Bayes: .772 – k-weak Naive Bayes: .919

The combination of attributes with the best weight were the radius, smoothness, and compactness. We have no idea why this is true, and we are unsure if there is any particular significance to these attributes being picked as opposed to the others. What we do know is that when these three attributes are held apart from the others, the classifier suddenly became much better at detecting malignancy in tumors. This shows that k-Naive classifiers have the potential to outperform full and Naive Bayes even beyond in the artificial sets of data we created for this chapter. In the future, we might consider what particular fields might have interdependent data that these classifiers can help out with. That, however, will have to wait for another project.

4
Appendix

The following sections contain Python programs for implementing K-Naive Bayesian classifies. The programs heavily rely on numpy, a scientific computing package for Python, and will not function without them. Numpy runs in Python 2.6, so it may be necessary for readers to roll back their Python distribution if they want to use this code. *Unfortunately, there are some places where the code is longer than a page width in LaTeX. I have introduced an artificial line break here, marked with a *BR*. Eliminate it, and the corresponding line break, with a regular expression if you want to run the code.*

4.1 BayesClassifier.py

```
#! usr/bin/env python
# Bayesian classifier template
# by Lionel Barrow

import math
from numpy import *

class Classifier():

    # Constructor
    def __init__(self, classlist, datamatrix, k = 0):
        self.classes = sort(classlist)
        self.data = array(datamatrix)
```

```
        self.data = sort(self.data, axis = 0)
        self.npoints = shape(self.data)[0]
        self.nattributes = shape(self.data)[1]-1
        self.nclasses = len(self.classes)
        self.classdata = self.classconditioned()

    # To be overwritten
    def buildessentials(self):
        print "A subclass has accessed a fake superclass method: buildessentials."

    # To be overwritten
    def evaluate(self, example):
        print "A subclass has accessed a fake superclass method: evaluate."

    # Often useful
    def classconditioned(self):
        conditioned = []
        minindex = 0
        maxindex = 0
        for c in range(self.nclasses):
            currentclass = self.classes[c]
            while maxindex < self.npoints and currentclass == self.data[maxindex,-1]:
                maxindex += 1
            tempmatrix = self.data[minindex:maxindex,0:-1]
            conditioned.append(tempmatrix)
            minindex = maxindex
        return conditioned

    # Returns the probability of a value for an attribute
    # in class c if the normal distribution is assumed.
    def normprob(self, value, attribute, c):
        mean = self.means[c][attribute]
        variance = self.variances[c][attribute]
        if variance == 0: variance = 10 ** -5
        p = (1/variance) * self.phi((value-mean)/(variance ** .5))
        if p > 0:
            return p
        elif p == 0:
            print "Error: normprob returning zero."
            return 10 ** -100
        elif p < 0:
            print "Error: normprob returning a negative number."
            return 10 ** -1000

    # As above, but logarithmic.
    def logNormProb(self, value, attribute, c):
        mean = self.means[c][attribute]
        variance = self.variances[c][attribute]
        if variance == 0:
            print "Warning: logNormProb encountered variance of zero."
            variance = 10 ** -5
```

```
        p = log(1/variance) + self.logPhi((value-mean)/(variance ** .5))
        return p

    # Returns the probability that a value occurs in the standard normal
    # distribution.
    def phi(self, value):
        p = (1/((2*math.pi)**.5)) * (math.e ** (-.5 * (value ** 2)))
        if p > 0:
            return p
        else:
            print "Error: phi returning non-positive number."
            return 10 ** -1000

    def logPhi(self, value):
        p = (1/((2*math.pi)**.5)) * (math.e ** (-.5 * (value ** 2)))
        if p > 0:
            return log(p)
        elif p == 0:
            return -1000000
        elif p < 0:
            print "Error: logPhi calculated a negative probability"
            return -1000000
        print "Error: logPhi reached an unreachable line"
        print "Value =", value
```

4.2 NaiveBayes.py

```
#! usr/bin/env python
# Implements Naive Bayes
# by Lionel Barrow
import math
from BayesClassifier import *
from numpy import *

class NaiveBayes(Classifier):

    # Constructor
    def __init__(self, classlist, datamatrix, k = 0):
        Classifier.__init__(self, classlist, datamatrix)
        [self.means, self.variances] = self.buildessentials()

    # Creates the matrices of means and variances for the classes
    def buildessentials(self):
        means = []
        variances = []
        for c in range(self.nclasses):
            means.append([mean(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
            variances.append([var(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
```

```
        return [means, variances]

    # Evaluate a single example, using an existing model
    def evaluate(self, example):
        pmax = 0
        cmax = -1
        for c in range(self.nclasses):
            p = 1
            for i in range(len(example)):
                p *= self.normprob(example[i],i,c)
            if p > pmax:
                pmax = p
                cmax = c
        if cmax == -1:
            print "Error: cmax referenced before assignment in evaluate(", example,
            "), indicating possible loss of precision. Cmax will be set to 0.",
            "A likely source of this error is very low variance in evaluation"
            cmax = 0
        return cmax

    # As above, but use logarithmic calculations.
    def logEvaluate(self, example):
        pmax = -1000
        cmax = -1
        for c in range(self.nclasses):
            p = 0
            for i in range(len(example)):
                p += self.logNormProb(example[i],i,c)
            if p > pmax:
                pmax = p
                cmax = c
        if cmax == -1:
            print "Error: cmax referenced before assignment in logEvaluate(", example,
            "), indicating possible loss of precision. Cmax will be set to 0.",
            "A likely source of this error is very low variance in evaluation"
            cmax = 0
        return cmax
```

4.3 FullBayes.py

```
#! usr/bin/env python
# Implements full Bayes
# by Lionel Barrow
import math
from BayesClassifier import *
from numpy import *

class FullBayes(Classifier):
    def __init__(self, classlist, datamatrix, k = 0):
        Classifier.__init__(self, classlist, datamatrix)
```

```
        [self.covs, self.means, self.variances] = self.buildessentials()

    def buildessentials(self):
        covariances = []
        for i in range(len(self.classdata)):
            covariances.append(cov(transpose(self.classdata[i])))
        means = []
        variances = []
        for c in range(self.nclasses):
            means.append([mean(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
            variances.append([var(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
        return [covariances, means, variances]

    # Determines the class of a single example.
    def evaluate(self, example):
        pmax = 0
        for c in range(self.nclasses):
            p = 1
            for i in range(len(example)):
                trueattvalue = example[i]
                for j in range(len(example)):
                    if i != j:
                        trueattvalue -= self.covs[c][i,j]*((example[j]-
*BR*self.means[c][j])/(self.variances[c][j] ** .5))
                printme = self.normprob(trueattvalue,i,c)
                p *= printme
            if p > pmax:
                pmax = p
                cmax = c
        return cmax

    # As above, but uses a logarithmic estimate of probability.
    def logEvaluate(self, example):
        pmax = 100000000
        cmax = -1
        for c in range(self.nclasses):
            p = 0
            for i in range(len(example)):
                trueattvalue = example[i]
                for j in range(len(example)):
                    if i != j:
                        trueattvalue -= self.covs[c][i,j]*((example[j]-
*BR*self.means[c][j])/(self.variances[c][j] ** .5))
                p += self.logNormProb(trueattvalue,i,c)
            if abs(p) < abs(pmax):
                pmax = p
                cmax = c
        if cmax == -1:
            print "Error: cbest referenced before assignment in FullBayes'
```

```
*BR*logEvalute(", example, "), returning 0."
            cmax = 0
        return cmax
```

4.4 StrongKNaiveBayes.py

```
#! usr/bin/env python
# Implements strong and weak k-Naive Bayes
# by Lionel Barrow
import math
import itertools
import pickle
from BayesClassifier import *
from numpy import *

class StrongKNB(Classifier):
    #Constructor
    def __init__(self, classlist, datamatrix, k=0, exactlyK = True, strong = True):
        Classifier.__init__(self, classlist, datamatrix)
        [self.covs, self.means, self.variances] = self.buildessentials()
        self.k = k
        self.weights = [self.attributeWeight(att) for att in range(self.nattributes)]
        self.attributes = range(self.nattributes)
        self.independents = self.optimalnaivete(self.k, exactlyK)
        self.dependents = list(set(self.attributes) - set(self.independents))
        self.strong = strong

    def buildessentials(self):
        covariances = []
        for i in range(len(self.classdata)):
            covariances.append(cov(transpose(self.classdata[i])))
        means = []
        variances = []
        for c in range(self.nclasses):
            means.append([mean(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
            variances.append([var(self.classdata[c][:,i]) for i in
*BR*range(self.nattributes)])
        return [covariances, means, variances]

    def linearModel(self, example, attribute, c):
        trueattvalue = example[attribute]
        for j in range(len(example)):
            if j != attribute:
                if self.variances[c][j] == 0:
                    self.variances[c][j] = 10 ** -6
                x = self.covs[c][attribute,j]*((example[j]-
*BR*self.means[c][j])/(self.variances[c][j] ** .5))
                trueattvalue -= x
        return self.logNormProb(trueattvalue, attribute, c)
```

```
    def probDependent(self, example, attribute, c):
        trueattvalue = example[attribute]
        if self.strong:
            for att in self.dependents:
                if att != attribute:
                    trueattvalue -= self.covs[c][attribute, att]
*BR* * ((example[att] - self.means[c][att]) / (self.variances[c][att] ** .5))
        if not self.strong:
            for att in self.attributes:
                if att != attribute:
                    trueattvalue -= self.covs[c][attribute, att]
*BR* * ((example[att] - self.means[c][att]) / (self.variances[c][att] ** .5))
        return self.logNormProb(trueattvalue, attribute, c)

    def dependenceDerivative(self, example, attribute, c):
        top = self.linearModel(example, attribute, c)
        bottom = self.logNormProb(example[attribute], attribute, c)
        dd = top - bottom
        return dd

    def ddr(self, example, attribute):
        return self.dependenceDerivative(example, attribute, 0)
*BR*- self.dependenceDerivative(example, attribute, 1)

    def attributeWeight(self, attribute):
        ddrs = [self.ddr(example[0:-1], attribute) for example in self.data]
        return sum(ddrs) / self.npoints

    def optimalnaivete(self, k, exactlyK = True):
        # If exactlyK is on, we only look at combinations of exactly size k.
        # If exactlyK is off, we look at combinations of size less than or equal to k.
        perms = list(itertools.combinations(range(self.nattributes), k))
        if not exactlyK:
            for i in range(1,k):
                perms.extend(list(itertools.combinations(range(self.nattributes), i)))

        bestweightsum = -10000000
        for perm in perms:
            perm = list(perm)
            weightsum = sum([self.weights[attribute] for attribute in perm])
            if abs(weightsum) < abs(bestweightsum):
                bestperm = perm
                bestweightsum = weightsum
        return bestperm

    def logEvaluate(self, example):
        pbest = 1000000000
        cbest = -1
        for c in self.classes:
            p = 0
```

```
            for att in self.independents:
                p += self.logNormProb(example[att], att, c)
            for otheratt in self.dependents:
                p += self.probDependent(example, otheratt, c)
            if abs(p) < abs(pbest):
                cbest = c
                pbest = p
        if cbest == -1:
            print "Error: cbest referenced before assignment in StrongKNB's
*BR* logEvalute(", example, "), returning 0."
            cbest = 0
        return cbest
```

4.5 TestHarness.py

```
#! usr/bin/env python
# Classifier Test Harness
# by Lionel Barrow

import math
import itertools
from NaiveBayes import NaiveBayes
from FullBayes import FullBayes
from StrongKNaiveBayes import StrongKNB
from numpy import *
import random
from operator import itemgetter

class TestHarness():
    # Constructor
    def __init__(self, trainingdata, classes, inclassifier, k = 0, exactlyK = True,
*BR*strong = True):
        self.k = k
        self.inclassifier = inclassifier
        if self.inclassifier == StrongKNB:
            self.classifier = self.inclassifier(classes, trainingdata, k, exactlyK,
*BR*strong = strong)
        else:
            self.classifier = self.inclassifier(classes, trainingdata)
        self.classes = classes
        self.nclasses = len(classes)
        self.trainingdata = trainingdata
        self.exactlyK = exactlyK

    # Shuffles an array
    def randomize(self, array):
        pairs = []
        for element in array:
            pairs.append([random.random(), element])
        pairs = sorted(pairs, key=itemgetter(0))
```

```
        return [pairs[i][1] for i in range(len(pairs))]

    # Evaluates all test data using an existing classifier, returning
    # the error matrix
    def runTest(self, testdata):
        ndata = len(testdata)
        errormatrix = [zeros(self.nclasses) for i in range(self.nclasses)]
        for example in testdata:
            c = int(example[-1])
            example = example[0:-1]
            errormatrix[c][self.classifier.evaluate(example)] += 1
        return errormatrix

    # As above, but uses logarithmic calculations.
    def runLogTest(self, testdata):
        ndata = len(testdata)
        errormatrix = [zeros(self.nclasses) for i in range(self.nclasses)]
        for example in testdata:
            c = int(example[-1])
            example = example[0:-1]
            errormatrix[c][self.classifier.logEvaluate(example)] += 1
        return errormatrix

    #Flatten one level of nesting
    def flatten(self, listOfLists):
        return itertools.chain.from_iterable(listOfLists)

    # Implements folded cross-validation
    def crossValidatedTest(self, folds):
        assert 1 < folds <= self.classifier.npoints
        errormatrix = array([zeros(self.nclasses) for i in range(self.nclasses)])
        foldsize = int(self.classifier.npoints/folds)
        tempdata = self.randomize(self.trainingdata)
        for i in range(folds):
            testdata = tempdata[0:foldsize]
            tempdata = tempdata[foldsize:]
            tempdata = array(tempdata)
            self.classifier = self.inclassifier(self.classes, tempdata, self.k)
            errormatrix = errormatrix + array(self.runTest(testdata))
            tempdata = append(tempdata, testdata, axis = 0)
        return errormatrix

    # As above, but uses logarithmic calculations
    def logCrossValidatedTest(self, folds):
        assert 1 < folds <= self.classifier.npoints
        errormatrix = array([zeros(self.nclasses) for i in range(self.nclasses)])
        foldsize = int(self.classifier.npoints/folds)
        tempdata = self.randomize(self.trainingdata)
        for i in range(folds):
            testdata = tempdata[0:foldsize]
            tempdata = tempdata[foldsize:]
```

```
            tempdata = array(tempdata)
            if self.inclassifier == StrongKNB:
                self.classifier = self.inclassifier(self.classes, tempdata, self.k,
*BR*self.exactlyK)
            else:
                self.classifier = self.inclassifier(self.classes, tempdata)
            errormatrix = errormatrix + array(self.runLogTest(testdata))
            tempdata = append(tempdata, testdata, axis = 0)
        return errormatrix

    # Works for multi-class problems
    def accuracy(self, errormatrix):
        errormatrix = array(errormatrix)
        return trace(errormatrix)/sum(errormatrix)

    def truePositiveRate(self, errormatrix, classvalue = 0):
        erromatrix = array(errormatrix)
        return errormatrix[classvalue][classvalue]/sum(errormatrix[classvalue])

    def falsePositiveRate(self, errormatrix, classvalue = 0):
        errormatrix = array(errormatrix)
        return (sum(errormatrix[classvalue]) - errormatrix[classvalue][classvalue])
*BR*/ sum(errormatrix[classvalue])

    # These metrics only make sense in the context of binary classification
    def precision(self, errormatrix, classvalue = 0):
        return self.truePositiveRate(errormatrix, classvalue) /
*BR*(self.truePositiveRate(errormatrix, classvalue) +
*BR*self.falsePositiveRate(errormatrix, classvalue))

    def recall(self, errormatrix, classvalue = 0):
        if classvalue == 0:
            return self.truePositiveRate(errormatrix, 0) /
*BR*(self.truePositiveRate(errormatrix, 0) + self.falsePositiveRate(errormatrix, 1))
        if classvalue == 1:
            return self.truePositiveRate(errormatrix, 1) /
*BR*(self.truePositiveRate(errormatrix, 1) + self.falsePositiveRate(errormatrix, 0))

    def fscore(self, errormatrix, classvalue = 0):
        return (2 * self.precision(errormatrix, classvalue) * self.recall(
*BR*errormatrix, classvalue)) / (self.precision(errormatrix, classvalue) +
*BR*self.recall(errormatrix, classvalue))

    def specificity(self, errormatrix, classvalue = 0):
        if classvalue == 0:
            return self.truePositiveRate(errormatrix, 1) / (self.
*BR*truePositiveRate(errormatrix, 1) + self.falsePositiveRate(errormatrix, 0))
        if classvalue == 1:
            return self.truePositiveRate(errormatrix, 0) / (self.
*BR*truePositiveRate(errormatrix, 0) + self.falsePositiveRate(errormatrix, 1))
```

```
# Sensitivity and recall are the same thing
def sensitivity(self, errormatrix, classvalue = 0):
    return self.recall(errormatrix, classvalue)
```

Bibliography

[1] Kevin Korb and Ann Nicholson, *Bayesian Artificial Intelligence*, Chapman and Hall, New York, NY, 2000.

[2] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *Elements of Statistical Learning*, Springer, Stanford, CA, 2008.

[3] Harry Zhang, *The Optimality of Naive Bayes*, Proceedings of the 17th International FLAIRS Conference (2004).

[4] Peter Reutemann, *ARFF Datasets*, 2005. `https://list.scms.waikato.ac.nz/pipermail/wekalist/2005-April/003747.html`.

[5] Weka Documentation, *Attribute-Relation File Format (ARFF)*, 2002. `http://www.cs.waikato.ac.nz/~ml/weka/arff.html`.

[6] Wikipedia, *File: ROC space-2.png*, 2009. `http://en.wikipedia.org/wiki/File:ROC_space-2.png`.

[7] A. Frank and A. Asuncion, *UCI Machine Learning Repository*, 2010. University of California, Irvine, School of Information and Computer Sciences, `http://archive.ics.uci.edu/ml`.