Data Discretization Simplified: Randomized Binary Search Trees for Data Preprocessing

Donald Joseph Boland Jr.

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2007

Keywords: Data Mining, Discretization, Randomized Binary Search Trees

Abstract

Data Discretization Simplified:
Randomized Binary Search Trees for Preprocessing

Donald Joseph Boland Jr.

Data discretization is a commonly used preprocessing method in data mining. Several authors have put forth claims that a particular method they have written performs better than other competing methods in this field. Examining these methods we have found that they rely upon unnecessarily complex data structures and techniques in order to perform their preprocessing. They also typically involve sorting each new record to determine its location in the preceding data. We describe what we consider to be a simple discretization method based upon a randomized binary search tree that provides the sorting routine as one of the properties of inserting into the data structure. We then provide an experimental design to compare our simple discretization method against common methods used prior to learning with Naïve Bayes Classifiers. We find very little variation between the performance of commonly used methods for discretization. Our findings lead us to believe that while there is no single best method of discretization for Naïve Bayes Classifiers, simple methods perform as well or nearly as well as complex methods and are thus viable methods for future use.

Dedication

To My Wife Kelly

To My Family

Acknowledgments

I would like to first express my truest and sincerest thanks to Dr. Tim Menzies. Over the past year and a half of working together, he has provided me with the guidance and support necessary to complete this project and grow as a student, researcher, and computer scientist. He has provided the inspiration to approach problems in computer science with a degree of curiosity which I had not previously experienced and taught me a variety of useful skills that I do not think I would have adopted otherwise, most specifically SWP: Script When Possible, which made completing this thesis bearable and easily changeable and repeatable when new ideas or wrinkles were introduced. My life is now encapsulated in a Subversion Repository where nothing can be easily lost and many things can travel easily, and I would not have adopted such a lifestyle without having the opportunity to work with Dr. Menzies. His interest in his student's success, his dedication to research and teaching, and his faith in my abilities have been a great inspiration in allowing me to complete this work. It has been a great honor and privilege to know and work with him.

I would also like to thank the other members of my committee, Dr. Roy Nutter and Professor Cindy Tanner for their support both in this project and working with me during my tenure at West Virginia University. Dr. Nutter's vast interests, from computer forensics to electric cars and everything in between has only helped to increase my interest in studying a variety of fields and not just isolating myself in one particular interest or field. His willingness to serve as an advisor while I searched for an area of interest at West Virginia University allowed me to reach this point. Professor Tanner, my first supervisor as a teaching assistant at West Virginia University, afforded me the opportunity to work with students as an instructor and mentor in her CS 111 labs. It is an opportunity that has allowed me to get a taste of what being a college instructor could be like and also has afforded me skills like being able to speak comfortably in front of groups, answer questions on the fly, and quickly adopt and understand programming languages well enough to instruct on them. I greatly appreciate her willingness to work with me and provide me with the latitude to learn these skills.

I would like to thank Lane Department of Computer Science and specifically Dr. John Atkins for expressing an interest in having me attend West Virginia University and for providing a variety of opportunities over the last few years so that I could pursue this graduate education. I have had the opportunity to study and work with so many great professors only because of the opportunities that were created by the teaching and research assistantships made available by West Virginia University.

I would like to thank my family for their continuing support and encouragement. Without their interest in my continuing success, their help in keeping me motivated, and their good humor when I my mood needed lightened, I would not have been able to achieve any of the successes involved with completing this document nor been able to stand finishing it.

Last, but far from least, I would like to thank my wife, Kelly. Her continuing love, patience, and willingness to play our lives by ear, along with the her unending support, made it possible to complete this project while getting married in the middle of it. I greatly appreciate her support in help me to maintain my sanity and other interests in the process. I look forward to spending more time with her and less time in front of my computer as this project comes to a close and our life together really begins.

Contents

List of Figures

Chapter 1

Introduction

Today's modern societies are built on information. Computers and the Internet can make information available quickly to anyone looking for it. More importantly, computers can process that information more quickly than many humans. They can also provide information about how best to make a decision that normally would have been made previously by a human being with imperfect knowledge built on their individual education and experience but not necessarily the best information. Computer can thus aid us in making the right decisions at the right moment using the best information available. This thesis deals with helping to refine the way computers decide which information is most pertinent and make, or help their human users make, decisions based upon it. We will discuss methods of automatically extracting patterns from large amounts of data, and methods by which we can improve the ways in which they perform. Specifically, we will explore a novel *discretization* method for continuous data. Such discretization is a common preprocessing method that is known to improve various data mining approaches. We will offer a new method based upon the *randomized binary search tree* data structure and compare its performance with existing state of the art discretization methods.

Chapter 1 provides background information about this thesis, specifically discussing the motivation behind the research herein, the purpose of this thesis, contributions that this thesis makes to the field of computer science and more specifically the topic area of data mining. Finally, this chapter explains the layout for the rest of this document.

Section 1.1 describes the problem that motivated this thesis, specifically discretization and the search for a simple solution that performs at about the same level as existing methods.

Section 1.2 states the purpose of the research of this thesis.

Section 1.3 states the contributions of this thesis to related research.

Section 1.4 explains the layout of the rest of this document and what can be expected in the following chapters.

1.1 Motivation

Data mining is the process of analyzing data in order to find undiscovered patterns in the data and solve real world problems [22]. It may be data about historic trends in beach erosion to help a local community determine how much sand needs dredged and replaced each year, or survey data about when people begin their Christmas shopping in order to help retailers determine the best time of year to begin setting up Christmas displays and ordering seasonal merchandise. Data about a set of tests that identify cancer might be analyzed to determine which tests are most capable of identifying the cancer and allow doctors to use these tests earlier in the cancer screening process, or data about fuel purchases or consumption analyzed and used as a basis for vendors to know how much fuel they should have on hand at a particular time of year, how often they should be restocked, and specific amounts of each fuel grade or type might be needed. Data mining can be used to analyze a vast variety of data in order to solve the problems faced by our society or provide more information to help people make the best decisions.

Real world data such as that collected for the problems above can provide a variety of issues for data miners, but one of the chief problems involved in preparing data for the learner is ensuring that data can be easily read and manipulated by the learner. One of the most common difficulties that learners have is dealing with numeric values. Most learners require data to take on a value belonging to a small, fixed set, which is often unobtainable with raw numeric values that can fall in large or infinite ranges and take on many possible values even when constrained by a range. The process of transitioning raw numeric values to a form that can be easily read and manipulated by learners is called *discretization* [22]. Numerous researchers report that discretization leads to better, more accurate learning, especially in Naïve Bayes Classifiers. However, they very often disagree about which method of discretization works best. Because of how useful discretization can be for classification, yet because questions remain about whether there is one best method to use, discretization will be the subject of this thesis.

1.2 Statement of Thesis

While data discretization is an important topic in data mining, it is one burdened with a vast variety of methods, most of which take on complex data structures and require a search over the entire data set to determine how a value should be discretized. We believe that there should be a simpler approach that works similarly to these methods. To that end, we have implemented a discretization method based on a randomized binary search tree as the underlying storage data structure. We contend that this method uses the properties of *randomized binary search trees* to avoid a search over the entire data set performing discretization and do so with a simple structure that can be understood by most.

1.3 Contributions

The contributions of this thesis are:

- The DiscTree algorithm that is implemented to create the DiscTree discretization method;
- A review of a variety of currently existing discretization methods; and,
- An experimental comparison of some common discretization methods against the implemented DiscTree discretization method.

A surprise finding of this comparison is that many discretization methods perform at very similar levels.

The results of the comparison leads us to the belief that discretization is a simpler task than it is made out to be in some of the literature. The DiscTree algorithm is simple in comparison to many of the state-of-the-art methods and performs just as well as some methods that claim superiority in the field of discretization. We believe that while various methods exist for discretization and some may perform better on a specific data sets than others, that in general simple methods perform well and can be just as useful as and used in place of complex methods.

1.4 About This Document

The rest of the chapters of this thesis are laid out as follows:

Chapter 2 provides an explanation of the premise of data and how it is used in data mining. It also provides a review of various learners in data mining. It examines several possible learning methods and explains why we have chosen to use the Naïve Bayes Classifier for our experimentation with discretization methods.

Chapter 3 provides a review of common data mining discretization methods. It highlights the methods commonly found in the literature on discretization and specifically reviews the methods we will compare in our experiment.

Chapter 4 explains the experimental design used to test the variety of data discretization techniques described in Chapter 3. It also explains our methods for generating and comparing results.

Chapter 5 contains the results of the experiment, relevant tables and data plots, and a explanation of those results.

Chapter 6 explains conclusions derived from the results of the experiment. It discusses the key findings and areas of future work that could expand upon this thesis. It also provides a summary of this document.

Chapter 2

Background: Data and Learners

Chapter 2 provides background information on data mining, specifically the topics of data and classification. It provides information about some of the common classification methods and explains our selection of the Naïve Bayes classifier as a test platform for discretization.

Section 2.1 describes the use of data in data mining, including types of data and a basic explanation of the format of the data used in this thesis. Section 2.2 describes the machine learning process of classification and discusses a sampling of various classifiers, including decision tree and Naïve Bayes classifiers. Section 2.3 explains the usefulness of the information of this Chapter and how it leads to our selection of a classification method for the experiments in this document and the justification for that selection.

2.1 Data and Data Mining

2.1.1 Data

In this modern age, almost everything we do is a source of data. Prompt payment of bills is recorded by credit agencies to maintain or increase a credit score or credit limit, while late payments may decrease it or decrease future credit opportunities. Purchases from websites are recorded to determine other items or services that the company or its business partners might offer or to send reminders when a service needs renewed or an item replaced. Grades, standardized test scores, extra-curricular involvement, and student personal information are all collected by colleges and universities to be analyzed for admission and scholarships. Almost any imaginable piece of information is useful to someone, and most of it can and does get recorded as data in electronic databases.

Data captured from the real world comes in a variety of forms. Values may arrive as a series of selections, such as a choice of favorite color from the set blue, red, green, yellow, orange, pink, purple, or a choice of marital status from the set single, married, divorced, widowed. Such qualitative data, where the values are chosen from a finite set of distinct possible values, is called *nominal* or *categorical* data. Ordinal data, where the fixed categories have some sort of relation to each other, such as age ranges 0 to 9, 10 to 19, ... ,110 to 120 where ”older” and ”younger” ranges can be discussed, may also be referred to as *discrete* data [22]. However, because there exists no concept of distance between ordinal data values - that is, you can not add two of such values to obtain a third or subtract one from another and be left with a third - they are often treated like nominal values. Other data may arrive as measurements, such as the monthly rainfall of a city, the average rushing yards per touch of a football player, or a person’s average weekly spending at the grocery store. These measurements, which make take on an almost unlimited number of quantitative values, are called *numeric* or *continuous* data, and may includes both real (decimal) and integer values [22].

Data is most often stored in files or databases. The basic unit of these storage structures is the record, or one data *instance*. Each instance can be considered to be a line in a data file or a row in a database table. Each instance is made up of values for the various *attributes* that comprise it. The attributes or *features* of each instance, the columns of our database table or file, are the information we wish to know for each instance. From the previous example about student admissions and financial aid data, a student instance might be comprised of a SAT score attribute, an ACT score attribute, a class ranking attribute, a GPA attribute, a graduation year attribute, and an attribute that denotes whether the college or university collecting that information gave that student financial aid. Instances often consist of mixed format data; that is, an instance will often have some nominal or discrete attributes and some continuous attributes [12]. Another example of a set of instances can be found in Figure 2.1. Each record or instance is a row in the table and is labeled here with a number that is not part of the data set for reference purposes. Each column has the name of the attribute that it represents at the top.

Instance	*Attributes*				*Class*
	outlook	temperature	humidity	windy	play
1	sunny	85	85	false	no
2	sunny	80	90	true	no
3	overcast	83	86	false	yes
4	rainy	70	96	false	yes
5	rainy	68	80	false	yes
6	rainy	65	70	true	no
7	overcast	64	65	true	yes
8	sunny	72	95	false	no
9	sunny	69	70	false	yes
10	rainy	75	80	false	yes
11	sunny	75	70	true	yes
12	overcast	72	90	true	yes
13	overcast	81	75	false	yes
14	rainy	71	91	true	no

Figure 2.1: The WEATHER data set, with both nominal and continuous values

While advances in storage technology have allowed the collection and storage of the vast amount of data now available, the explosion of available data does not always mean that the collected data is being used to its full potential. Often, the pure massiveness of the data collected can overwhelm those who have requested it be stored. They may find themselves staring at a mountain of data that they didn't expect and don't know how they will ever analyze. Even if they do manage to view it all, they may only see the facts that are obvious in the data, and sometimes may even miss these. Fortunately, the same computers that are storing the collected data can aid these data-swamped users in their analysis.

2.1.2 Data Mining

Data analysis may seem trivial when data sets consist of a few records consisting of few attributes. However, human analysis quickly becomes impossible when datasets become large and complex, consisting of thousands of records with possibly hundreds of attributes. Instead, computers can be used to process all of these records quickly and with very little human interaction. The process of using computers to extract needed, useful, or interesting information from the often large pool of available data is called *data mining*. More precisely, data mining is the extraction of implicit, previously unknown, and potentially useful information about data [22].

The technical basis of data mining is called *machine learning* [22]. A field within artificial intelligence, it provides many of the algorithms and tools used to prepare data for use, examine that data for patterns, and provide a theory based on that data by which to either explain previous results or predicting future ones [17,22]. These tools provide the information they gather to analysts, who can then use the results to make decisions based on the data patterns, anticipate future results, or refine their own models. Data mining thus becomes a tool for descriptive prediction, explanation, and understanding of data that might otherwise be lost within the ever growing sea of information [22].

2.2 Classification

Classification, also referred to as *classification learning*, is a type of data mining whereby a computer program called a *learner* is provided with a set of pre-classified example instances from which it is expected to learn a way to classify future, unseen, unclassified instances [22]. Most often, the pre-classified examples are prepared by experts or are real, past examples which are supposed to represent the known or accepted rules about the data. The learner is provided with these in order to then form its own rules for how to treat future instances. It does this, in general, by examining the attributes of the example instances to determine how they are related to that instances *class*. The class of an instance is an attribute which denotes the outcome for the instance. From the previous student financial aid example, if we were using a classifier to determine whether students should receive student aid, this class attribute would be the attribute denoting whether the student received financial aid or not. In the data set in Figure 2.1, the class attribute, play, takes on the values of *yes* and *no*, denoting a decision as to whether some decision is made based on the weather. The learner would examine the set of example instances and build a *concept* by which it relates the other attributes to the class attribute to make a set of rule for how to decide which class future instances will be assigned [22]. The method by which the learner determines the concept it will use on future examples differs based upon the type of *classification learner* used. A wide variety of classification learners exist, but among the most popular are decision tree learners, rule-generating learners, and Naïve Bayes classifiers.

2.2.1 Decision Tree Learners

Decision tree learners use a method called *decision tree induction* in order to construct its concept for classification. In decision tree induction, an attribute is placed at the root of the tree (see Section 3.11.1) being created and a branch from that root is created for each value of that attribute. This process is then repeated recursively for each branch, using only the instances that are present in the created branch [17, 22]. The process stops when either too few examples fall into a created branch to justify splitting it further or when the branch contains a pure set of instances (i.e. the class of each example in the branch is the same). Once a decision tree has been built using training examples, test examples can be classified by starting at the root of the tree and using the attribute and conditional tests at each internal node and branch to reach a leaf node that provides a class for examples that reach the given leaf. Decision trees are thus trees whose leaf nodes provide classifications to examples who reach those leaves by meeting the conditional statements of the preceding branches of the tree. Figure 2.2 provides an example of a decision tree.

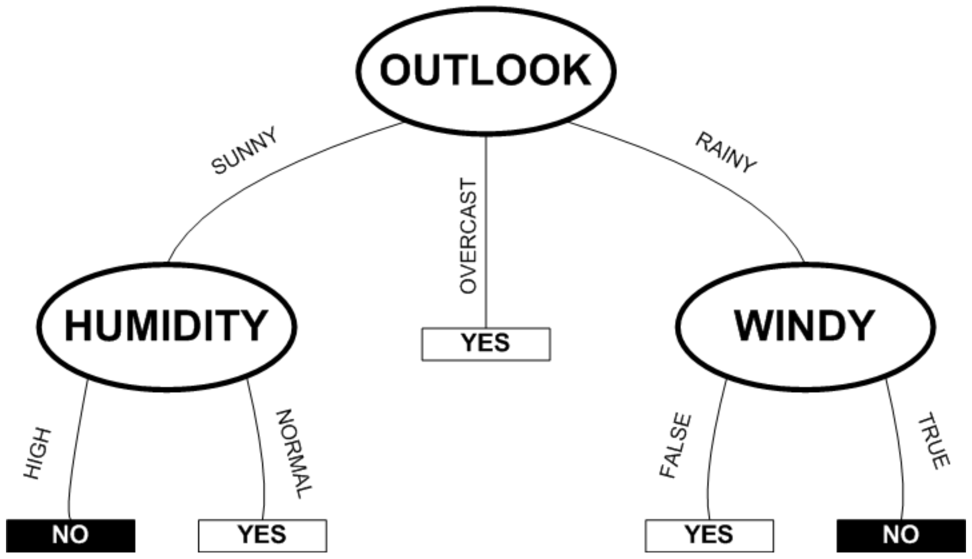


Figure 2.2: A Sample Decision Tree

An example of a decision tree learner is J48. J48 is a JAVA implementation of Quinlan's C4.5 (version 8) algorithm [18]. J48/C4.5 treat numeric attributes using a binary-chop at any level, splitting the attribute into two parts that can later be chopped again if necessary (i.e. in this case an attribute may be reused). C4.5/J48 uses information theory to assess candidate attributes in each tree level: the attribute that causes the *best split* is the one that *most simplifies* the target concept. Concept simplicity is measured using information theory and the results are measured in bits. It does this using the following equations:

$$entropy(p_1, p_2, ..., p_n) = -p_1 log(p_1) - p_2 log(p_2) - ... - p_n \log(p_n) \tag{2.1}$$

$$or$$

$$entropy(p_1, p_2, ..., p_n) = -\sum_{i=1}^{n} p_i \log p_i$$

$$info([x,y,z]) = entropy(\frac{x}{x+y+z}, \frac{y}{x+y+z}, \frac{z}{x+y+z}) \tag{2.2}$$

$$gain(attribute) = info(current) - avg.\ info(proposed) \tag{2.3}$$

A good split is defined as one most decreases the number of classes contained in each branch. This helps to ensure that each subsequent tree split results in smaller trees requiring fewer subsequent splits.

Equation 2.1 defines the *entropy* - the degree of randomness of classes in a split. The smaller the entropy is - the closer it is to zero - the less even the class distribution in the split; the larger the entropy - the closer it is to one - the more evenly divided the classes in the split.

Information, measured in bits, specifies the purity of a branch in the decision tree. The information measure of a given leaf node of a decision tree specifies how much information would be necessary to specify how a new example should be classified at the should the given example reach the given leaf node in the tree. Equation 2.2 allows the calculation of that amount of information. For example, if the leaf node contained 5 example instances, 3 of class *yes* and 2 of class *no*, then the information needed to specify the class of a new example that reached that leaf node would be:

$$info([3,2]) = entropy(\frac{3}{5},\frac{2}{5}) = -\frac{3}{5}\log\frac{3}{5} - \frac{2}{5}\log\frac{2}{5} \approx 0.971\ bits$$

The *information gain*, defined in Equation 2.3, of a split is the decrease of information needed to specify the class in a branch of the tree after a proposed split is implemented. For example, consider a tree with twenty(20) training instances with an original class distribution of thirteen(13) *yes* instances and seven *no* instances. A proposed attribute value test would split the instances into three branches; one containing only seven *yes* instances and one *no*instances, the second five *yes* and one *no*, and the third the remaining instances. The information of the original split is calculated, $info([13,7]) \approx 0.934\ bits$. Each of the information measures for the splits that would be created are also generated, and an average value derived. This average value is the class information entropy of the attribute, a formula for which an be found in Equation 2.4:

$$E(attribute) = \frac{|S_1|}{|S|}entropy(S_1) + ... + \frac{|S_n|}{|S|}entropy(S_n) \tag{2.4}$$

Where S_1 through S_n are the subsets created when *attribute* takes on n unique values and thus creates n branches if used as the split point in the tree; S is the original distribution of classes for this split; and $|S_i|$ is the size - number of instances - in S_i. Applying this formula to our previous example, we get:

$$\begin{aligned} info([7,1]) \approx&\ 0.544\ bits \\ info([5,1]) \approx&\ 0.650\ bits \\ info([1,5]) \approx&\ 0.650\ bits \\ E([7,1],[5,1],[1,5]) = \frac{8}{20}*.544 + \frac{6}{20}*.650 + \frac{6}{20}*.650 \approx&\ 0.413\ bits \end{aligned}$$

Then the information gain for the proposed split would be:

$$gain(attribute) = info([13,7]) - E([7,1],[5,1],[1,5]) = 0.934\ bits\ -\ 0.413\ bits = 0.521\ bits$$

The gain for each attribute that might be used as the splitting attribute at this level of the tree would be compared and the one that maximizes this gain would be used as the split; in the case of a tie an arbitrary choice could be made. However, simply using gain can present an issue in the case of highly branching attributes, such as an unique ID code assigned to each instance. Such an attribute would create a separate branch for each attribute and have an extremely small (zero) information score that would result in a very high gain. While such a split attribute would be desired using just the gain measure, it would not be desired in a tree split because it would lead to an issue of *over-fitting*. Over-fitting occurs when a few, very specific values are used in the creation of a classification concept, that results in a concept that always or most often will result in a misclassification during testing. The ID code attribute would cause such a problem to occur, most likely never predicting instances incorrect that did not appear in the training set. In order to avoid this, another measure is used that takes into account both the number and size of child nodes of a proposed split. This measure is called the *gain ratio* [22]. To calculate gain ratio, we start with the gain calculated previously, and divide it by the information that is derived from the number of instances (the sum of the number of instances of each class) in each split. From the previous example, we could calculate the gain ratio as follows:

$$\begin{aligned}
gain(attribute) &= 0.521\ bits \\
info([8,6,6]) &= entropy([\frac{8}{20},\frac{6}{20},\frac{6}{20}]) \\
&= -\frac{8}{20}\log\frac{8}{20} - \frac{6}{20}\log\frac{6}{20} - \frac{6}{20}\log\frac{6}{20} \\
info([8,6,6]) &\approx 1.571\ bits \\
gain\ ratio = \frac{gain((attribute))}{info([8,6,6])} = \frac{0.521\ bits}{1.571\ bits} &\approx 0.332\ bits
\end{aligned} \tag{2.5}$$

The attribute with the highest gain ration is then used as the split point. Additionally, certain other tests may be included in some decision tree induction schemes to ensure that the highly branching attribute described previously is not even considered as a possible splitting attribute. As described previously, the splitting process in decision tree induction continues in each of the created branches until some stopping criterion is reached, be it too few instances left in a branch to justify splitting, a pure branch, or some other test.

Trees created by C4.5/J48 are pruned back after they are completely built in order to avoid *over-fitting error*, where a specific branch of the tree is too specific to one or a few training examples that might cause an error when used against the testing data. This methodology uses a *greedy* approach, setting some threshold by which the accuracy of the tree in making classifications is allowed to degrade and removing the branches in reverse order until that threshold is met. This ensures that branches do not become over-fitted for a specific instance, which could decrease the accuracy of the tree - especially if the one training instance that fell into that branch was an extreme outlier, had been corrupted by noise in the data, or was simply a random occurrence that got grouped into the training set.

Quinlan implemented C4.5 decision tree post-processor called C4.5rules. This post-processor generates succinct rules from cumbersome decision tree branches via (a) a greedy pruning algorithm that removes statistically unnecessary rules followed by (b) removal of duplicate rules and finally (c) exploring subsets of the rules relating to the same class [18]. It is similar to the rule-learners discussed in Section 2.2.3

2.2.2 Naive Bayes

Naïve Bayes classifiers are highly studied statistical method used for classification. Originally used as a *straw man* [9, 28] - a method thought to be simple and that new methods should be compared against in order to determine their usefulness in terms of improved accuracy, reduced error, etc - it has since been shown to be a very useful learning method and has become one of the frequently used learning algorithms.

Naïve Bayes classifiers are called naïve because of what is called the *independent attribute assumption*. The classifier assumes that each attribute of an instance is unrelated to any other attribute of the instance. This is a simplifying assumption used to make the mathematics used by the classifier less complicated, requiring only the maintenance of frequency counts for eacy attribute. However, real world data instances may contain two or more related attributes whose relationship could affect the class of a testing instance. Because of the independent attribute assumption, that relationship would most likely be ignored by the Naïve Bayes classifier and could result in incorrect classification of an instance. When a data set containing such relationships is used with the Naïve

Bayes classifier, it can cause the classifier to skew towards a particular class and cause a decrease in performance. Domingos and Pazzani show theoretically that the independence assumption is a problem in a vanishingly small percent of cases [9]. This explains the repeated empirical result that, on average, Naïve Bayes classifiers perform as well as other seemingly more sophisticated schemes. For more on the Domingos and Pazzani result, see Section 2.3.2

A Naïve Bayes classifier is based on Bayes' Theorem. Informally, the theorem says $next = old * new$; in other words, what we'll believe *next* is determined by how *new* evidence affects *old* beliefs. More formally:

$$P(H|E) = \frac{P(H)}{P(E)} \prod_i P(E_i|H) \tag{2.6}$$

That is, given fragments of evidence regarding current conditions E_i and a prior probability for a class $P(H)$, the theorem lets us calculate a posterior probability $P(H|E)$ of that class occurring under the current conditions. Each class (hypothesis) has its posterior probability calculated in turn and compared. The classification is the hypothesis H with the highest posterior $P(H|E)$.

Equation 2.6 offers a simple method for handling missing values. Generating a posterior probability means tuning a prior probability to new evidence. If that evidence is missing, then no tuning is needed. In this case Equation 2.6 sets $P(E_i|H) = 1$ which, in effect, makes no change to $P(H)$. This is very useful, as real world data often contains missing attribute values for certain instances; take, for instance, the student data mentioned previously. Not all students will take a particular standardized test, so using both the ACT and SAT scores in classification might be harmed in other methods if a missing value were to occur. However, with Naïve Bayes, this missing value does not harm or help the chance of classification, making it ideal for data that may having missing attribute values.

When estimating the prior probability of hypothesis H, it is common practice [23,24] to use an *M-estimate* as follows. Given that the total number of classes/hypothesis is C, the total number of training instances is I, and $N(H)$ is the frequency of hypothesis H within I, then:

$$P(H) = \frac{N(H) + m}{I + m \cdot C} \tag{2.7}$$

Here m is a small non-zero constant (often, $m = 2$). Three special cases of Equation 2.7 are:

- For high frequency hypothesis in large training sets, $N(H)$ and I are much larger than m and $m \cdot C$, so Equation 2.7 simplifies to $P(H) = \frac{N(H)}{I}$, as one might expect.
- For low frequency classes in large training sets, $N(H)$ is small, I is large, and the prior probability for a rare class is never less than $\frac{1}{I}$; i.e. the inverse of the number of instances. If this were not true, rare classes would never appear in predictions.
- For very small data sets, I is small and $N(H)$ is even smaller. In this case, Equation 2.7 approaches the inverse of the number of classes; i.e. $\frac{1}{C}$. This is a useful approximation when learning from very small data sets when all the data relating to a certain class has not yet been seen.

The prior probability calculated in Equation 2.7 is a useful lower bound for $P(E_i|H)$. If some value v is seen $N(f = v|H)$ times in feature f's observations for hypothesis H, then

$$P(E_i|H) = \frac{N(f = v|H) + l \cdot P(H)}{N(H) + l} \tag{2.8}$$

Here, l is the *L-estimate*, or *Laplace-estimate* and is set to a small constant (Yang &Webb [23, 24] recommend $l = 1$). Two special cases of are:

- A common situation is when there are many examples of an hypothesis and numerous observations have been made for a particular value. In that situation, $N(H)$ and $N(f = v|H)$ are large and Equation 2.8 approaches $\frac{N(f=v|H)}{N(H)}$, as one might expect.
- In the case of very little evidence for a rare hypothesis, $N(f = v|H)$ and $N(H)$ are small and Equation 2.8 approaches $\frac{l \cdot P(H)}{l}$; i.e. the default frequency of an observation in a hypothesis is a fraction of the probability of that hypothesis. This is a useful approximation when very little data is available.

For numeric attributes it is common practice for Naïve Bayes classifiers to use the Gaussian probability density function [22]:

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.9}$$

where $\{\mu, \sigma\}$ are the attributes's $\{$mean,standard deviation$\}$, respectively. To be precise, the probability of a continuous (numeric) attribute having exactly the value x is zero, but the probability that it lies within a small region, say $x \pm \varepsilon/2$, is $\varepsilon \times g(x)$. Since ε is a constant that weighs across all possibilities, it cancels out and needs not be computed. Yet, while the Gaussian assumption may perform nicely with some numeric data attributes, other times it does not and does so in a way that could harm the accuracy of the classifier.

One method of handling non-Gaussians is Johns and Langley's *kernel estimation* technique [11]. This technique approximates a continuous distribution sampled by n observations $\{ob_1, ob_2, ..., ob_n\}$ as the sum of multiple Gaussians with means $\{ob_1, ob_2, ..., ob_n\}$ and standard deviation $\sigma = \frac{1}{\sqrt{n}}$. In this approach, to create a highly skew distribution, multiple Gaussians would be added together. Conclusions are made by asking all the Gaussians which class they believe is most likely.

Finally, numeric attributes for Naïve Bayes classifiers can also be handled using a technique called *discretization*, discussed in Chapter 3. This has been the topic of many studies ([4, 14, 23–25, 28]) and has been shown to deal well with numeric attributes, as seen in [9] where a Naïve Bayes classifier using a simple method of discretization outperformed both so-called state-of-the-art classification methods and a Naïve Bayes classifier using the Gaussian approach.

Naïve Bayes classifiers are frustrating tools in the data mining arsenal. They exhibit excellent performance, but offer few clues about the structure of their models. Yet, because their performance remains so competitive with other learning methods their structures, this complaint is often overlooked in favor of their use.

2.2.3 Other Classification Methods

1-R

One of the simplest learners developed was 1-R [13, 22]. 1-R examines a training dataset and generates a one-level decision tree for an attribute in that data set. It then bases its classification decision on the one-level tree. It makes a decision by comparing a testing instance's value for the attribute that the tree was constructed against the decision tree values. It classifies the test instance as being a member of the class that occurred most frequently in the training data with the attribute value. If several classes occurred with equal frequency for the attribute value then a random decision is used at the time of final tree construction to set the class value that will be used for future classification.

The 1-R classifier decides which attribute to use for future classification by first building a set of rules for each attribute, with one rule being generated for each value of that attribute seen in the training set. It then tests the rule set of each attribute against the training data and calculates the error rate of the rules for each attribute. Finally, it selects the attribute with the lowest error - in the case of a tie the attribute is decided arbitrarily - and uses the one-level decision tree for this attribute when handling the testing instances. Pseudo code for 1-R can be found in Figure 2.2.3:

```
For each attribute:
    For each value of that attribute, make a rule as follows:
        Count how often each class appears
        Determine the most frequent class
        Make a rule such that assigns the given value the most frequent class
    Calculate the error rate of the rules for the attribute
Compare the error rates, determine which attribute has the smallest error rate
Choose the attribute whose rules had the smallest error rate
```

Figure 2.3: 1-R Pseudo-Code

The 1-R classifier is very simple and handles both missing values and continuous attributes. Continuous attributes are handled using discretization, discussed in Chapter 3. It specifically uses a method similar to *EWD*, defined in Section 3.2. Missing values are dealt with by creating a branch in the one-level decision tree for a *missing* value. This branch is used when missing values occur.

Because of its simplicity, 1-R often serves a ”straw-man” classification method, used as a baseline for performance for new classification algorithms. While 1-R sometimes has classification accuracies on par with modern learners - thus suggesting that the structures of some real-world data are very simple - it also sometimes performs poorly, giving researchers a reason to extend beyond this simple classification scheme [17].

Rule Learners

Rather than patch an opaque learner like Naïve Bayes classifierswith a post-processor to make them more understandable to the average user, it may be better to build learners that directly generate succinct, easy to understand, high-level descriptions of a domain. For example, RIPPER [5] is one of the fastest *rule learners* in the available literature. The generated rules are of the form $condition \longrightarrow conclusion$:

$$\underbrace{Feature_1 = Value_1 \wedge Feature_2 = Value_2 \wedge \ldots}_{condition} \longrightarrow \underbrace{Class}_{conclusion}$$

The rules generated by RIPPER perform as well as C45rules - a method which creates rules from C4.5 decision trees - yet are much smaller and easier to read [5].

Rule learners like RIPPER and PRISM [3] generate small, easier to understand, symbolic representations of the patterns in a data set. PRISM is a less sophisticated learner than RIPPER and is no longer widely used. It is still occasionally used to provide a lower bound on the possible performance. However, as illustrated below, it can still prove to be surprisingly effective.

```
(1) Find the majority class C
(2) Create a R with an empty condition that predicts for class C.
(3) Until R is perfect (or there are no more features) do
        (a) For each feature F not mentioned in R
        (b) For each value v in F, consider adding F = v to the condition of R
        (c) Select F and v to maximize p/t where t is
            total number of examples of class C and p is the number
      of examples of class C selected by F=v.
      Break ties by choosing the condition with the largest p.
  (d) Add F = v to R
(4) Print R
(5) Remove the examples covered by R.
(6) If there are examples left, loop back to (1)
```

Figure 2.4: PRISM pseudo-code.

Like RIPPER, PRISM is a *covering* algorithm that runs over the data in multiple passes. As shown in the pseudo-code of Figure 2.4, PRISM learns one rule at each pass for the *majority class* (e.g. in Figure 2.1, at pass 1, the majority class is *yes*). All the examples that satisfy the condition are marked as *covered* and removed from the data set currently begin considered for a rule. PRISM then recurses on the remaining data.

The output of PRISM is an ordered *decision list* of rules where $rule_j$ is only tested on instance x if all conditions in $rule_{i:i<j}$ fail to cover x. PRISM returns the conclusion of the first rule with a satisfied condition.

One way to visualize a covering algorithm is to imagine the data as a table on a piece of paper. If there exists a clear pattern between the features and the class, define that pattern as a rule and cross out all the rows covered by that rule. As covering recursively explores the remaining data, it keeps splitting the data into:

- What is easiest to explain during this pass, and
- Any remaining ambiguity that requires a more detailed analysis.

PRISM is a naïve covering algorithm and has problems with *residuals* and *over-fitting* similar to the decision tree algorithms. If there are rows with similar patterns and similar frequencies occur in different classes, then:

- These *residual* rows are the *last* to be removed for each class;
- so the *same* rule can be generated for *different* classes. For example, the following rules might be generated: if x then class=*yes* and if x then class=*no*.

As mentioned in the discussion on decision tree learners, in *over-fitting*, a learner fixates on rare cases that do not predict for the target class. PRISM's over-fitting arises from part 3.a of Figure 2.4 where the algorithm loops through all features. If some feature is poorly measured, it might be noisy (contains spurious signals/data that may confuse the learner). Ideally, a rule learner knows how to skip over noisy features.

RIPPER addresses residuals and over-fitting problem three techniques: *pruning*, *description length* and *rule-set optimization*. For a full description of these techniques, which are beyond the scope of this thesis, please see [8]. To provide a quick summary of these methods:

- *Pruning:* After building a *rule*, RIPPER performs a back-select in a *greedy* manner to see what parts of a *condition* can be deleted, without degrading the performance of the rule. Similarly, after building a *set of rules*, RIPPER performs a back-select in a *greedy* manner to see what *rules* can be deleted, without degrading the performance of the rule set. These back-selects remove features/rules that add little to the overall performance. For example, back pruning could remove the residual rules.

- *Description Length:* The learned rules are built while minimizing their *description length*. This is an information theoretic measure computed from the size of the learned rules, as well as the rule errors. If a rule set is over-fitted, the error rate increases, the description length grows, and RIPPER applies a rule set pruning operator.

- *Rule Set Optimizaton:* tries replacing rules with straw-man alternatives (i.e. rules grown very quickly by some naïve method).

Instance-Based Learning

Instance-based learners perform classification in a *lazy* manner, waiting until a new instance is inserted to determine a classification. Each new added instance is compared with those already in the data set using a *distance metric*. In some instance-based learning methods, the existing instance closest to the newly added instance is used to assign a ”group” or classification to the new instance. Such methods are called *nearest-neighbor* classification methods. If instead the method used the majority class, or a distance-weighted average majority class, of the *k* closest existing instances, the classification method is instead called a *k-nearest-neighbor* classification method.

While such methods are interesting to explore, their full and complete explanation is beyond the scope of this thesis. This introduction is provided as a simple basis for the idea of instance-based learning rather than specific details about specific methods. For more information about instance-based classification methods, we recommend starting with [22], which provides an excellent overview and explores specific instance-based methods such as *k-means, ball trees, and kD-trees*.

2.3 Summary

2.3.1 Data Mining and Classification

Data Mining is a large field, with many areas to study. This chapter has touched primarily on classification and classifiers. Classification is a very useful tool for a variety of industries. Classifiers can review a variety of medical test data to make a decision about whether a patient is at high risk for a particular disease. They can be used by retailers to determine which customers might be ideal for special offers. They could also be used by colleges and universities to determine which students they should admit, which students to spend time recruiting, or which students should be provided financial aid. These are just a few of the very large number of instances where classification could be used to the benefit of the organization who choses to use it.

Because classification is of such use to so many organizations, many people have studied it. The result of that study is the variety of different classification methods discussed in this chapter, from rule-based and instance-based learning to decision tree induction methods and Naïve Bayes classifiers. The goal of all this research is to find a better classifier, one that performs quickly and more accurately than previous classifiers. Yet, other data mining methods exist that can help to extend the accuracy of current methods, enabling them to be more accurate without additional manipulation of the classifier itself. These methods are often preprocessing steps in the data mining process, better preparing the data for use by the classifier. One such method is *discretization*. Discretization, in general, removes numeric data - which can often cause concept confusion, overfitting, and decrease in accuracy - from the original data and substitutes a nominal attribute and corresponding values in its place. Discretization is discussed in detail in Chapter 3. Because of its usefulness as a preprocessing method to classification, we propose to examine the effects of several methods of discretization on a classifier. But which classifier would best serve as a testing platform?

2.3.2 Classifier Selection

A variety of literature exists comparing many of these classifier methods and how discretization works for them. In [14], three discretization methods are used on both the C4.5 decision tree induction algorithm and the Naïve Bayes Classifier. The authors of that paper find that each form of discretization they tested improved the performance of the Naïve Bayes Classifier in at least some cases. Specifically:

> Our Experiments reveal that all discretization methods for the Naive-Bayes classifier lead to a large average increase in accuracy.

On the other hand, when the same methods were used on the C4.5 learner only two datasets saw significant improvement. This result leads us to believe that Naïve Bayes classifiers truly provides a platform for discretization methods to improve results and have a true, measurable impact on the classifier.

In addition to that study, [9] compared the performance of the Naïve Bayes classifiers against C4.5 decision tree induction, PEBLS 2.1 instance-based learning, and CN2-rule induction. It compared those methods against both a Gaussian-assumption Naïve Bayes classifier, which uses an assumption that all continuous features fit in a normal distribution to handle such values, and a version of Naïve Bayes that uses Equal Width Discretization (see Section 3.2) as a preprocessor to handle any continuous data instances. It found that the simple Naïve Bayes classifier using EWD performed the best out of the compared methods, even compared against methods considered to be state-of-the-art, and that the Naïve Bayes classifier with the Gaussian-assumption performed nearly as well. The paper also went on to test whether violating the attribute independence assumption caused the classifier to significantly degrade, and found that the Naïve Bayes classifier still performed well when strong attribute dependencies or relationships were present in the data.

Finally, some of the most recent developments in discretization have been proposed specifically for use with the Naïve Bayes classifier. The most modern discretization method used in our experiment, aside from the DiscTree method implementation, is the PKID discretization method (see Section 3.6). This method was derived with the specific intent of it being used with the Naïve Bayes classifier, and in order to provide a comparison with the results of the study performed with its implementation, we believe it necessary to perform a comparison using that classifier. The same author has proposed numerous other methods of discretization for Naïve Bayes as well, specifically in [23–25, 27, 28]. This leads us to believe that we too may be able to improve the performance of this classifier.

As a result, we propose to use the Naïve Bayes classifier for our experimental comparison of discretization methods, despite all the other types of available learners. We feel it is necessary to choose one learner with which to compare the discretization methods in order to provide for easy comparison of the discretization methods without fear that the classifier is providing some or all of any notable performance differences. We feel the Naïve Bayes classifier will provide the best comparison point due to its use to derive the most recent compared results [25] and is the learner where the benefits of discretization have been most analyzed and best displayed, as seen in [14]. While it does make assumptions about the data attributes being independent, we feel that based on [9] we can reasonably move forward that this assumption will have a minimal affect on the data, and because we are not comparing across classification methods but rather between various discretization methods used on the same classifier, that this assumption will equally affect all results if present and can thus be discounted. Thus, we are confident that the simple Naïve Bayes classifier will provide an acceptable base for our experimental comparison of the discretization methods that will now be presented.

For the purposes of our experiment, we will use the Naïve Bayes Classifier implemented in the *Waikato Environment for Knowledge Analysis*, or *WEKA* [22]. The learner is called using the wttp script in Appendix B. and is called weka.classifiers.bayes.NaiveBayes.

Chapter 3

Discretization

Chapter 3 describes a variety of data mining preprocessing methods that are used to convert continuous or numeric data, with potentially unlimited possible values, into a finite set of nominal values.

Section 3.1 describes the general concepts of discretization. Section 3.2, Section 3.3, and Section 3.4 describe a few simple discretization methods. Section 3.5 describes entropy-based approach to discretization. Section 3.6 describes proportional k-interval discretization. Section 3.7 describes an update to PKID to handle small data sets, while Section 3.8 describes Non-Disjoint Discretization. Section 3.9 describes how the creation of WPKID provided a modification to Non-Disjoint Discretization to get the benefits of decreased error rates in small data sets. We briefly discuss why we don't discuss in detail other discretization methods provided in some of the related papers on the subject in Section 3.10. Section 3.11 describes the contribution of this thesis, discretization using a randomized binary search tree as the basic storage and organizing data structure.

3.1 General Discretization

Data from the real world is collected in a variety of forms. Nominal data, such as a choice from the limited set of possible eye colors blue, green, brown, grey, usually describe qualitative values that can not easily be numerically described. Ordinal or discrete data, such as a score from a set 1, 2,..., 5 as used to rate service in a hotel or restaurant, have relationships such as "better" or "worse" between their vales, yet because these relationships can not be quantified such data are typically treated as or in similar fashion to nominal values. Numeric or quantitative values, such as the number of inches of rainfall this year or month, can take on an unlimited number of values. Figure 3.1 illustrates two such continuous attributes from a previously mentioned data set.

Instance	1	2	3	4	5	6	7	8	9	10	11	12	13	14
temperature	85	80	83	70	68	65	64	72	69	75	75	72	81	71
humidity	85	90	86	96	80	70	65	95	70	80	70	90	75	91
play	no	no	yes	yes	yes	no	yes	no	yes	yes	yes	yes	yes	no

Figure 3.1: The Continuous Attribute Values, Unsorted, of the WEATHER Data Set

Yet, while a variety of data types occur, and while many learners are often quite happy to deal with numeric, nominal, and discrete data, there are problems that may arise as a result of this mixed data approach. One instance where this can be easily illustrated is in decision tree induction. Selection of a numeric attribute as the root of the tree may seem to be a very good decision from the stand point that while many branches will be created from that root, many of those branches may contain only one or two instances and most are very likely to be pure. As a result, the tree would be quickly induced, but would result mostly in a ”lookup table” for class decision based on previous values [12]. If the training data is not representative, the training data contains noise, or a data value in the training examples that normally is representative of one class instead takes on a different class value and is induced into the tree, the created decision tree could then perform very poorly. Thus, using a continuous value when inducing trees may not be wise and should be avoided. This idea can be carried over into various learners, including the Naïve Bayes Classifier, where the assumption of a normal distribution may be very incorrect for some data sets and leaving this data in continuous form may result in an erroneous classification concept.

As a result of the threat to the accuracy and thus usability of the classifiers when continuous data is used, a method of preprocessing these values to make them usable is frequently part of the learning task. *Data discretization* involves converting the possibly infinite, usually sparse values of a continuous, numeric attribute into a finite set of ordinal values. This is usually accomplished by associating several continuous to a single discrete value. Generally, discretization transitions a quantitative attribute X_i to a qualitative, representative attribute X_i^*. It does so by associating each value of X_i^* to a range or interval of values in X_i [28]. The values of X_i^* are then used to replace the values of X_i found in the original data file. The resulting discrete data for each attribute is then used in place of the continuous values when the data is provided to the classifier.

Discretization can generally be described as a process of assigning data attribute instances to bins or buckets that they fit in according to their value or some other score. The general concept for discretization as a binning process is dividing up each instance of an attribute to be discretized into a number distinct buckets or bins. The number of bins is most often a user-defined, arbitrary value; however, some methods use more advanced techniques to determine an ideal number of bins to use for the values while others use the user-defined value as a starting point and expand or contract the number of bins that are actually used based upon the number of data instances being placed in the bins. Each bin or bucket is assigned a range of the attribute values to contain, and discretization occurs when the values that fall within a particular bucket or bin are replaced by identifier for the bucket into which they fall.

While discretization as a process can be described generally as converting a large continuous range of data into a set of finite possible values by associating chunks or ranges of the original data with a single value in the discrete set, it is a very varied field in terms of the type of methodologies that are used to perform this association. As a result, discretization is often discussed in terms of at least three different axes. The axis discussed must often is *supervised vs. unsupervised* [12,14,22]. Two other axes of frequent discussion are *global vs. local*, and *dynamic vs. static* [12, 14]. A fourth axis is also sometime discussed, considering *top-down or bottom-up* construction of the discretization structure [12].

Some discretization methods construct their discretization structure without using the class attribute of the instance while making the determination of where in the discretization structure the attribute instance belongs [12, 14, 22]. This form of discretization allows for some very simple methods of discretization, including several binning methods, and is called *unsupervised discretization*. However, a potential weaknesses exists in that two data ranges of the discretization structure in the unsupervised discretization method may have some overlap one way or another in regard to attributes with the same class attribute value being on both sides of the range division or *cut point*. If the discretization method had some knowledge of the class attribute or use of the class attribute, the cut points could be adjusted so that the ranges are more accurate and values of the same class reside within the same range rather than being split in two. Methods making use of the class attribute as part of the decision about how a value should be placed in the discretization structure are referred to as *supervised discretization* methods [12, 14, 22].

Some classifiers include a method of discretization as part of their internal structure, including the C.45 decision tree learner [14]. These methods employ discretization on a subset of the data that falls into a particular part of the learning method, for example branch of a decision tree. The data in this case is not discretized as a whole, but rather particular local instances of interest are discretized if their attribute is used as a cut point. This typically learner-internal method of discretization is called *local discretization* [12, 14]. Opposite to this is the idea of batch or *global discretization*. These methods of discretization transform all the instances of the data set as part of a single operation. Such methods are often run as external components in the learning task, such as a separate script that then provides data to the learner or even calls the learner on the discretized data.

Static discretization involves discretization based upon some user provided parameter k to determine the number of subranges created or cut points found in the data. The method then performs a pass over the data and finds appropriate points at which to split that data into k ranges. It treats each attribute independently, splitting each into its own subranges accordingly [14]. While the ranges themselves are obviously not determined ahead of time, a fixed, predetermined number of intervals will be derived from the data. *Dynamic discretization* involves performing the discretization operation using a metric to compare various possible numbers of cut point locations, allowing k to take on numerous values and using the value which scores best on the metric in order to perform the final discretization.

Finally, discretization can be discussed in terms of the approach used to create the discretization structure. Some methods start by sorting the data of the attribute being discretized and treating each instance as a cut point. It then progresses through this data and ”merges” instances and groups of instances by removing the cut points between them according to some metric. When some stop point has been reached or no more merges can occur, the substitution for values occurs. Such an approach is said to be *bottom-up discretization* [12], as it starts directly with the data to be discretized with no framework already in place around it and treating each item as an individual to be split apart. Alternatively, discretization can begin with a single range for all the values of the continuous data attribute and use some approach by which to decide additional points at which to split the range. This approach is called *top-down discretization* [12] and involves starting with the large frame of the entire range and breaking it into smaller pieces until a stopping condition is met.

Many different methods of discretization exist and others are still being created. The rest of this Chapter will discuss some of the commonly used discretization methods, provide information about some of the state-of-the-art methods, and share the new discretization method we have created. The "temperature" attribute of the WEATHER data set has been provided in sorted form in Figure 3.1 in order to provide for illustration of future methods

Instance	7	6	5	9	4	14	12	8	10	11	2	13	3	1
temperature	64	65	68	69	70	71	72	72	75	75	80	81	83	85
play	yes	no	yes	yes	yes	no	yes	no	yes	yes	no	yes	yes	no

Figure 3.2: The "temperature" Attribute Values, Sorted, of the WEATHER Data Set

3.2 Equal Width Discretization (EWD)

Equal Width Discretization, also called Equal Interval Width Discretization [14], Equal Interval Discretization , Fixed k-Interval Discretization [25], or EWD, is a binning method considered to be the simplest form of discretization [14]. EWD involves determining the minimum and maximum values of the data attribute to be discretized, and then dividing the continuous range contained by $[minval, maxval]$ into k distinct continuous subranges of size $z = \frac{maxval - minval}{k}$ where the first subrange would contain the values in the range $minval \leq x < z$, the second subrange would contain values in the range $z \leq x < 2z$, etc. The final subrange would contain values in the range $(k-l)z \leq x \leq maxval$. According to [25], k is most often set to be either 5 or 10.

As an example of how this method works, consider setting k=5 and processing the data from Figure 3.1. This method would identify a minimum value of 64 and a maximum value of 85.

Calculating $z = \frac{min - max}{k} = \frac{64-85}{5} = 4.2$. Using this result, we derive the ranges and their contents as seen in Figure 3.3

Range	[64, 68.2)	[68.2, 72.4)	[72.4, 76.6)	[76.6, 80.8)	[80.8,85]
Values	64	69	75	80	81
	65	70	75		83
	68	71			85
		72			
		72			

Figure 3.3: A Sample of EWD as Run on the ”temperature” Attribute of the WEATHER Data Set with *k*=5

This method may endure problems because of uneven distribution of instances; some subranges may contain a large number or a majority of the instances, while others may contain very few or no instances at all [22]. This is illustrated in Figure 3.3, where the range $68.2 \leq x < 72.4$ contains 5 instances - over one third of the instances in the original data - while the range $76.6 \leq x < 80.8$ contains only one instance from the data set. Additionally, outliers have been found to be capable of drastically skewing the initial range and the subranges that would result from dividing it [14].

For our experiment, we will use a Equal Width Discretization method we have called tenbins, which sets $k = 10$. This method is implemented using a script created in the GAWK programming language. The source code for our implementation can be found in Appendix C.

3.3 Equal Frequency Discretization(EFD)

Equal Frequency Discretization (EFD) is similar to Equal Width Discretization, except that the user-defined value k is used to delineate the number of instances that should fall in each of the variably-sized ranges. Each of n data instance is read and the ranges computed by counting across the values so that a break occurs after each $\frac{n}{k}$ instance. The final interval may be slightly smaller or larger depending on the size of the dataset and how the rounding of $\frac{n}{k}$ is handled. Binning using this method is referred to as histogram binning, because plotting the number of values in each subspace against the number of values in each other subspace will result in a flat histogram [22].

As an example of this method, again consider the temperature data and allow k=5. Starting from the original 14 instances, computing the number of instances per interval would result in an interval size of $\frac{14}{5}$ or 2.8 instances. To allow for a countable number of instances, we will round up to 3. In such a case, the instances might be split as illustrated in Figure 3.4.

Instances	3	3	3	3	2
Values	64	69	72	75	83
	65	70	72	80	85
	68	71	75	81	

Figure 3.4: A Sample of EFD as Run on the ”temperature” Attribute of the WEATHER Data Set with k=5

This method may result in instances with the same numeric value ending up in different bins or values that have the same class and small value differences being placed in different bins [14, 22]. In the example provided by Figure 3.4, the two instance values of 75 occur in two separate bins due to the size limitations on each bin.

3.4 Bin Logging

Bin Logging is a special case of Equal Width Discretization where the value k, is set to the maximum of 1 and $2 \cdot \log(l)$ ($max(1, 2 \cdot \log(l))$), where l is the number of distinct - unique - values of the attribute to be discretized. The Bin Logging discretization method was derived from a histogram binning method in the statistics program called S-Plus [4, 14]. This was one of the methods that was used by Dougherty et al. that showed the accuracy of Naïve Bayes Classifiers benefited from the use of discretization [14].

3.5 Entropy-based Discretization

As discussed previously, entropy is a measure of the randomness of class distribution in a split of data. The more instances of each class that exist in a given split, the higher the entropy. On the other hand, if a single class is the majority or all of a split, the entropy measure is much lower, and can reach 0 if only one class is contained in a split. The lower the entropy of a split, the higher the likelihood of a correct decision about the classification of a data point found in that split.

Examining this effect in decision trees, Fayyad and Irani proposed a method of discretization based on the entropy measure [10]. Called a entropy minimization heuristic, this approach to discretization begins by first sorting all the instance values of an attribute in ascending order. It then identifies potential *cut points* by examining the class of each instance; if the class value changes between two instances values, then the midpoint between those values can be considered as a potential cut point. After completing a list of potential cut points, it then evaluates each potential cut point using Equation 2.1. The cut point that minimizes the result of Equation 2.1 is the ”best” split and used to separate the initial set of values into two subsets. The process is then repeated recursively on each of the halves and continues until the algorithm reaches a stopping criterion.

The stopping criterion for method the method in [10] is the *minimum description length principle* (MDLP) [4, 12, 14, 22]. This principle believes that the best theory is the one that minimizes the size of the theory and any information necessary to note the exceptions to the theory [22]. It will stop discretization only if the best split at the given level results in a lower information gain than the amount of information it would cost to encode the theory being created by the new split. This can be calculated using Equation 3.1, where N is the number of instances, c is the number of classes, E is the measure of entropy for the existing split, E_1 and E_2 measure the entropy in the proposed splits, with c_1 and c_2 as the number of classes in the proposed splits.

$$gain < \frac{\log(N-1)}{N} + \frac{\log(3^c - 2) - cE + c_1 E_1 + c_2 E_2}{N} \tag{3.1}$$

If the *gain* calculated with Equation 2.3 is less than the resulting calculation, no new intervals are created from the split currently being considered.

While entropy-based discretization was originally developed for decision tree learners (see Section 2.2.1), it has been shown to be successful in working with Naïve Bayes classifiers [14]. Several authors [17, 25] note that because of its initial purpose for use with decision trees, the entropy-minimization approach typically creates nominal value attributes with few values (few splits with many instances), which while useful in the decision tree environment may not be necessary for the Naïve bayes classifier, which does not suffer from the fragmentation and over-fitting issues that cause problems for the decision trees. However, the creation of a few attributes has not been shown to decrease this method's performance; rather, in [14] found the entropy-minimization method to be the most method which most improved the accuracy of Naïve Bayes classifiers.

In attempting to run the entropy based method over the temperature data from the WEATHER dataset, the entropy-minimization heuristic found that all the data should be part of one range and did not split it beyond that range.

For our experiment, we will use the *WEKA* implementation of the entropy-minimization heuristic, a supervised, attribute filter called *Discretize*. It will be called from a BASH command-line program that can be found in Appendix E and was provided by Dr. Tim Menzies of West Virginia University's Lane Department of Computer Science (tim@menzies.us).

3.6 Proportional k-Interval Discretization

Proportional k-Interval Discretization, or PKID, is a method of discretization proposed specifically for use with a Naïve Bayes Classifier [25]. It seeks to find a balance between *discretization bias* and *discretization variance*.

Classification or probability estimation bias is error that occurs as a result of a flaw in the learned classification strategy that causes the classifier to incorrectly classify instances [23]. Discretization bias is defined as the effect that applying a particular discretization method to data for the classifier has on that classifier's classification bias. As interval sizes in discretization methods grow large, they contain less information about each individual value they contain, leading to a higher discretization bias [23,25]. A higher discretization bias leads to higher bias in classification bias.

Classification variance is error that arises from random variation in training data and random behavior of the algorithm used in learning. Classification variance can thus be said to be a measure of how sensitive an algorithm is to changes in the training data [23]. Discretization variance is the effect that a discretization method has on the classification variance of an algorithm to which its discretized data is provided. Discretization variance is lowered by increasing interval size in order to ensure that the interval contains a large number of instances.

Obviously there is a degree of conflict between minimizing discretization variance and discretization bias. Variance reduction would require fewer intervals, each large and containing many instances. On the other hand, bias reduction requires more intervals in order to maintain more distinguishing information about the values contained in each interval. This thus creates the issue of a ”trade-off” between bias and variance, where increasing or decreasing one has the opposite affect on the other.

In proposing Proportional k-Interval Discretization(PKID) for Naïve Bayes Classifiers, [25] proposes to provide equality to both discretization bias and variance. The proposed method is to function similar to Equal Interval Discretization (see Section 3.2); however, rather than having a fixed number of intervals be created (such as k=5 or k=10 in EWD), the method instead determines the number of intervals given the number of available training instances. Assuming there are N training instances for which there are known attribute values, PKID creates $s = \sqrt{N}$ intervals, each which should contain approximately $t = \sqrt{N}$ instances. These rules are restated in Equation 3.2, where N is the number of training instances, s is the number of intervals to place them in, and t is the number of training instances per interval. By providing a nearly identical number of intervals and instances per interval, PKID provides for equal consideration of size and number of intervals and thus to both discretization bias and variance.

$$\begin{aligned} s \times t &= N \\ s &= t \end{aligned} \tag{3.2}$$

PKID has certain special rules by which it performs the discretization. First, it only deals with known values for numeric attributes - unknown or missing values are simply ignored. Second, all identical values are kept in a single interval rather than being split among intervals to maintain the *sqrtN* size; thus, the actual size of each interval may vary slightly from the *sqrtN* proposed size depending on the number of identical values of an attribute. Finally, the standard size of an interval should be $\lfloor\sqrt{N}\rfloor$ (an integer value). Larger size is only allowed when identical values of an attribute occur in the interval, or if that interval is the last interval, which according to [25] have size between $\lfloor\sqrt{N}\rfloor$ and $\lfloor\sqrt{N}\rfloor$. As part of the final rule, a statement is made that no interval may be smaller than $\lfloor\sqrt{N}\rfloor$. However, in testing the implementation provided in the *WEKA* learning toolkit, an interval may be smaller if duplicate values occur that prevent enough instances from being available to fall into that interval.

In the initial experiment with PKID and Naïve Bayes classifiers, it was found to have a lower error rate than EWD and the method proposed by Fayyad and Irani dealing with entropy (see Section 3.5). It also reacts to increases in the number of examples available during training, something not done by either of the other methods studied.

A sample run of the PKID algorithm on the temperature attribute of the WEATHER data set can be found in Figure 3.5. Note that for the 14 instances in that case, $\sqrt{n} = \sqrt{14} = 3.742$. However, the output from PKID produces only 3 intervals, each with four or five instances; this number of intervals is smaller than expected but is probably caused by the duplicate values of 72 and 75 found in the data.

Range	(-infinity, 70.5)	[70.5, 77.5)	[77.5, infinity)
Instances	5	5	4
Values	64	71	80
	65	72	81
	68	72	83
	69	75	85
	70	75	

Figure 3.5: A Sample of PKID as Run on the ”temperature” Attribute of the WEATHER Data Set

While PKID has been shown to perform well with large data sets, it has also been shown to perform sub-optimally when faced with learning from a small training data set [23]. It has proposed that this is due to the equal weighting of discretization bias and variance reduction; where as in small data sets discretization variance reduction may be more important.

For our experiment, we will use the *WEKA* implementation of PKID, a filter called *PKIDiscretize*. It will be called from a BASH command-line program that can be found in Appendix D.

3.7 Weighted Proportional k-Interval Discretization (WPKID)

A solution for the PKID problem of sub-optimal performance on small training data set was proposed in the WPKID algorithm. For smaller datasets, discretization variance reduction has a bigger impact on a Naïve Bayes classifiersperformance than discretization bias [23]. WPKID weights discretization variance reduction more than bias for small training sets. This is accomplished by setting a minimum interval size of at least m instances so that the probability estimation always has a certain degree of reliability.

Going back to the equations for PKID, WPKID replaces Equation 3.2 with Equation 3.3.

$$\begin{aligned} s \times t &= N \\ s - m &= t \end{aligned} \tag{3.3}$$

Where m = 30, because, according to [23] it is commonly assumed the minimum sample space from which reliable statistical inferences should be drawn. This new approach should remove the disadvantage on smaller data sets by establishing a bias-variance trade-off that is appropriately adjusted to the size of the available training data.

An implementation of this algorithm was unavailable for public testing when this thesis was completed.

3.8 Non-Disjoint Discretization (NDD)

In suggesting Non-Disjoint Discretization, [24, 26] note that when substituting a range (a, b] for a continuous value v, more reliable results are obtained if v is closer to the middle of the interval than if the value false close to either boundary. As a result, NDD creates a series of overlapping intervals for each attribute, locating a value of v_i toward the middle of the corresponding interval (a_i, b_i].

NDD uses a sizing strategy similar to the strategy employed by PKID; specifically, it assigns s - the number of intervals - and t - the number of training instances per interval - according to

Equation 3.2. It then identifies t' *atomic intervals*, $(a'_1, b'_1]$, $(a'_2, b'_2]$, ..., $(a'_t, b'_t]$, each of size s'. Equation 3.4 describes the equations used to determine the size and number of atomic intervals.

$$\begin{aligned} s' &= \frac{s}{\alpha} \\ s' * t' &= N \end{aligned} \tag{3.4}$$

Where α is any odd number and does not vary between intervals. [24] suggests that $\alpha = 3$.

An interval is formed from each set of three consecutive *atomic intervals* such that the interval k, where $1 \leq k \leq t' - 2$, denoted by $(a_k, b_k]$ satisfies $a_k = a'_k$ and $b_k = b'_{k+2}$.

With the atomic intervals, when a value v is seen, it is assigned to the interval $(a'_{i-1}, b'_{i+1}]$ where i is the index of the atomic interval which contains v. This means v will always fall towards the middle of the interval, except when $i = 1$ in which case v is assigned to the interval $(a'_1, b'_3]$, and when $i = t'$ in which case v is assigned to $(a'_{t'-2}, b'_{t'}]$. Grouping atomic intervals to form discretization values produces overlapping intervals, resulting in the name non-disjoint.

An implementation of this algorithm was unavailable for public use when this thesis was completed. As a result, this method is not compared to the other methods described herein.

3.9 Weighted Non-Disjoint Discretization (WNDD)

WNDD [24] is performed the same way as NDD, creating atomic intervals so that values will occur, in most cases, toward the middle of the interval used to replace them. However, WNDD adds the additional restriction of the minimum interval size imposed by WPKID, specifically Equation 3.3. As explained before, the minimum interval size parameter prevents the discretization algorithm from forming intervals with too few examples for reliable probability estimation.

An implementation of this algorithm has not yet been released for use, and thus this method was not tested as part of our comparison.

3.10 Other Methods

While other methods of discretization do exist, we have listed and explained the ones commonly cited methods in some of the more recent examinations of discretization for Naïve Bayes Classifiers. Other methods such as *Iterative Discretization*, *Fuzzy Discretization*, and *Lazy Discretization* are noted in [24]; however, each is shown to perform worse (slower, less accurately) than the methods discussed herein. Additionally, their implementations are not widely available and they could not be tested as part of this experiment.

3.11 DiscTree Algorithm

In the course of reviewing other methods of discretization, we discovered that many of the methods involved sorting the values of a continuous data attribute from smallest to largest as one of the number of steps in deciding to which subrange of the data an individual instance belongs. This approach often involves using a binary sorting method for deciding the ordering of data. As the sorting approach adds additional steps on top of actually constructing the discretization structure, we wondered if there wasn't an approach that could sort the data as it built the discretization structure, thus removing an individual step.

An additional interest was whether the structure we would identify for our discretization method could be simple to explain to others. Discretization methods are often not easy to explain due to their complicated structures or the methods they use to establish the continuous ranges related to the individual discrete values created. We wanted to identify a structure that could be easily understood and explained to people who might use it. We decided to base our discretization method on the general data structure of a tree; more specifically, we proposed to implement a discretization method using a randomized binary search tree as the discretization method's underlying data structure.

To explain our choice more clearly, we'll explain the general tree data structure and build from it up to the randomized binary search tree and our method.

3.11.1 Trees

A commonly discussed data structure in computer science is that of a *tree*. We seek here to explain the basic premises of that data structure so to explain how we used a type of tree for our work. Full explanations of the tree data structure can be found in a variety of sources, including books, articles, and on the Internet. If you would like to review a comprehensive discussion of trees, we recommend a review of [20, 1085-1093].

A tree, by definition, is a connected, acyclic, undirected graph [20]. Every two points in the graph are connected by one unique path. A point, or node, in the tree can be connected to any number of other nodes in the tree so long as their connection does not create a circular path from one node to the other and back to the first. Figure 3.6 illustrates a simple tree with nodes and the paths between them.

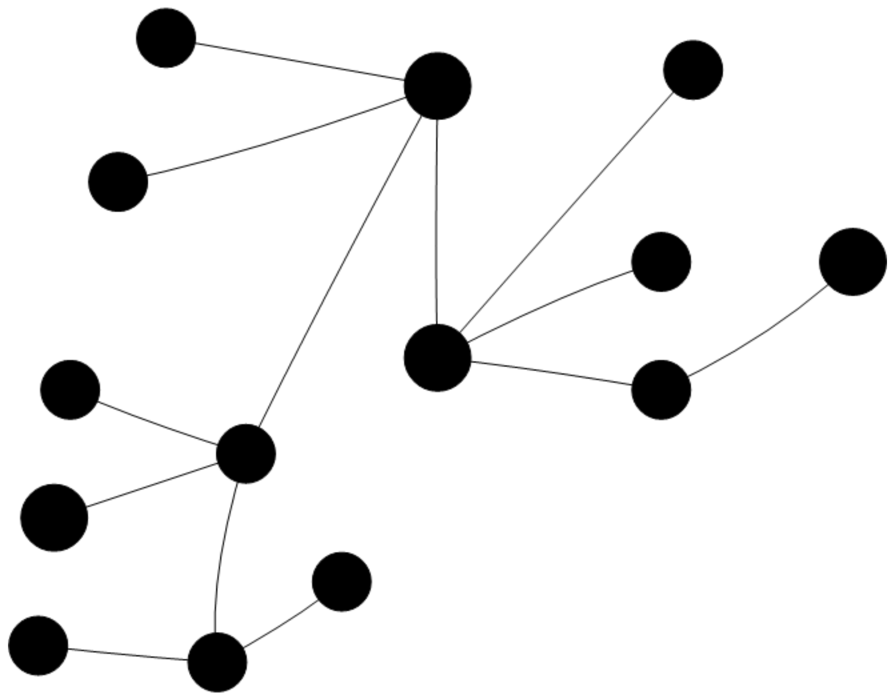


Figure 3.6: A Simple Tree

A tree can be structured in such a way that it becomes ”rooted” [20]. This occurs when one node is distinguished from the others as a starting point from which all other points can be discussed relatively. This starting point is referred to as the *root* or *root node* of the tree, and is referred to by the character *r*. A parameter of the tree T that is $root(T)$ which should store the current root node of the tree. Figure 3.7 is an example of a rooted tree, the node labeled 7 as the root.

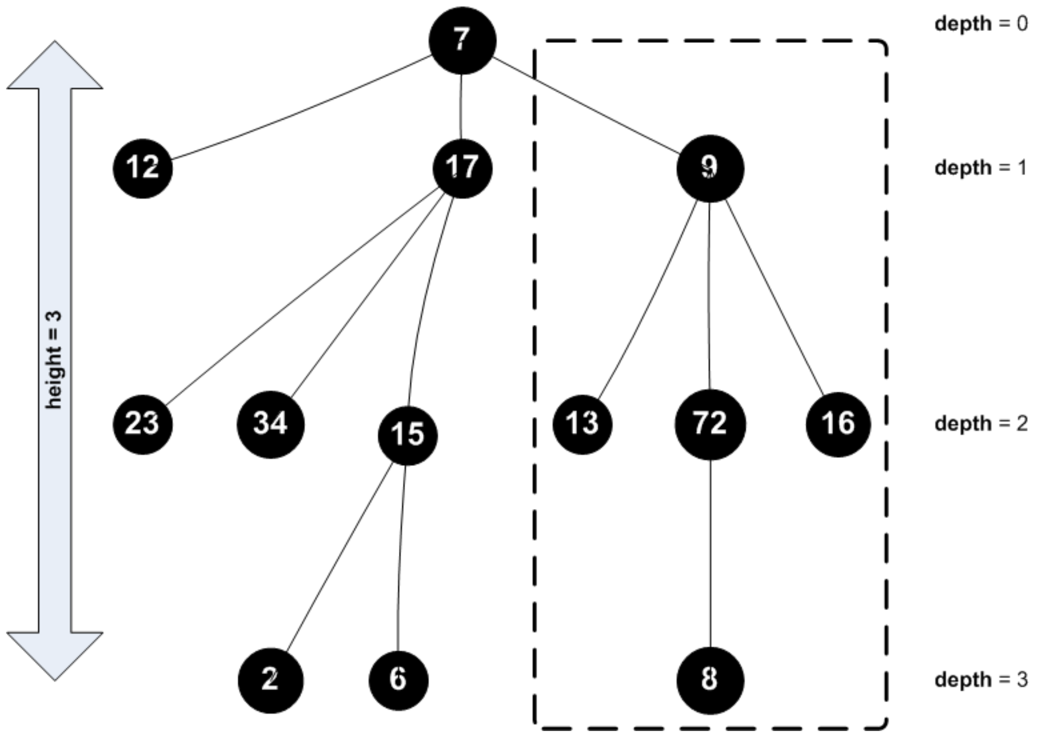


Figure 3.7: A Rooted Tree

As nodes are discussed relatively to *r*, relationships can be said to exist between the nodes. Consider a node *x* in a tree with root *r*. A node *y* that exists on the path from *r* to *x*, denoted (r,x), is called an *ancestor* of *x*. For example, in Figure 3.7 the node labeled 9 is an ancestor of the node labeled 8. Alternatively, *x* can be referred to as a *descendant* of *y* (and *y* a descendant of *x*). In Figure 3.7, the node labeled 9 is also a descendent of the node labeled 7. If node *y* is the ancestor immediately preceding node *x* in the tree - that is, there x and y share an edge then node *y* is said to be the *parent* of *x*. The node labeled 7 is also the parent of the node labeled 9 in Figure 3.7. In such a situation, node *x* is also referred to as a *child* of node *y*. Thus the node labeled 9 is a child of the node labeled 7. If two nodes share a parent, then they may be called sibling nodes, such as the nodes labeled 12, 17, and 9 in Figure 3.7. Any node *x* at the end of a path from the root, (r,x), and having no children is said to be an *external* or *leaf* node, such as the node labeled 2. Any node with children is called an *internal* node, such as the node labeled 17 in Figure 3.7.

Trees can be said to contain *subtrees*. A *subtree* is a branch of a tree that itself is a tree. For example, consider a tree rooted at r that has children x and y, each with a variety of children and other descendants. The *subtree rooted at x* would be the tree containing the descendants of x with x at the root, while the subtree rooted at y would contain all the descendants of y with y as its root. Again consider Figure 3.7. In that tree, we could discuss the subtree rooted at the node labeled 9, for example.

Parameters should exist for implementations of the tree data structure that store the parent for a given node x (we will call this parameter $parent(x)$). Calling $parent(9)$ would return the node labeled 7. Child nodes of a node x can be stored in a variety of ways to maintain ordering including the use of arrays or matrices. We are not specifically interested in this abstract level of trees in this thesis we will not discuss the specific implementations of these methods.

3.11.2 Binary Trees

A special case of traditional trees are called Binary Trees. A recursive definition describes how they function. Each Binary Tree contains either no nodes - and is thus, in fact, the *empty* or *null* tree - or contains three distinct sets of nodes: a root node, a left subtree that is also a binary tree that meets this definition, and a right subtree that is also a binary tree that meets this definition. Children of a binary tree that are themselves empty or null trees are said to be *missing* or *absent*. A sample binary tree can be found in Figure 3.8.

In regard to the child node(s) of a root node in the binary tree that are not absent, they must exist either as the left child or the right child of the root node. That is, if the root node has only one child, that child node could be either a left or right child, but is either one or the other. When the root node has two children, one node is distinguished as the left child node and the other as the right child node. Parameters should exist for each node x such that $leftchild(x)$ returns the node that is the left child of x and $rightchild(x)$ returns the right child of x. For example, from Figure 3.8, $leftchild(7)$ would return the node labeled 17, while $rightchild(7)$ would return the node labeled 9. If node x does not have either child, the parameter for that child would have the value NULL; this

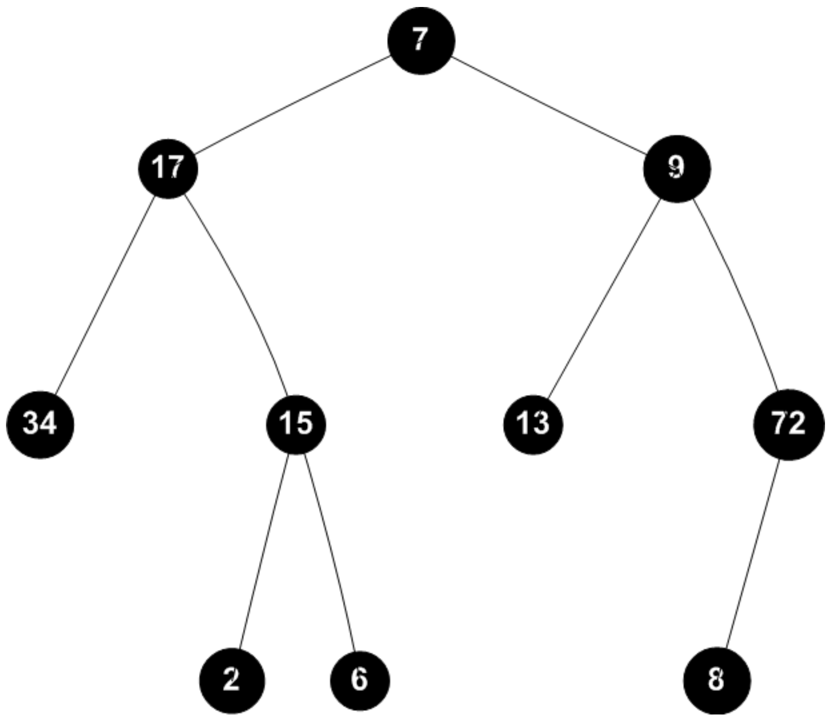


Figure 3.8: Illustrations of a Binary Tree.

would be the case with *leftchild*(34) and *rightchild*(34), each of which would return NULL.

A relevant concept in binary trees is the concept of *completeness* or *fullness*. A *complete* or *full* binary tree is one in which each node is either a leaf node or has exactly two child nodes. If one node in the tree (including the root node) has only one child, then the binary tree is not considered to be complete. For example, Figure 3.8 does not display a complete tree because the node labeled 72 has a left child but no right child.

While the position of a value in the binary tree does not denote anything in regard to its relative value to other nodes (and thus the right child of a node could be positioned there with no known relation to the left child of that node), the concept of left and right subtrees are very important for a related branch of computer science data structures, the binary search tree.

3.11.3 Binary Search Trees

Binary Search Trees (BSTs) uses the organizational structure of a binary tree to create a data structure that can allow for quick and easy sorting and searching of the data inserted into it. The binary search tree maintains the *binary search tree property*. For each node created in the tree, a key value is assigned to the node, such that for a value v to be placed in the tree, a node d is created for which $key(d) = v$. Suppose there existed a binary search tree rooted at x, and y is identified as the left child of x. Then the $key(y) \leq key(x)$. Suppose instead that y is a right child of x. Then the $key(y) \geq key(x)$. An example of a Binary Search Tree can be found in Figure 3.9 - this is, in fact, the binary tree in Figure 3.8 reorganized to meet the binary search tree property.

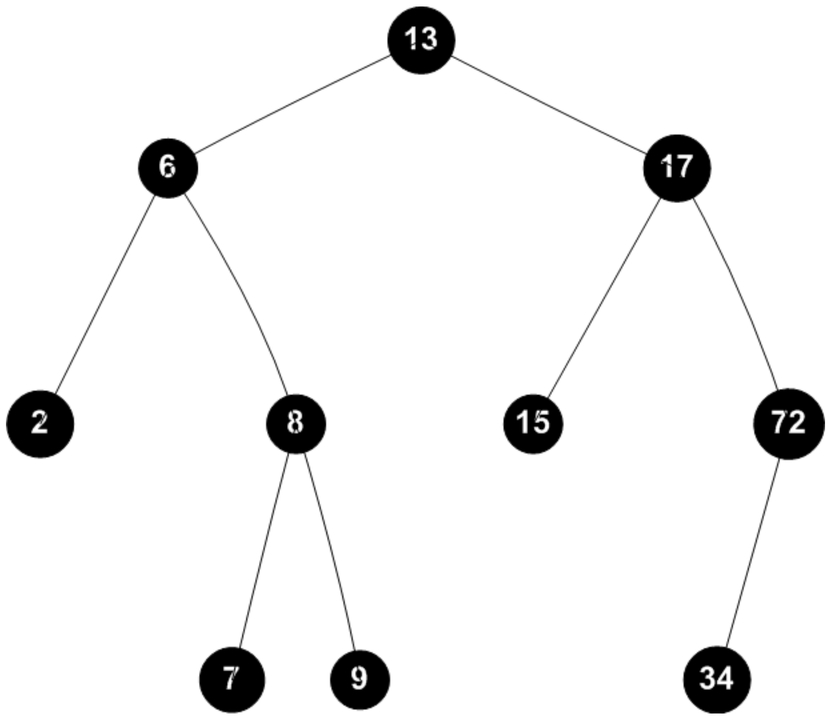


Figure 3.9: Illustration of a Binary Search Trees

A BST can be walked in numerous orders in order to retrieve the key values it stores. An *in-order tree walk* or *in-order traversal* visits each node in the tree, printing the root of each subtree of the tree after it has printed the key values of left subtree from that root and before it prints the key values of the right subtree. An *pre-order tree walk* prints the key of the root of each subtree prior to printing the values of the left and then right subtrees. Finally, a *post-order tree walk* prints the key of the root of a subtree after the key values for both the left and then the right subtree of that root are printed. Pseudo code for an in-order walk is provided in Figure 3.10.

```
Function inOrderTreeWalk(x):
        if x != NULL then
                inOrderTreeWalk(leftchild(x))
                print key(x)
                inOrderTreeWalk(rightchild(x))
```

Figure 3.10: In-Order Walk Pseudo Code

Because of their organization based on key value, the binary search tree provides a very useful mechanism for searching trees. In addition to a general search routine, whose pseudo code can be found in Figure 3.11, the binary search tree can easily find the minimum and maximum values of the data set (the leftmost and rightmost leaf nodes respectively), and also determine the preceding and succeeding key values of a node. The function names for determining the minimum, maximum, predecessor, or successor of the a node z are *treeMinimum*(z), *treeMaximum*(z), *treePredecessor*(z), and *treeSuccessor*(z) respectively.

```
Function treeSearch(x, k):
        if x != NULL OR k = key(x) then
                return x
        if k < key(x)
                searchTree(leftchild(x),k)
        else
                searchTree(rightchild(x),k)
```

Figure 3.11: BST Search Pseudo Code

Two of the most important functions of a Binary Search Tree are the INSERT and DELETE functions. As one might imagine, INSERT places a new value into the binary search tree after assigning it a key. DELETE removes node with the specified key from the tree. Both functions modify the structure of the tree but in doing so maintain the binary search tree property. INSERT is fairly straightforward, having only one special case when the tree first node is being added to the tree. Essentially, it approaches the insertion of the new node by allowing the node to fall all the way down into the tree to become a leaf node and either a left or right child node to a node that currently exists in the tree (unless the tree is empty, in which case the node is added as the root of the tree). Pseudo code for a BST INSERT can be found in Figure 3.12, where *T* is the tree (BST) into which node *z* is being inserted.

```
Function treeInsert(T, z):
        y = NULL
        x = root[T]
        while x != NULL:
                y=x
                if key(z) < key(x) then
                        x=leftchild(x)
                else
                        x=rightchild(x)
        parent(z)=y
        if y == NULL then *** SPECIAL CASE, TREE EMPTY ***
                root(T) = z
        else
                if key(z) < key(y) then
                        leftchild(x)=z
                else
                        rightchild(x)=z
```

Figure 3.12: BST INSERT Pseudo Code

While INSERT is straightforward, DELETE presents a few difficulties. Specifically, the algorithm must handle the deletion of a node that may have children and must ensure that those children are reconnected back to the tree when their parent node is removed. Essentially, we are presented with three possible situations to handle with DELETE when we wish to remove node *z*:

- *z* is a leaf node and has no children. In this case, DELETE is simple - we simply remove the node *z* and its parent's reference to it as a child.
- *z* has one child. In this case, the *parent*(z) has the child that corresponds to node *z* set equal to the child of *z* and *z* is effectively spliced out of the tree.
- *z* has two children. In this case, the successor of *z*(called node *y*), is spliced out of the tree and its information is placed in the node that holds *z*'s information.

Pseudo code for the BST's DELETE algorithm can be found in Figure 3.13.

```
Function treeDelete(T, z):
        if leftchild(z) == NULL OR righttchild(z) == NULL then
                y=z
        else
                y=treeSuccessor(z)
        if leftchild(y} != NULL then
                x=leftchild(y)
        else
                x=rightchild(y)
        if x != NULL then
                parent(x)=parent(y)
        if parent(y) == NULL then
                root(T)=x
        else
                if y == leftchild(parent(y)) then
                        leftchild(parent(y))=x
                else
                        rightchild(parent(y))=x
        if y != z then
                key(z)=key(y)
                copy node y's data into node z
        return y
```

Figure 3.13: BST DELETE Pseudo Code

While binary search trees are useful for their organization and ease of search, they can suffer from an instability issue caused by insertion in which instead of growing a full binary search tree, the insert begins with either the maximum value or minimum value at the root and only builds a right or left subtree. Such a tree would be very *unbalanced*, where a *balanced* tree, with equal or near equal height in both subtrees, is desired to minimize search times. One proposed solution to help balance the tree is the *randomzied binary search tree*.

3.11.4 Randomized Binary Search Trees

Randomized Binary Search Trees (RBSTs) take their name from the randomization of the INSERT function used to add new data to the tree. In adding randomization to this approach, we allow that, given an existing tree that contains n instances, that the current instance x has a $\frac{1}{n}$ chance of becoming the root node of the tree; what is more, even if the node is not selected to be the root node of the overall tree, this random insert is attempted at the root of each subtree that x belongs in, with a chance to insert at that root equal to $\frac{1}{size\ of\ subtree}$. If the new node does not ever get selected as a root of the tree or its subtrees, then it is simply added as a node in the tree as it would be using the INSERT function described in Figure 3.12.

Randomized Binary Search Tree provide some protection against unbalanced problems that can occur as a result of doing a BST insert on sorted data or on data that starts with the minimum or maximum value of the tree. Because of the random nature of the tree, it is expected that the tree should be somewhat balanced or at least not so heavily skewed as the ordinary BST when provided with the data that causes the deficiency. It is because of this protection that we select to use the randomized binary search tree as our base data structure when creating the DiscTree algorithm.

Our implementation of the RBST insert is based upon the idea of always inserting a new node as a leaf of the tree and ”bubbling up” the inserted value to its decided place in the tree using rotations of keys in the tree [19]. This approach reorganizes the tree without the complications of trying to split and rejoin the tree as has been proposed in [16]. Pseudo code for our implementation can be found in the portion of Figure 3.14. Function *rbstinsert* draws a random number to determine whether to insert the new value at the root of the current subtree; if it decides in favor of inserting it at the root, it calls function *insertR* to insert at the root of the current subtree by rotating the new value up from its initial insertion point as a leaf node to its place as the root of the decided-upon subtree. If the new node is not found to be the root of a subtree, it is inserted simply as a leaf node as it would in the binary search tree.

3.11.5 DiscTree

The DiscTree Algorithm makes use of the organization of the randomized binary search tree by using such trees to create a storage and discretization structure for each attribute. For each of the attributes of the data set, a separate RBST is created and each value from the data set is inserted into its attribute’s RBST. At specific intervals, the tree is garbage-collected to ensure that it does not grow boundlessly. Additionally, as the DiscTree algorithm becomes aware of additional data - beyond the current expected training size - it increases the allowable size of that attribute’s tree to allow for more possible nodes in the tree after a garbage-collect. The specific details of the algorithm will be explained here.

The DiscTree algorithm performs discretization in two passes. The first pass builds randomized binary search trees for each (numeric) attribute in the provided data set. In order to build the tree, each instance is inserted into the tree in the form of a node that contains the information depicted

```
Function insertR(h, x):
        if h == NULL then
                return new Node(x)
        if(x.key() == h.item.key())
                updateNode(h)
        if(x.key() < h.item.key())
                leftchild(h) = RBSTInsert(leftchild(h), x)
                h = rotateR(h)
        else
                rightchild(h) = RBSTInsert(rightchild(h), x)
                h = rotateL(h)
        return h

Function rbstinsert(h, x):
        if h == NULLthen
                return newNode(x)
        if(x.key() == h.item.key())
                updateNode(h)
        if( rand() * h.all() < 1.0)
                return insertR(h, x)
        else
                if(x.key() < h.item.key())
                        rbstinsert(leftchild(h), x)
                else
                        rbstinsert(rightchild(h), x)
        h = insertT(h, x)

Function rotateR(h):
        x = leftchild(h)
        leftchild(h) = rightchild(x)
        rightchild(x) = h
        return x

Function rotateL(h):
        x = rightchild(h)
        rightchild(h) = leftchild(x)
        leftchild(x) = h
        return x

Function newNode(x):
        y.item = x
        leftchild(y) = NULL
        rightchild(y) = NULL
        y.count = 1
        //initialize other values as necessary
        return y

Function updateNode(h):
        h.count = h.count + 1
        return h
```

Figure 3.14: RBST INSERT Functions Pseudo Code

in figure *DTNode*. Repeated values that already exist in the tree increment the counters of the existing node; the repeated values can also result in moving the node up in the tree if it is decided during the *rbstinsert* of the value that it should be higher in the tree than it currently is. Missing values (depicted in our input as question marks (?))are ignored; it is expected that the classifier will handle any such values.

During insertion, a garbage collection mechanism may be called. Garbage collection occurs, if the tree has reached or exceeded its maximum allowable size, after every 35 non-missing instances are read. This is value we have chosen to use and have not examined a particular justification for. It works for the data we are dealing with but might be an area of future concentration.. Garbage collection occurs as follows: Tree nodes are ”visited” in Breadth-First Search [20] order. Each visited node is counted, and when the count meets the maximum size limit, any additional nodes are then pruned from the tree. This Breadth-First approach is taken to help preserve the target of a balanced/full search tree.

The tree begins with a maximum allowable size of 7 nodes. This size is allowed to grow as the tree is provided with more and more training instances. Specifically, after each 75 nodes of training data, we allow the tree size to grow by a power of 2 such that, if the initial number of allowable nodes is initially 7, then the allowable number of nodes $7+2^{(2+1)} = 7+2^3 = 7+8 = 15$. More generally, the tree size increases as seen in Equation 3.5, allowing that *maxsize* is equal to the current maximum tree size and 2^i is the current largest power of 2 that can be subtracted from *maxsize* without a negative result.

$$maxsize = maxsize + 2^{(i+1)} \tag{3.5}$$

Which, when applied to the value 15, would result in:

$$maxsize = 15+2^{(3+1)} = 15+2^4 = 15+16 = 31$$

Again, 75 instances was a value we chose without any experimental testing to determine an optimum value. This may be another area that could be improved upon with further study. We have attempted some brief experiments on tree-size start points and found performance to be very similar for trees of size 7 and 15 and that those models tended to perform at about the same level.

Future work could be conducted in that area as well to determine a best starting size and growth rate.

After each attribute value of each instance is read and inserted into that attribute's RBST, the discretization method begins a second pass. It begins it by garbage collecting any trees that are larger than their maximum allowable size. It then determines which nodes and subtrees contain enough instances within them to be used a discretization points for the data set. It makes this determination by examining the nodes to determine which nodes contain at least $\sqrt{N}$ instances, where N is the number of non-unknown data values seen for the attribute. This is akin to Webb's PKID discretization method [25], which requires each bin of their discretization method to contain at least $\sqrt{N}$. However, beyond that justification we have identified no other reason to support the $\sqrt{N}$ instances per bin imposition; this may, like the number of instances between garbage collect and tree size, be an area where this algorithm could be improved in the future. When the nodes and subtrees have been determined, they are substituted into the data in place of the original possible values for the data. This is done by examining each instance again, and replacing each attribute's value in that instance with the node or subtree which most closely matches the value of the attribute value of the instance that meets the $\sqrt{N}$ requirement. When this is done, the data is considered discretized. As with previous examples, we provide the discretized values for the temperature attribute of the WEATHER data set. This information is available in Figure 3.16

The DiscTree Algorithm as a whole can be described by the pseudo code in Figure 3.15

We implemented the DiscTree algorithm using a GAWK script in order to implement the data processing and randomized binary search tree data structure. The source code is located in Appendix A.

```
Pass 1:
For each instance:
        For each attribute:
                Insert each instance's value into that  attribute's RBST
                If specified number of records seen, allow tree size to grow
                If specified number of records seen, Garbage Collect.

Pass 2:
Garbage Collect each RBST tree to ensure size requirement before discretization
For each attribute:
        Replace the attribute's value type with the list of tree nodes that meet size requirement
For each instance:
        For each attribute:
                Find the tree value that:
                        (a) most closely matches the current value and
                        (b) meets size requirement
                Replace the current value with the name of the chosen node
```

Figure 3.15: DiscTree Algorithm Pseudo Code

Node	temperature#2	temperature#5	temperature#7	temperature#9	temperature#14
Instances	2	3	3	2	4
Values	64	68	71	75	80
	65	69	72	75	81
		70	72		83
					85

Figure 3.16: A Sample of the DiscTree Algorithm as Run on the ”temperature” Attribute of the WEATHER Data Set

Chapter 4

Experiment

Chapter 4 describes the experimental approach used to compare the discretization techniques. Section 4.1 explains the data used to perform this comparison. Section 4.2 explains the cross-validation method used to generate the results from the learner. Section 4.3 explains the different measures used to compare the effects of discretization on the Naïve Bayes classifier. Section 4.4 explains the Mann-Whitney approach used to compare the results of each discretization method against its competitors.

4.1 Test Data

In order to perform the comparison between the various discretization methods, we sought to use a variety of data sets of both small - under 1000 instances - and large data sets. We also sought data sets that used varying numbers of classes. We have decided to use 24 data sets from the UCI machine learning repository [1] which contain varying numbers of numeric and discrete attributes but each has a discrete class attribute making it an easy candidate for classifier learning. The data sets we have elected to use and their describe features can be found in Figure 4.1.

Data Set	*Instances*	*Attributes*	*Numeric*	*Nominal*	*Classes*
hayes-roth	132	5	1	4	3
iris	150	4	4	0	3
hepatitis	155	19	6	13	2
wine	178	13	13	0	3
flag	194	28	10	18	4
imports-85	205	25	15	10	5
audiology	226	69	0	69	24
breast-cancer	286	9	0	9	2
heart-h	294	13	6	7	5
heart-c	303	13	6	7	5
ecoli	336	7	7	0	8
auto-mpg	398	7	5	2	3
wdbc	569	30	30	0	2
soybean	683	35	0	35	19
credit-a	690	15	0	15	2
breast-cancer-wisconsin	699	9	0	9	2
diabetes	768	8	8	0	2
vowel	990	13	10	3	11
segment	2310	19	19	0	7
splice	3190	61	0	61	3
kr-vs-kp	3196	36	0	36	2
waveform-5000	5000	40	40	0	3
mushroom	8124	22	0	22	2
letter	20000	16	16	0	26

Figure 4.1: Data Sets Used for Discretization Method Comparison. The attributes column refers to the number of non-class attributes that exist in the data set; the data set would have one more nominal attribute if the class were counted.

4.2 Cross-Validation

Cross-validation is a statistical method that divides data into a fixed *n folds*, or partitions, of approximately the same size [22], for example, three folds each containing one-third of the data. The first fold is then used for classifier testing, while the remaining *n-1* folds are used for classifier training. The process is repeated so that each of the *n* folds is used for testing. The resulting process is known as *n*fold cross-validation. This process can be combined with *stratification*, or random sampling to create the folds so as to guarantee that each class is represented evenly in both the testing and training sets, which results in approximately the same class distribution across folds. The result is called *stratified nfold cross-validation*.

Tenfold stratified cross-validation is a standard way of measuring the error rate or accuracy of a learning scheme on a particular dataset [11, 12, 22]. In this form of cross-validation, the data is divided randomly into ten subsets of approximately the same size. The classifier is run ten times. Each run, the classifier uses a different subset as the testing set while the other subsets are combined to use as the training set. This results in ten different accuracy/error estimates for the learner that can be averaged to provide an overall accuracy/error estimate.

In order to ensure that variation in randomization or classifiers do not cause invalid or unreliable results in a tenfold cross-validation, it is recommended in [22] that the tenfold cross-validation process be repeated ten times, resulting in running the learning algorithm 100 times on a dataset. The results are then averaged to obtain the accuracy/error rate for the learning algorithm.

In our experiment, we will use this *ten by tenfold cross-validation* (10 x 10fold) process to compare the results of each discretization method. We will first run each discretization method for this experiment on the data, creating a total of five data sets to test - the original data set, the data set discretized using PKID, the data set discretized using the entropy-based method, the data set discretized using the tenbins equal width discretization method, and the data set discretized using the DiscTree algorthim. Each of the five data sets is then used with a Naïve Bayes Classifier to complete a ten x ten fold cross validation. The results are then compared as described in 4.4.

It is worth noting that one of the results we seek to compare against is that from [25]. The authors of that paper, and in all of their other papers comparing discretization methods that we are aware of ([23,24,27,28]), use a ten by threefold cross-validation (10x3). We put forth that we are provide more runs for comparison using our ten by tenfold cross-validation approach and that, as a result, we can be confident that our results are comparable to the previous results because they used fewer runs, each of which would have a larger affect on the average result used to compare discretization methods.

We implemented cross-validation as a BASH script. The source code for is available for review in Appendix B.

4.3 Classifier Performance Measurement

As part of the cross-validation script we have created, we have provided a means of calculating several discretization method/classifier *performance measures*. These methods are described as follows:

Allow that *A, B, C, D* denote, in order, the true negatives, false negatives, false positives, and true positives found by a binary detector. True negative denotes when the detector does not identify the class of an instance as the desired class because it is truly not of the desired class; false negatives, on the other hand, denote when the detector does not identify the class of an instance as the desired class despite the fact that it is of the desired class. False positives denote when the detector believes an instance does belong to the desired class but, in fact, it does not. Finally, true positives are those instances where the detector determines the class of the instance is of the desired class and it is, in fact, of that class. These definitions are illustrated in Figure 4.2.

		instance of desired class?	
		no	yes
identified as	no	A (true negative)	B (false negative)
desired class?	yes	C (false positive)	D (true positive)

Figure 4.2: A Tabular Explanation of *A, B, C, & D*

The following measures can be calculated from the values of *A, B, C, & D*. Each of the results falls between 0 and 1; when multiplied by 100, the scores produced can be discussed as percentages between 0 and 100%.

$$accuracy = acc = \frac{A+D}{A+B+C+D} \tag{4.1}$$

$$probability\ of\ detection = pd = recall = \frac{D}{B+D} \tag{4.2}$$

$$probability\ of\ false\ alarm = pf = \frac{C}{C+A} \tag{4.3}$$

$$not\ probability\ of\ false\ alarm = !pf = 1 - pf \tag{4.4}$$

$$precision = prec = \frac{D}{D+C} \tag{4.5}$$

$$balance = bal = 1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}} \tag{4.6}$$

Equation 4.1 describes the *accuracy* of a classifier (or in our case, a discretization method/classifier pair). By this, we mean the percentage of cases in which the classifier accurately identifies the true positives and true negatives; that is, it correctly identifies when an instance belongs to the desired class and when it does not.

Equation 4.2 describes the probability of detection of the desired class. It is also referred to as *recall*. This equation measures how much of the desired class is found and correctly identified.

Equation 4.3 describes the probability of false alarm, where an instance is identified as belonging to the desired class when it, in fact, does not. Equation 4.4 identifies the proportion of cases that were not false alarms. This measure is most often used, as opposed to the Equation 4.3 as it allows for positive comparison of the methods; that is, the desired score *!pf* would be a high score like those desired for *pd* or *acc*, where as the desire when dealing with Equation 4.3 is to find the method with the smallest result.

Equation 4.5 describes the precision of the method used. The precision of a method identifies the proportion of instances which were identified as being of the desired class that are actually of the desired class. A high precision is desired over a lower one.

Equation 4.6 describes the method's balance between *pd* and *pf*. While the desired balance would be *pd*= 1 and *pf*= 0, this does not occur in practice, with methods instead often taking on *risk-adverse* balances - with a high *pf* in order to ensure a high *pd* - or *cost-adverse* balances - where in order to keep from having too many costly alarms, the method accepts a low *pd* in order to ensure a low *pf*. Higher *bal* scores mean that the method performs closer to the ideal balance and thus a higher score is desired.

4.4 Mann-Whitney

Many papers comparing results of classification compare those results using standard parametric statistical tests, such as the *t*-test, to determine whether the results for one classifier differ significantly from the results of other classifiers. In the case that the test does find statistical significance, a classifier may be said to significantly better or worse than another classifier. The problem with this approach is that it assumes a specific distribution for the data being examined for significance; for example, in the case of a *t*-test, that assumption is that the distribution takes on a curve similar to *N(0,1)*, the normal curve between 0 and 1 [6]. Unfortunately, data can not be guaranteed to fit a specific curve, and outliers in the data can cause the mean and standard deviation measured used for parametric measures to become greatly skewed from the actual bulk of the data. As a result, it is considered prudent that nonparametric tests be used to compare results.

The problem of comparatively assessing *L* learners (or *L* discretization methods on one learner) run on multiple sub-samples of *D* data sets has been extensively studied in the data mining community. T-tests that assume Gaussian distributions are becoming deprecated. Demsar [7] argues that non-Gaussian populations are common enough to require a methodological change in data mining.

Demsar [7] offers a definition of *standard methods* in data mining. In his study of four years of proceedings from the *International Conference on Machine Learning*, he found that the standard method of comparative assessment were t-tests over some form of repeated sub-sampling such as cross-validation (see Section 4.2). Such t-tests assume that the distributions being studied are Gaussian. However, as [7] warns, results can be highly non-Gaussian when the presence of one or more outliers skews the distribution of the results. Thus calculations performed by such tests using Gaussian assumptions should be avoided.

After reviewing a wide range of comparisons methods, [7] advocates the use of the 1945

Wilcoxon [21] signed-rank test that compares the ranks for the positive and negative differences (ignoring the signs). Writing five years earlier, [2] commented that the Wilcoxon test has its limitations, specifically that the sample sizes must be the same to be compared by the Wilcoxon test. Demsar's report offers the same conclusion. To fix this problem, [7] augments Wilcoxon with the Friedman test.

One test not studied by [7] is Mann and Whitney's 1947 modification [15] to Wilcoxon rank-sum test (proposed along with his signed-rank test). We prefer this test since:

- The Mann-Whitney U test does not require that the sample sizes are the same. So, in a single U test, a learner L_1 can be compared to all its rivals.
- The U test does not require any post-processing (such as the Friedman test) to conclude if the median rank of one population (the results of learner L_1) is greater than, equal to, or less than the median rank of another (the results of learner $L_2, L_3, .., L_x$). In our specific case, we will be applying the U-test to a discretization method/learner pair, although all other aspects remain the same.

Non-parametric tests such as the U test proposed by Mann and Whitney [15] mitigate the outlier problem. The U test uses *ranks*, not precise numeric values. For example, if method A generates $N_1 = 5$ values $\{5,7,2,0,4\}$ and method B generates $N_2 = 6$ values $\{4,8,2,3,6,7\}$, then these sort as follows:

Samples	A	A	B	B	A	B	A	B	A	B	B
Values	0	2	2	3	4	4	5	6	7	7	8

Figure 4.3: Sorted Values of Method A and Method B

On ranking, averages are used when values are the same:

Samples	A	A	B	B	A	B	A	B	A	B	B
Values	0	2	2	3	4	4	5	6	7	7	8
Ranks	1	2.5	2.5	4	5.5	5.5	7	8	9.5	9.5	11

Figure 4.4: Sorted, Ranked Values of Method A and Method B

Note that, when ranked in this manner, the largest value (8 in this case) gets the same rank even if it was ten to a hundred times larger. Because of this, such rank tests are less susceptible to large outliers.

Figure 4.5 shows the U test for the two treatments A and B discussed in Figure 4.3 and Figure 4.4. The test concludes that these methods are not statistically different (at the 95% significance level). As defined in Figure 4.5, this test counts the *wins*, *ties*, and *losses* for A and B (where A and B are single or groups of methods). In most cases, we will seek the method which *wins* most.

The sum and median of A's ranks is

$$\begin{aligned} sum_A &= 1+2.5+5.5+7+9.5 = 25.5 \\ median_A &= 5.5 \end{aligned}$$

and the sum and median of B's ranks is

$$\begin{aligned} sum_B &= 2.5+4+5.5+8+9.5+11 = 40.5 \\ median_B &= 6.75 \end{aligned}$$

The U statistic is calculated from $U_x = sum_x - (N_x(N_x+1))/2$:

$$U_A = 25.5 - 5*6/2 = 10.5$$
$$U_B = 40.5 - 6*7/2 = 19.5$$

These can be converted to a Z-curve using:

$$\begin{aligned} \mu &= (N_1 N_2)/2 &&= 516.4 \\ \sigma &= \sqrt{\frac{N_1 N_2 (N_1+N_2+1)}{12}} &&= 5.477 \\ Z_A &= (U_A - \mu)/\sigma &&= -0.82 \\ Z_B &= (U_B - \mu)/\sigma &&= 0.82 \end{aligned}$$

(Note that Z_A and Z_B have the same absolute value. In all case, these will be the same, with opposite signs.)

If $abs(Z) < 1.96$ then the samples A and B have the same median rankings (at the 95% significance level). In this case, we add one to both $ties_A$ & $ties_B$. most cases of this work (accuracy, pd, balance, precision), *higher* values are better since we are comparing errors. Hence:

- If $median_A > median_B$ add 1 to both $wins_A$ & $losses_B$.
- Else if $median_A < median_B$ add 1 to both $losses_A$ & $wins_B$.
- Else, add 1 to both $ties_A$ & $ties_B$.

Figure 4.5: An example of the Mann-Whitney U test.

Chapter 5

Experimental Results

Chapter 5 shares and briefly explains the results of the experimental comparison of the discretization methods used with a Naïve Bayes classifier. Section 5.1 describes a comparison of possible implementations of the DiscTree algorithm Section 5.2 describes the results of the experimental comparison between the selected version of the DiscTree algorithm and the other selected discretization methods. Section 5.3 summarizes the results in this section.

5.1 DiscTree Variant Selection

When the concept of DiscTree was originally created, we proposed three possible cases for how it would function. Specifically, there were two features that we questioned including in the final method to be compared. They were:

- Allowing the DiscTree algorithm to perform its organization and substitution on all data in a data set; that is, performing substitutions for both nominal and numeric data. We will refer to this as nominal attribute discretization.
- Performing the Garbage Collection described in Section 3.11.5.

In order to make the decision about which methodology to use, we created scripts for three possibilities:

- That both features would be included. Specifically, a script called disctree2 was implemented that contained both Garbage Collection and nominal attribute discretization.

- That only the Garbage Collection feature would be included. Specifically, a script called disctree3 was created that implemented Garbage Collection. However, it only would only perform discretization on numeric attributes.
- That neither Garbage Collection nor nominal attribute discretization would be used. We created a script called disctree4 that only allowed for numeric attributes to be discretized. The discretization trees could grow infinitely large; however, they still had the constraint of requiring a minimum number of instances in or below a subtree to use the root node of that subtree as a substitution point.

5.1.1 Accuracy Results

The results of a U-test comparing the three method's accuracy across all the results from each data set are found in Figure 5.1. The results find that *disctree3* ties *disctree4* in terms of accuracy, but performs better than *disctree2*. This tends to suggest that adding nominal attribute discretization to disctree would cause it to perform worse.

key	*win*	*loss*	*ties*	*win-loss*
disctree3	1	0	1	1
disctree4	0	0	2	0
disctree2	0	1	1	-1

Figure 5.1: overall for acc

In addition to the U-test results across all the data sets, we have isolated the results for each data set, sorted the resulting accuracy scores, and provided them in the plots found in Figure 5.1.1 to Figure 5.1.1. The *x-axis* of each of these graphs is the number of data points generated as results for that data set, with each point having one plot and being associated with an accuracy score. The *y-axis* of each of the graphs is the accuracy percentage of the method for plotted point.

While the methods perform similarly on many of the data sets, there are some stand out results that are worth noting:

- The script *disctree3*, in green on the plots, clearly wins in four data sets (auto-mpg, hayes-roth, imports-85, and iris) while only clearly losing in one data set (flag). In the case of the flag data set, its appearance at the end of each segment denotes it had lower accuracy than the other methods at several points in the data set.

- The script *disctree2*, in red on the plots, clearly wins in only two data sets (diabetes and flag), while it loses in several (heart-h, hepatitis, mushroom, soybean, and wine).

- The script *disctree4*, in blue on the plots, has no clear wins, and only one loss where has lower performance measures than either of the other scripts in the diabetes data set.

It is clear from these results that if accuracy were the only measure of contention, that the clear winner and choice of methods should be *disctree3*. However, we have put forth five methods of comparison, and will review the results for each before making a decision.

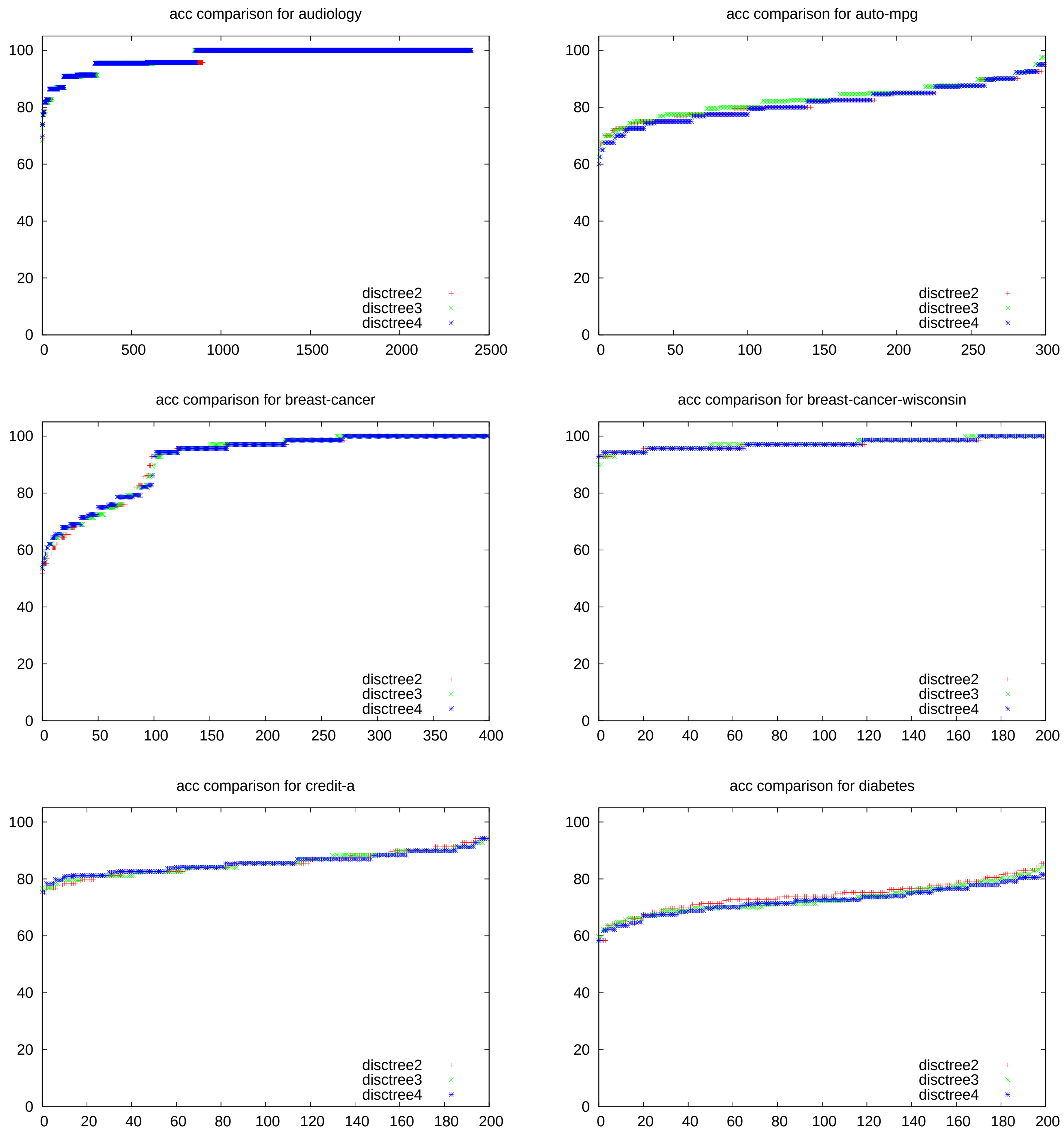


Figure 5.2: Plots of the Accuracy Scores, Sorted by Value

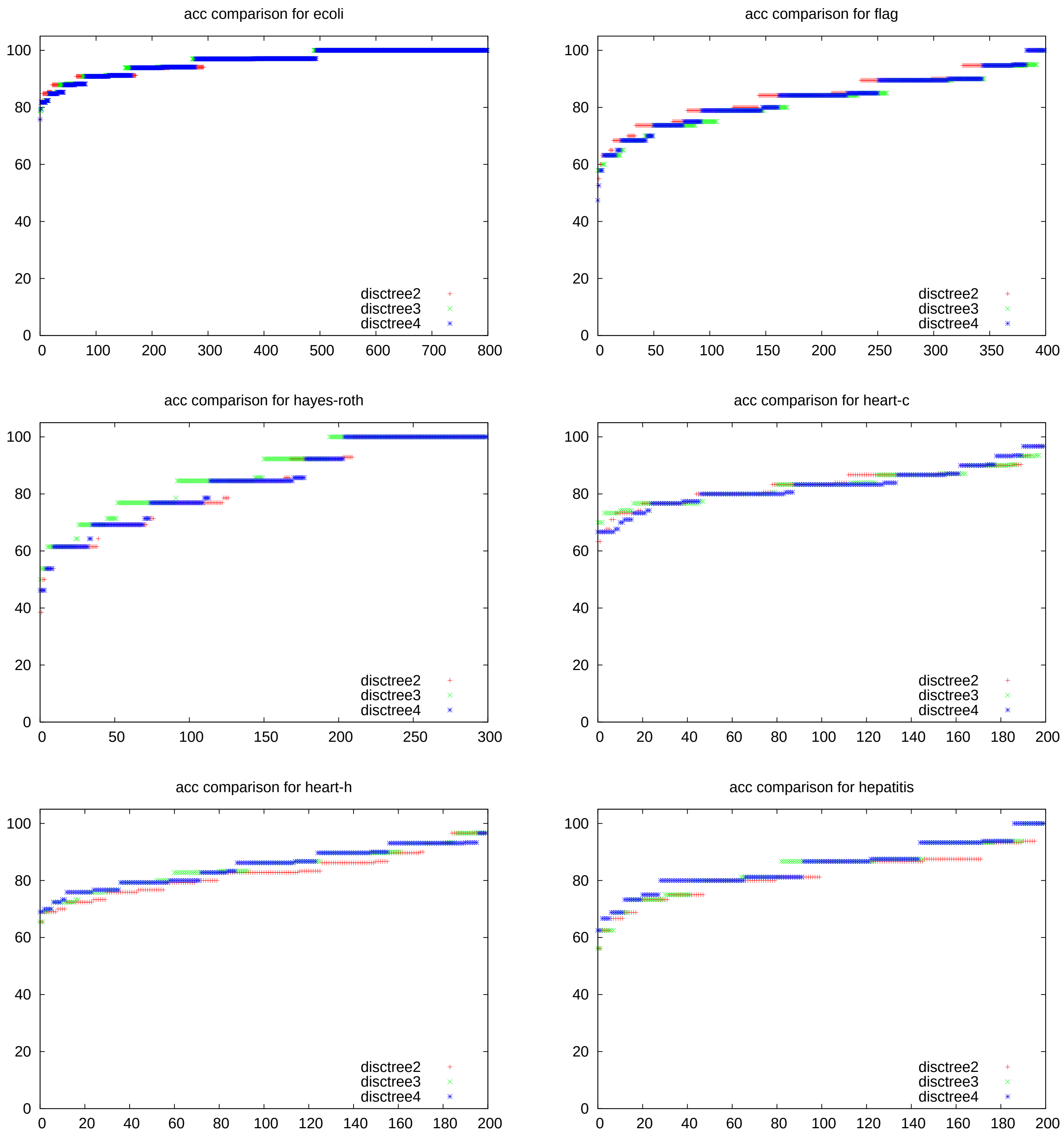


Figure 5.3: Plots of the Accuracy Scores, Sorted by Value

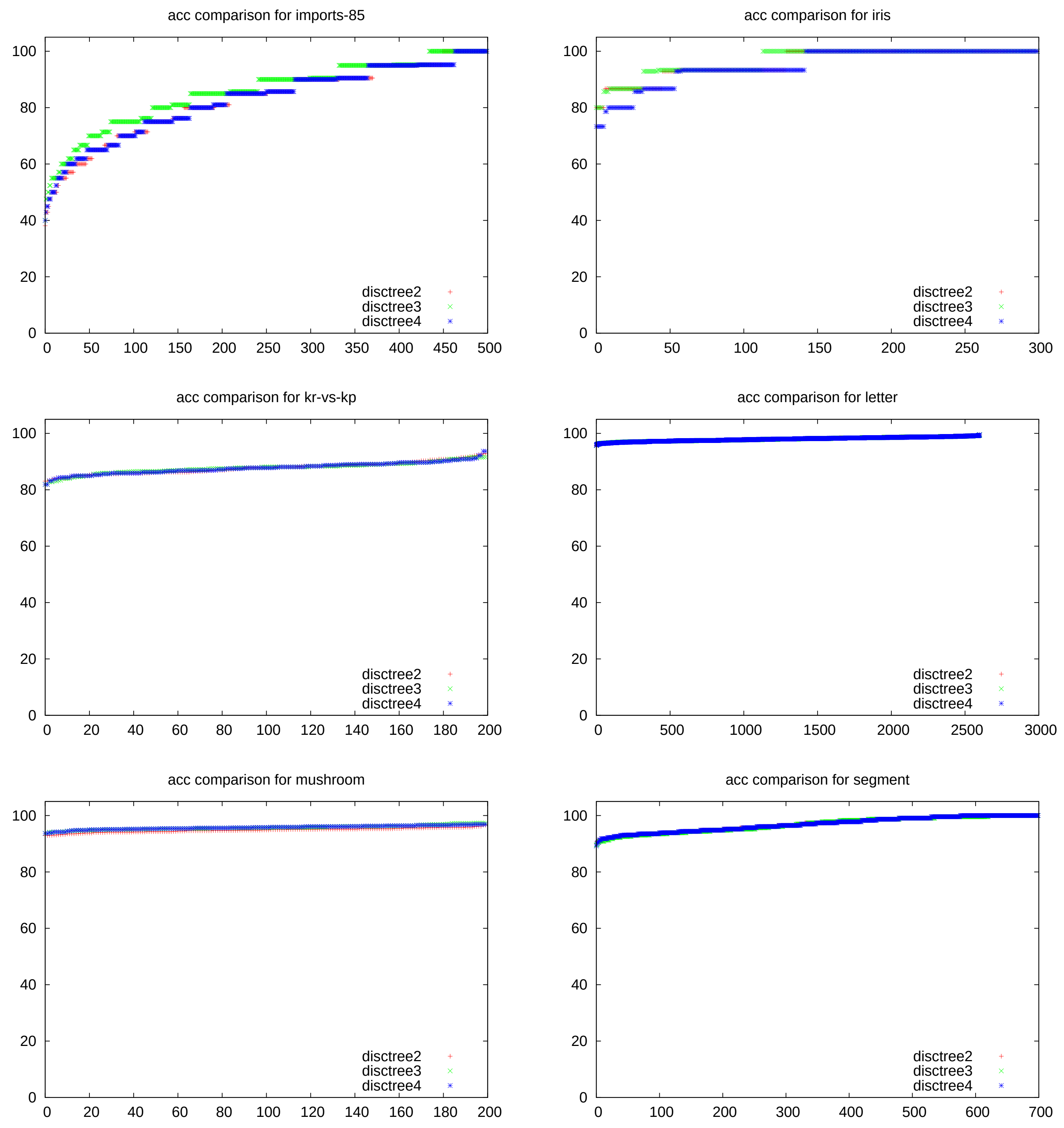


Figure 5.4: Plots of the Accuracy Scores, Sorted by Value

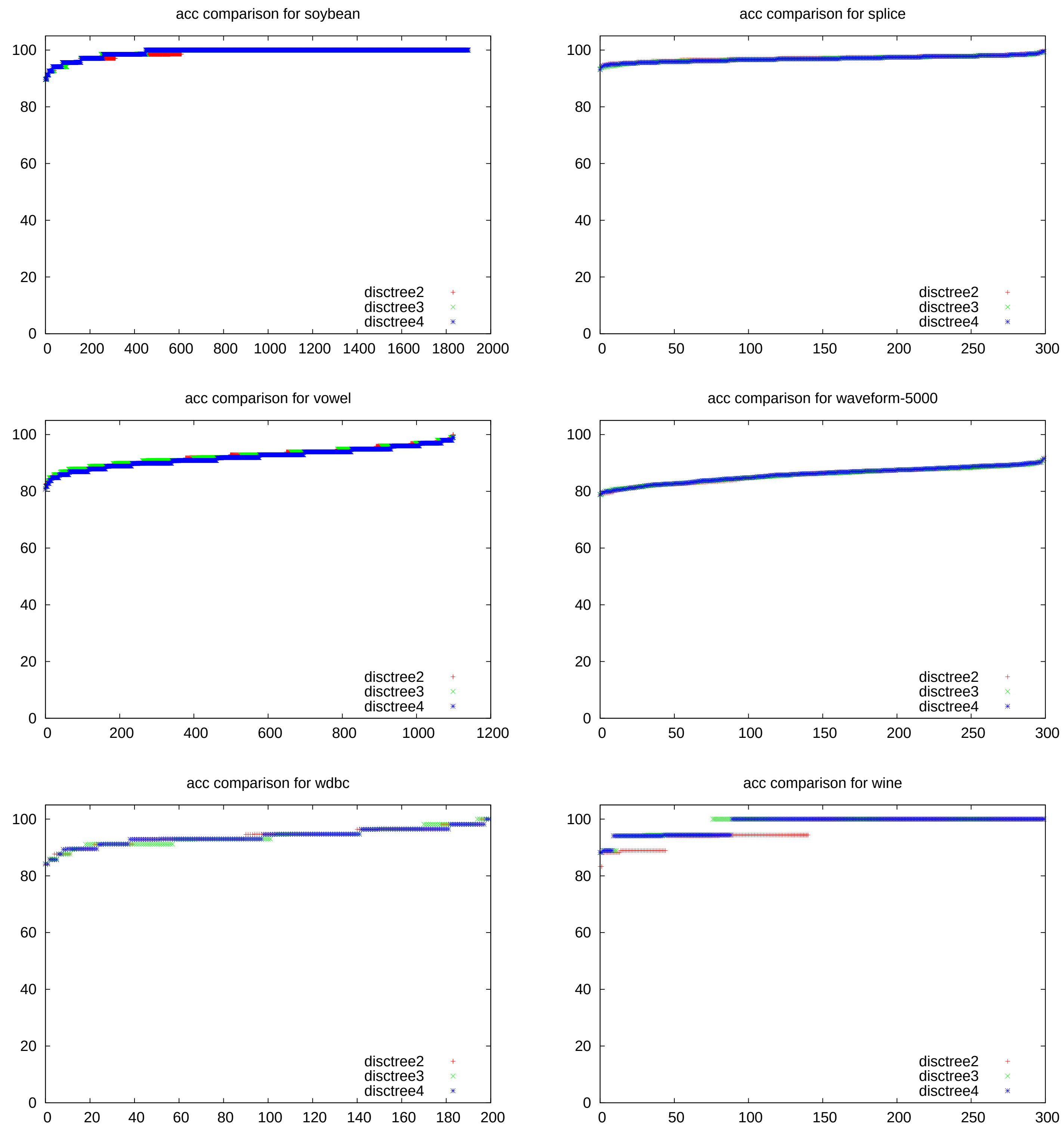


Figure 5.5: Plots of the Accuracy Scores, Sorted by Value

5.1.2 Balance Results

The results of a U-test comparing the three method's balance across all the results from each data set are found in Figure 5.6. The results find that the methods all tie and the U-test finds no clear winner in regard to which disctree script provides the best balance result.

key	*win*	*loss*	*ties*	*win-loss*
disctree4	0	0	2	0
disctree3	0	0	2	0
disctree2	0	0	2	0

Figure 5.6: overall for bal

In addition to the U-test results across all the data sets, we have isolated the results for each data set, sorted the resulting accuracy scores, and provided them in the plots found in Figure 5.1.2 to Figure 5.1.2. The *x-axis* of each of these graphs is the number of data points generated as results for that data set, with each point having one plot and being associated with one balance score. The *y-axis* of each of the graphs is the balance percentage achieved for the discretization method of plotted point.

While the methods perform similarly on many of the data sets, there are some stand out results that are worth noting:

- The script *disctree3*, green, clearly wins in six data sets (auto-mpg, ecoli, hayes-roth, hepatitis, imports-85, and iris) while only clearly losing in one data set (segment).
- The script *disctree2*, red, clearly wins in four data sets (audiology, diabetes, flag, and vowel), while it loses just as often (heart-h, hepatitis, soybean, and wine).
- The script *disctree4*, blue, has no clear wins, and has six losses (auto-mpg, diabetes, flag, imports-85, iris, vowel) where it achieves lesser balance scores than the other methods for many of the same data points.

While the U-test finds no clear winner between the methods, examining the data plots leads to the assertion that if balance were to be used to choose the DiscTree algorithm's representative method that *disctree3*, with the highest number of wins and fewest losses, would be the ideal choice.

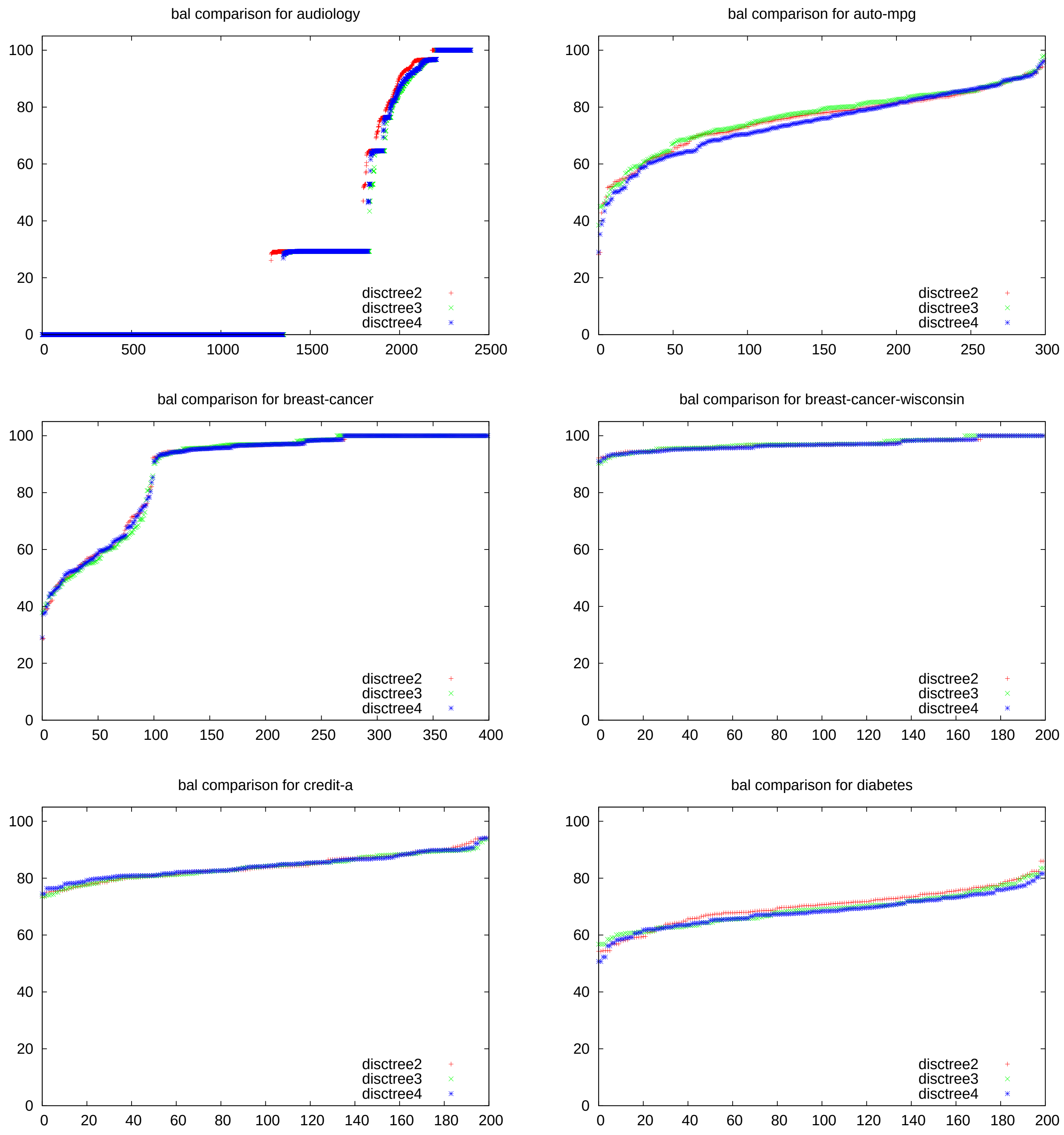


Figure 5.7: Plots of Balance Scores, Sorted by Value

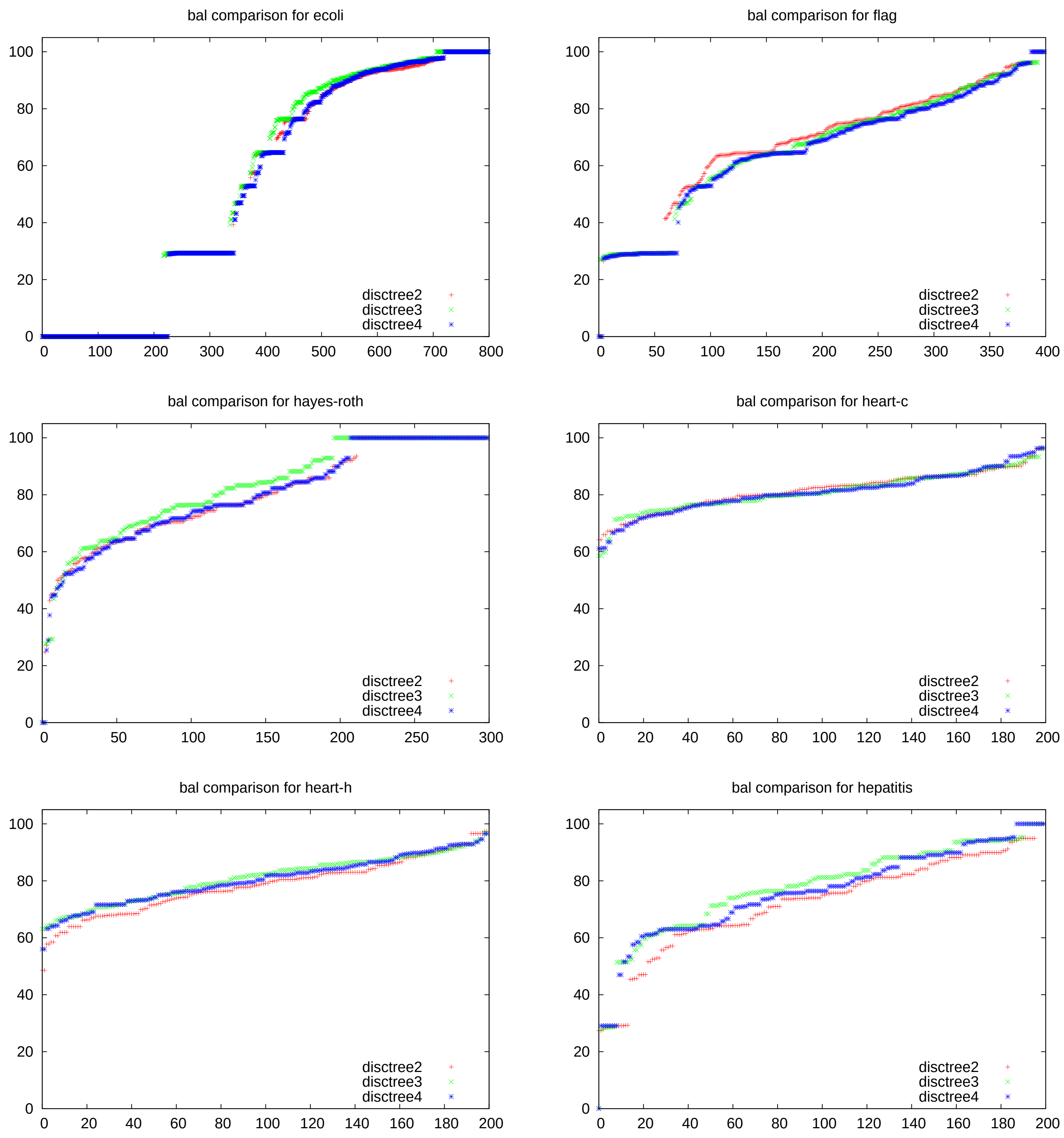


Figure 5.8: Plots of Balance Scores, Sorted by Value

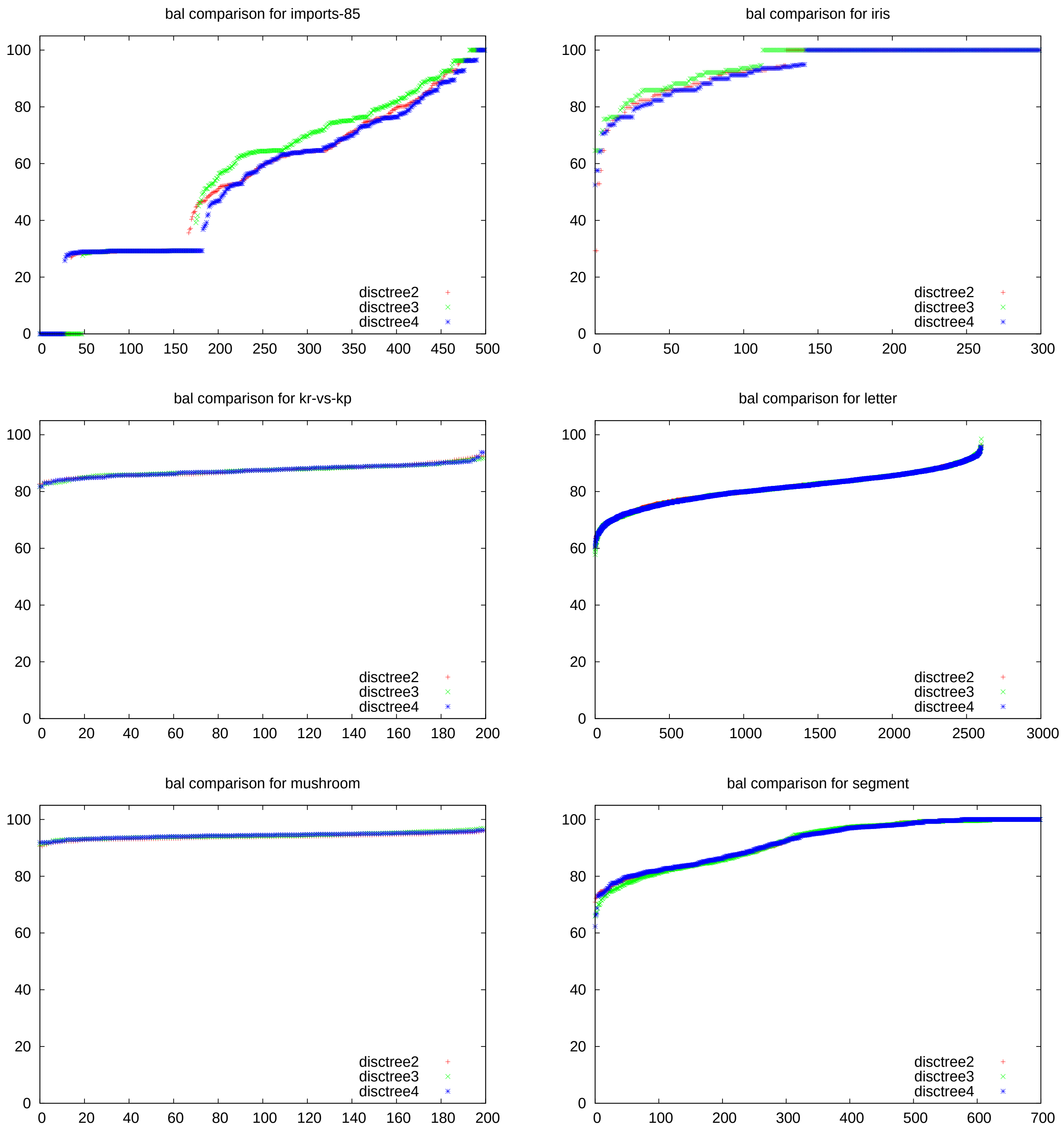


Figure 5.9: Plots of Balance Scores, Sorted by Value

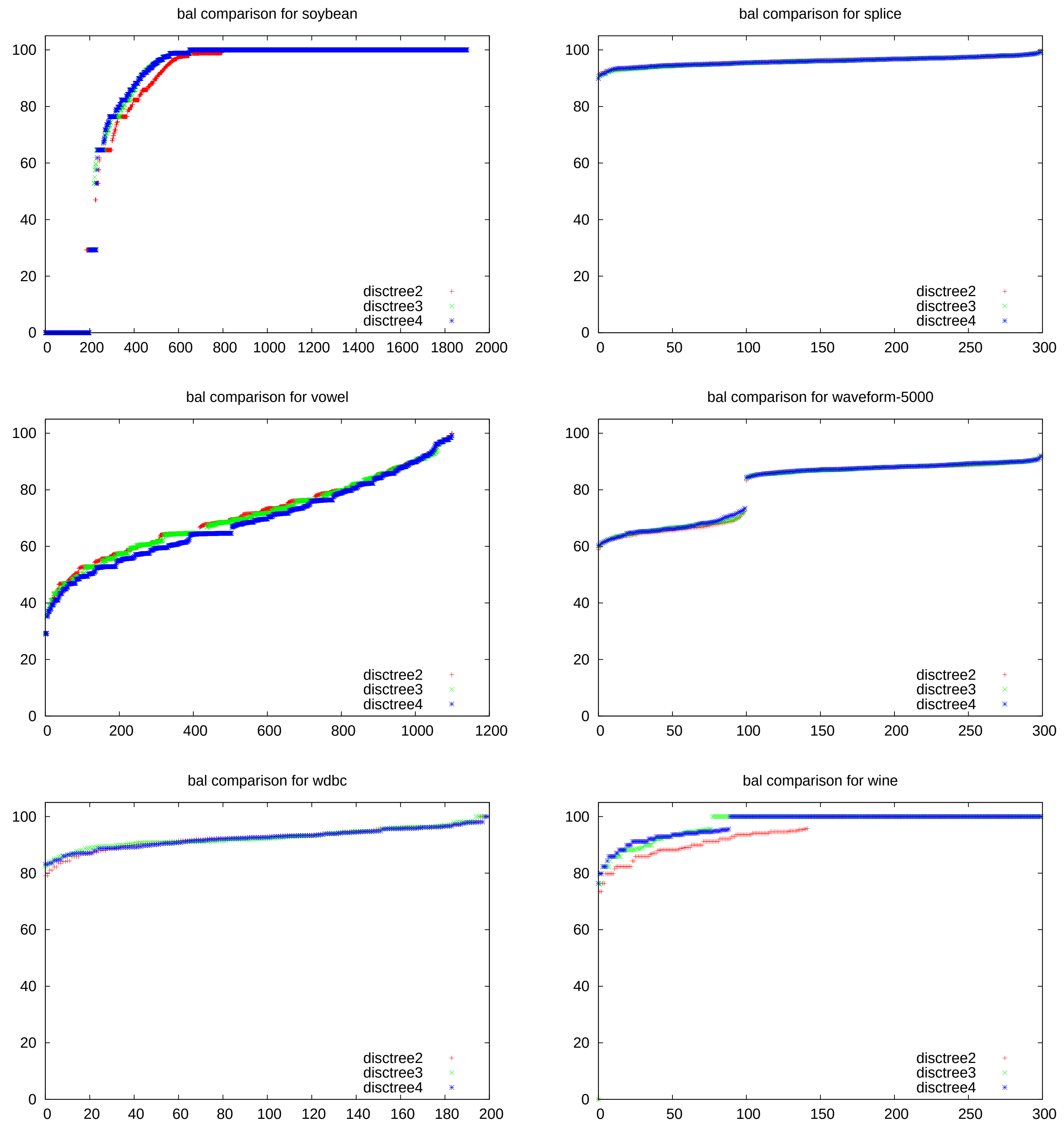


Figure 5.10: Plots of Balance Scores, Sorted by Value

5.1.3 Precision Results

The results of a U-test comparing the three method's precision across all the results from each data set are found in Figure 5.11. The results find that the methods all tie and the U-test finds no clear winner in regard to which disctree script provides the best precision result.

key	*win*	*loss*	*ties*	*win-loss*
disctree4	0	0	2	0
disctree3	0	0	2	0
disctree2	0	0	2	0

Figure 5.11: overall for prec

In addition to the U-test results across all the data sets, we have isolated the results for each data set, sorted the resulting accuracy scores, and provided them in the plots found in Figure 5.1.3 to Figure 5.1.3. The *x-axis* of each of these graphs is the number of data points generated as results for that data set, with each point having one plot and representing one precision score. The *y-axis* of each of the graphs is the precision percentage achieved for the discretization method of plotted point.

While the methods perform similarly on many of the data sets, there are some stand out results that are worth noting:

- The script *disctree3*, green, clearly wins in five data sets (auto-mpg, ecoli, hayes-roth, imports-85, and iris) while only clearly losing in one data set (flag).
- The script *disctree2*, red, clearly wins in four data sets (audiology, diabetes, flag, and vowel), while it loses in six (ecoli, hepatitis, heart-h, mushroom, soybean, and wine).
- The script *disctree4*, blue, had one clear win in the segment data set, and has four losses (diabetes, auto-mpg, hayes-roth, vowel) where it achieves lower precision scores than the other methods for many of the same data points.

While the U-test finds no clear winner between the methods, examining the data plots leads to the assertion that if precision were to be used to choose the DiscTree algorithm's representative method that *disctree3*, with the highest number of wins and fewest losses, would be the ideal choice.

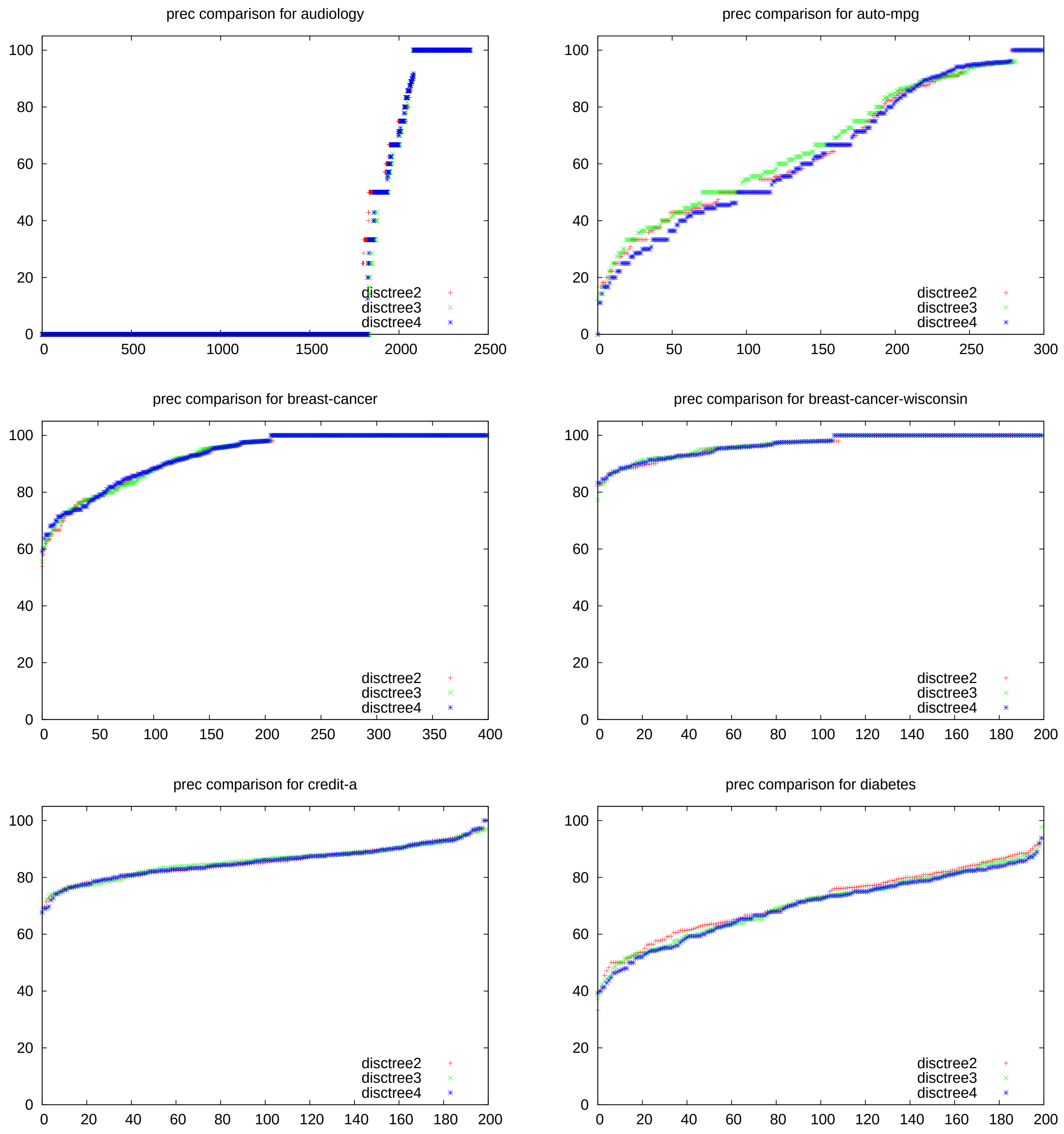


Figure 5.12: Plots of Precision Scores, Sorted by Value

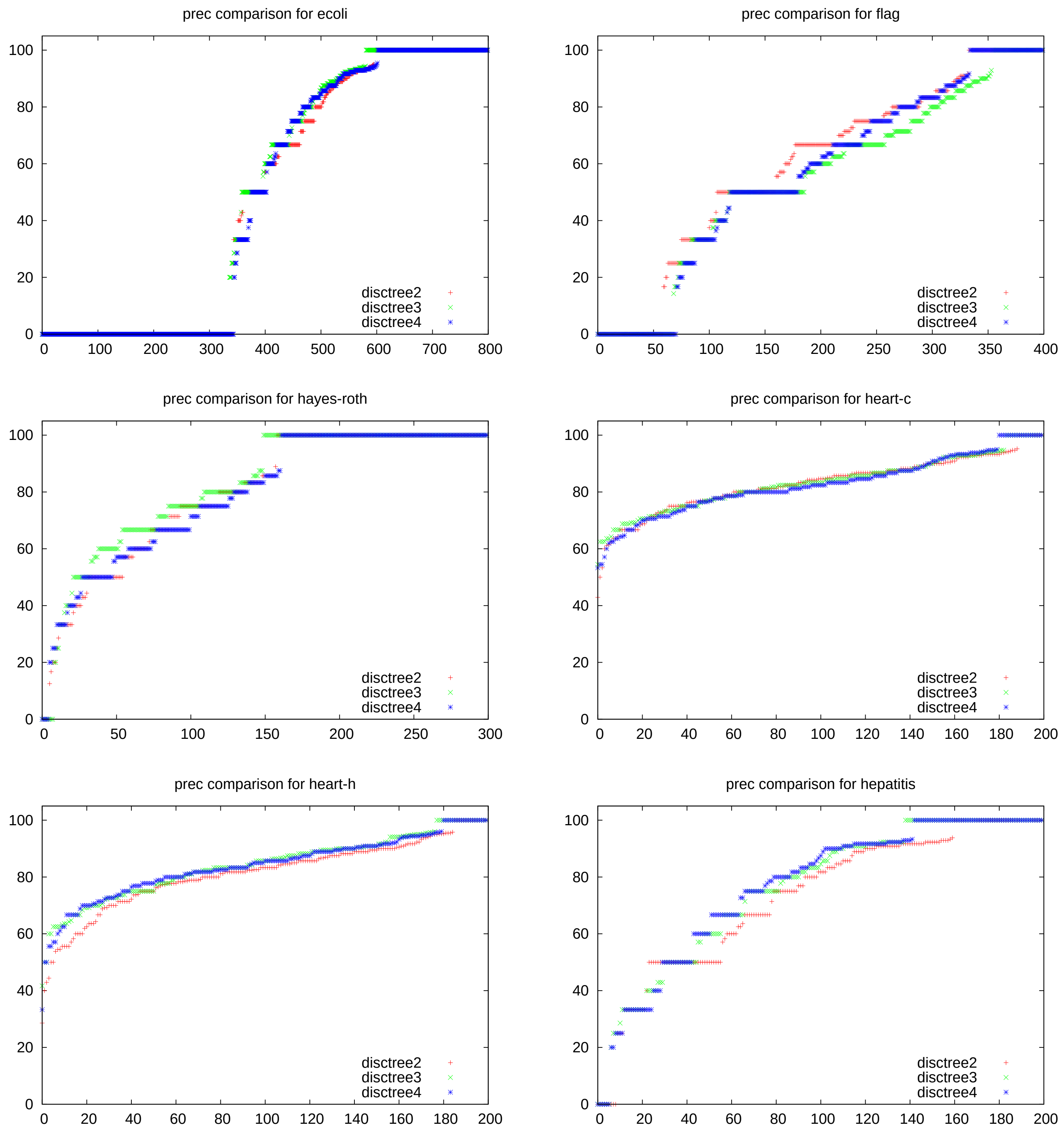


Figure 5.13: Plots of Precision Scores, Sorted by Value

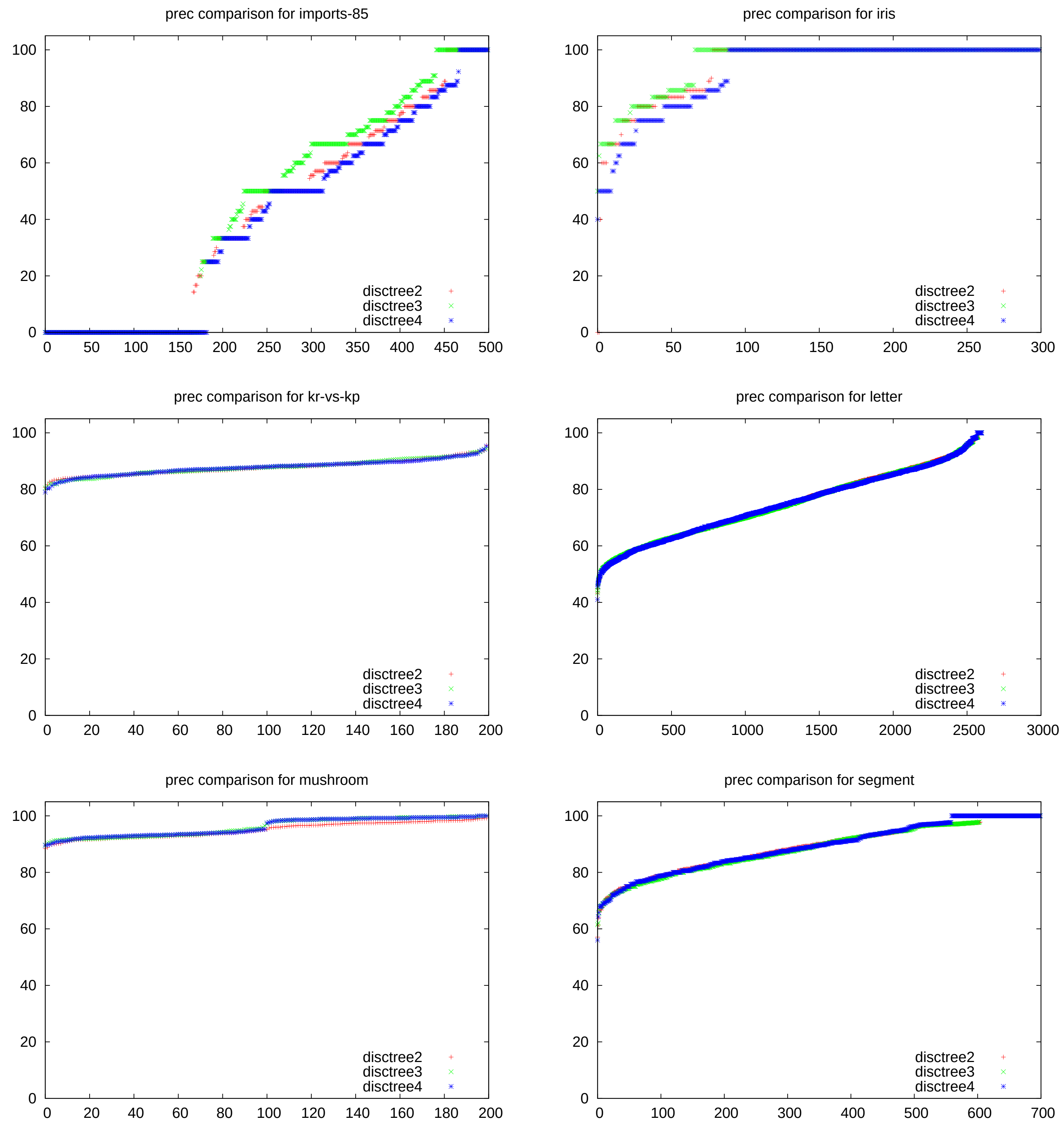


Figure 5.14: Plots of Precision Scores, Sorted by Value

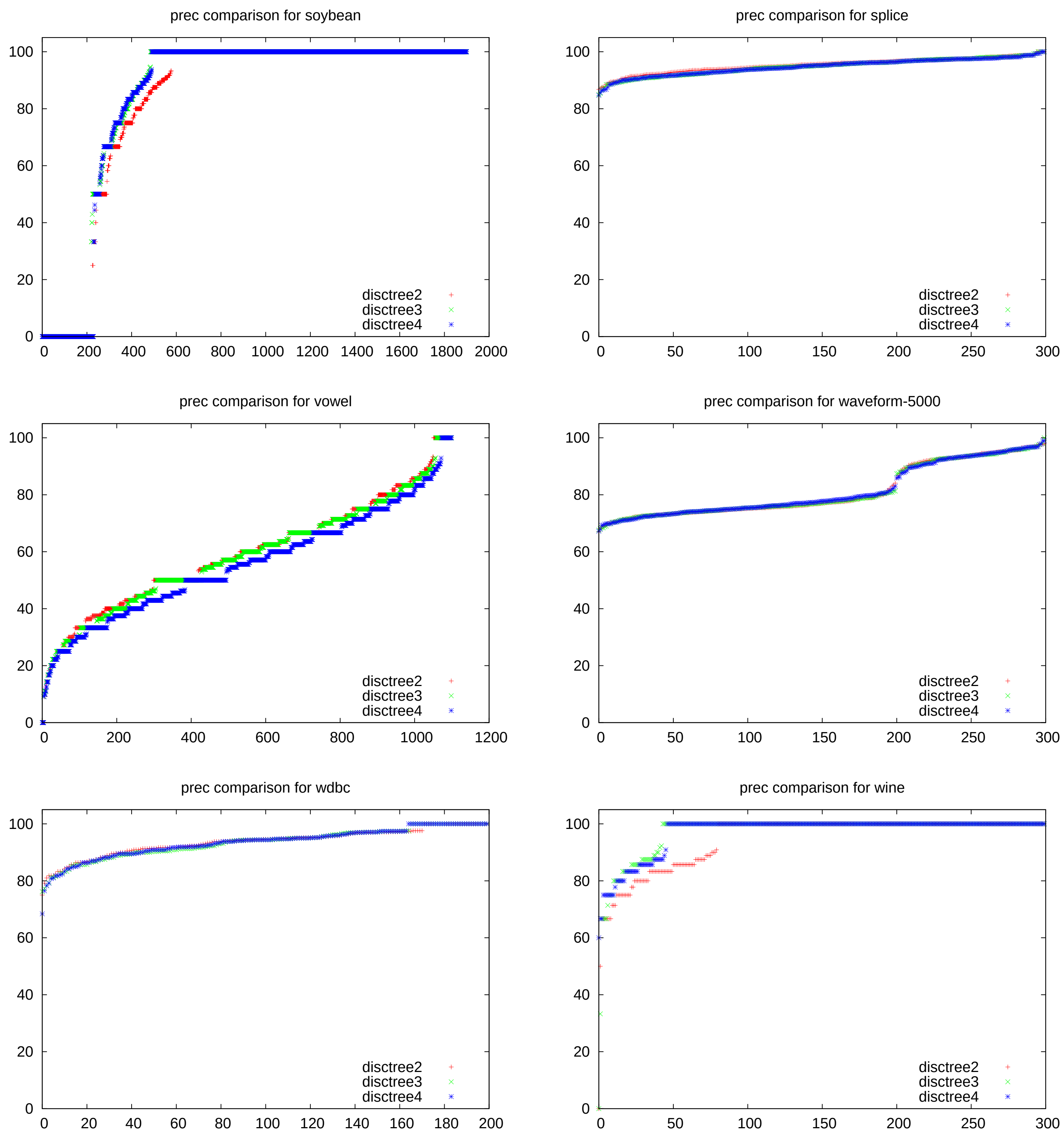


Figure 5.15: Plots of Precision Scores, Sorted by Value

5.1.4 Probability of Detection Results

The results of a U-test comparing the three method's probability of detection across all the results from each data set are found in Figure 5.11. The results find that the methods all tie and the U-test finds no clear winner in regard to which disctree script provides the best probability of detection result.

key	*win*	*loss*	*ties*	*win-loss*
disctree4	0	0	2	0
disctree3	0	0	2	0
disctree2	0	0	2	0

Figure 5.16: overall for pd

In addition to the U-test results across all the data sets, we have isolated the results for each data set, sorted the resulting accuracy scores, and provided them in the plots found in Figure 5.1.3 to Figure 5.1.3. The *x-axis* of each of these graphs is the number of data points generated as results for that data set, with each point having one plot and representing one probability of detection score. The *y-axis* of each of the graphs is the probability of detection percentage achieved for the discretization method of plotted point.

While the methods perform similarly on many of the data sets, there are some stand out results that are worth noting:

- The script *disctree3*, green, clearly wins in seven data sets (auto-mpg, ecoli, hayes-roth, heart-h, hepatitis, imports-85, and iris) while only clearly losing in two data sets (audiology, segment).
- The script *disctree2*, red, clearly wins in four data sets (audiology, diabetes, flag, and vowel), while it loses in seven (breast-cancer, ecoli, heart-h, hepatitis, mushroom, soybean, and wine).
- The script *disctree4*, blue, had one clear win in the wine data set, and has three losses (diabetes, flag, vowel) where it achieves lower probability of detection scores than the other

methods for many of the same data points.

While the U-test finds no clear winner between the methods, examining the data plots leads to the assertion that if probability of detection were to be used to choose the DiscTree algorithm's representative method that *disctree3*, with the highest number of wins and fewest losses, would be the ideal choice.

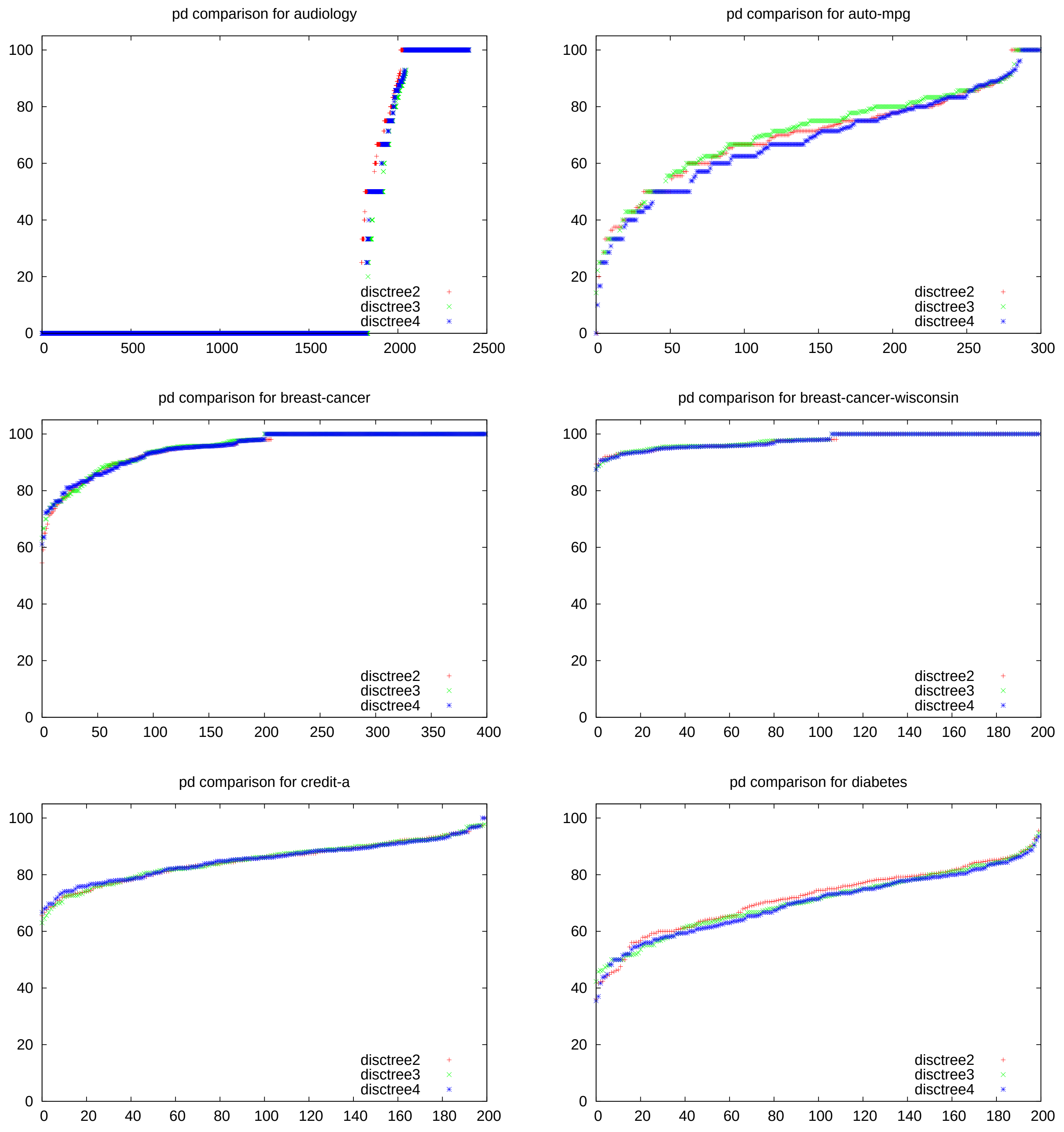


Figure 5.17: Plots of Probability of Detection Scores, Sorted by Value

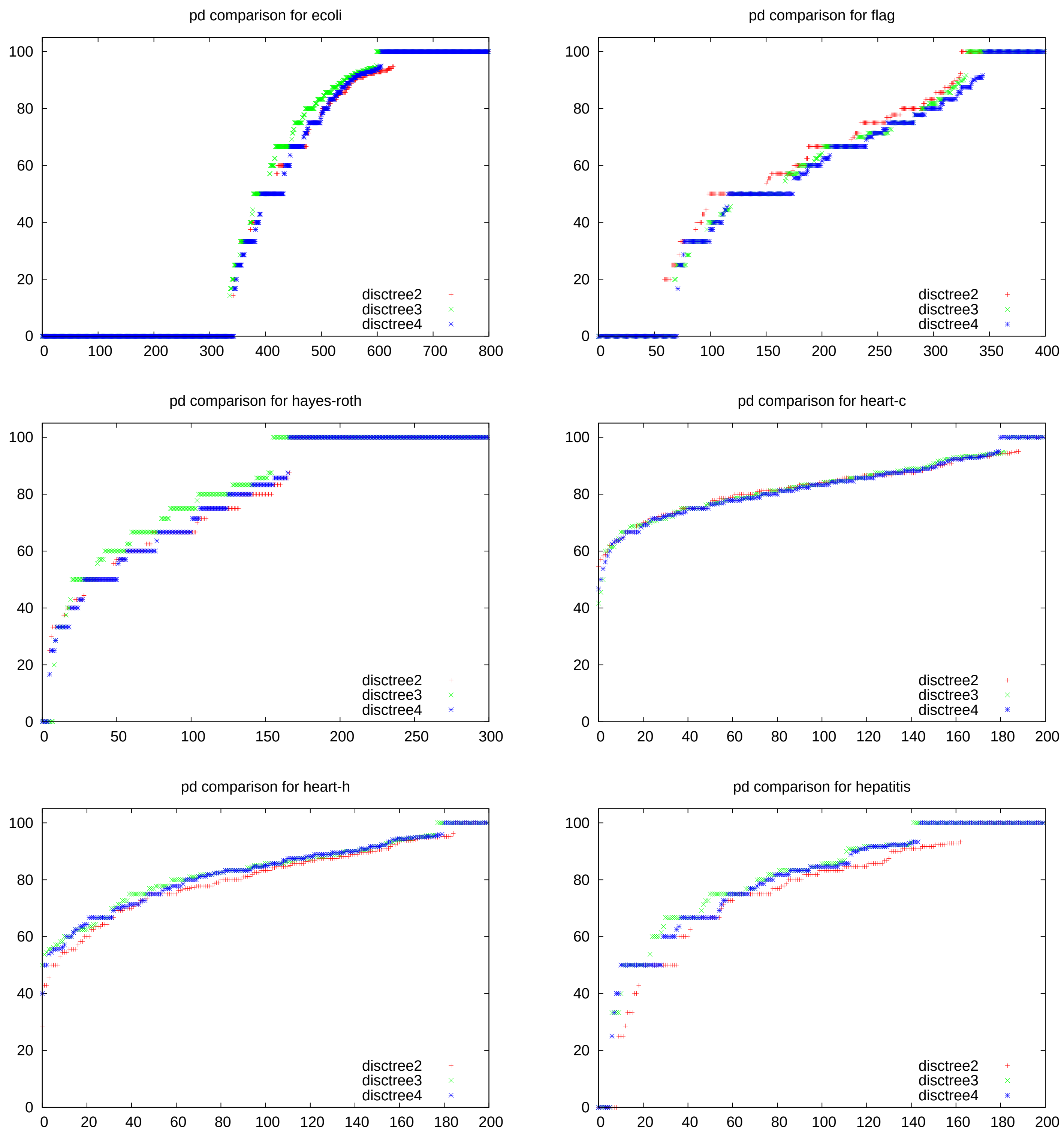


Figure 5.18: Plots of Probability of Detection Scores, Sorted by Value

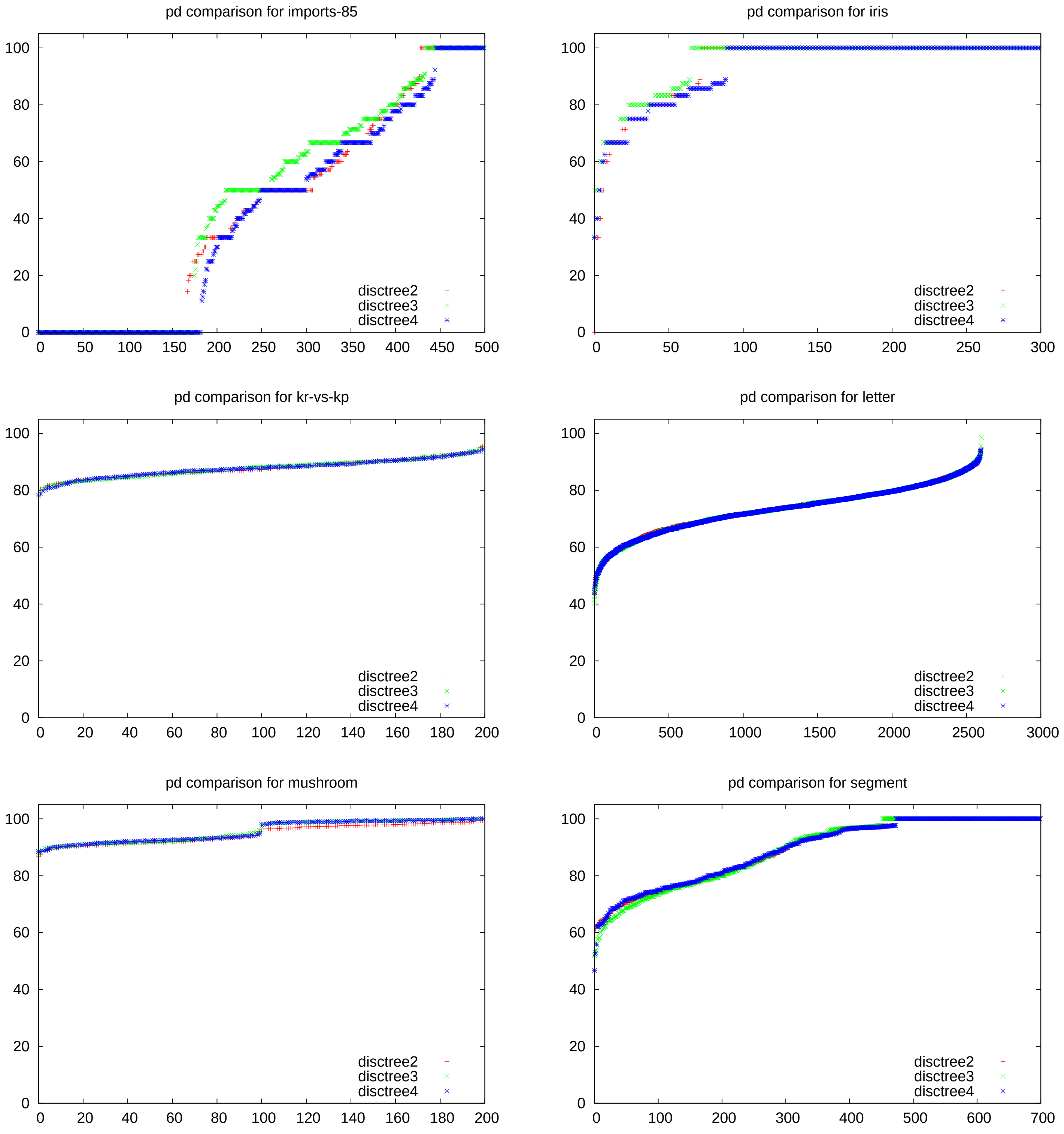


Figure 5.19: Plots of Probability of Detection Scores, Sorted by Value

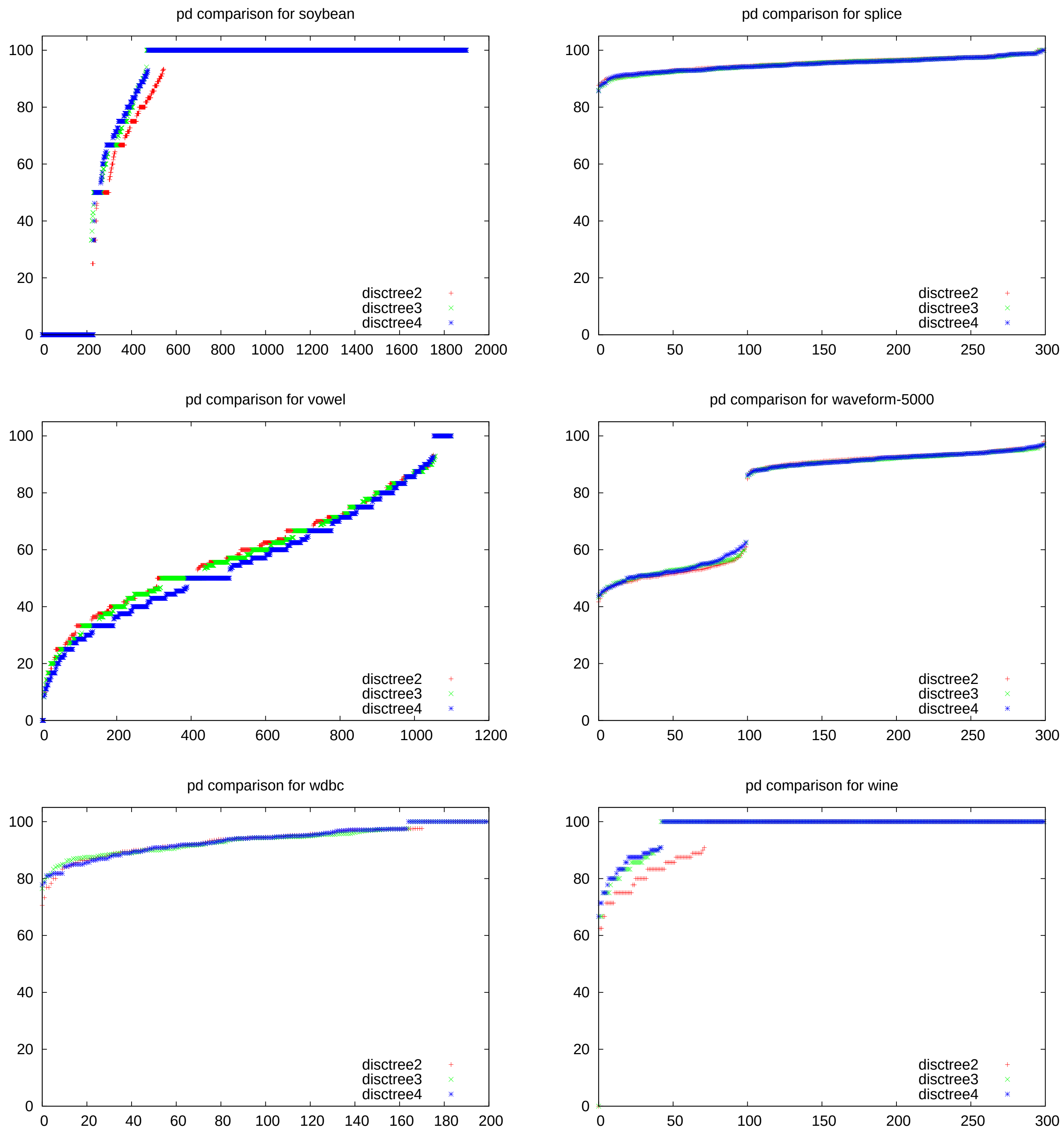


Figure 5.20: Plots of Probability of Detection Scores, Sorted by Value

5.1.5 Probability of Not False Alarm

The results of a U-test comparing the three method's probability of not false alarm across all the results from each data set are found in Figure 5.21. The results show *disctree3* and *disctree4* each with a win over *disctree2*, but in a tie with each other. The U-test thus tells us we should not choose *disctree2*, the method that includes both garbage collection and nominal attribute discretization, if we seek to maximize the probability of not false alarm results and minimizes the probability of false alarm scores for the method. It does not, however, tell us which of the other methods to chose.

key	*win*	*loss*	*ties*	*win-loss*
disctree4	1	0	1	1
disctree3	1	0	1	1
disctree2	0	2	0	-2

Figure 5.21: overall for npf

In addition to the U-test results across all the data sets, we have isolated the results for each data set, sorted the resulting accuracy scores, and provided them in the plots found in Figure 5.1.3 to Figure 5.1.3. The *x-axis* of each of these graphs is the number of data points generated as results for that data set, with each point having one plot and representing one probability of not false alarm score. The *y-axis* of each of the graphs is the probability of not false alarm percentage achieved for the discretization method of plotted point.

While the methods perform similarly on many of the data sets, there are some stand out results that are worth noting:

- The script *disctree3*, green, clearly wins in six data sets (auto-mpg, hayes-roth, heart-h, hepatitis, imports-85, and iris) while only clearly losing in one data set (flag).
- The script *disctree2*, red, clearly wins in two data sets (diabetes and flag), while it loses in six (audiology, heart-h, hepatitis, mushroom, soybean, and wine).

- The script *disctree4*, blue, has no clear wins but has two losses (diabetes and iris) where it achieves lower not probability of failure scores than the other methods for many of the same data points.

While the U-test finds no clear winner between the *disctree3* and *disctree4* methods, a review of the data plots leads us to assert that of the two methods, *disctree3* would be a better choice because it wins more often in the data plots.

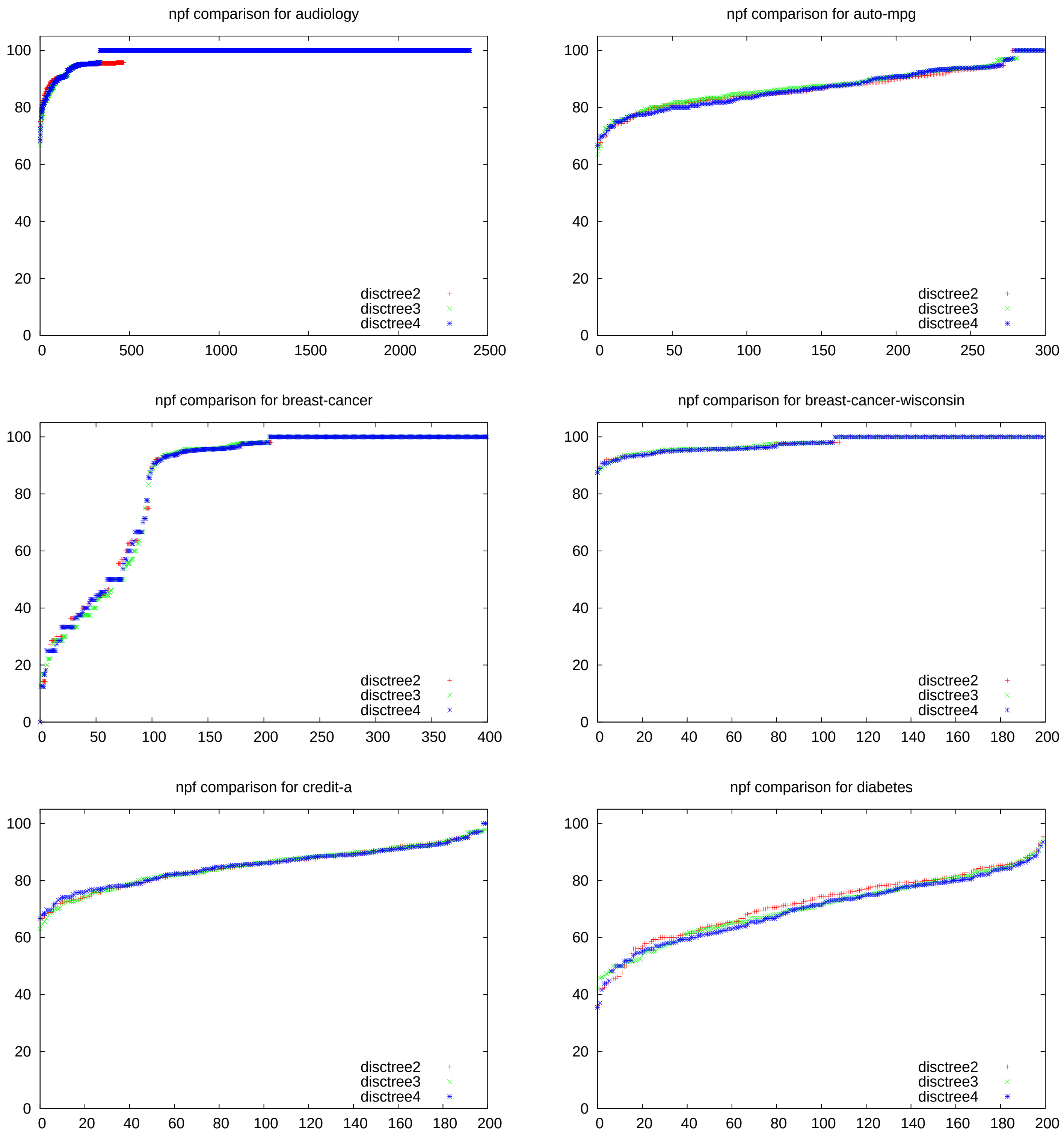
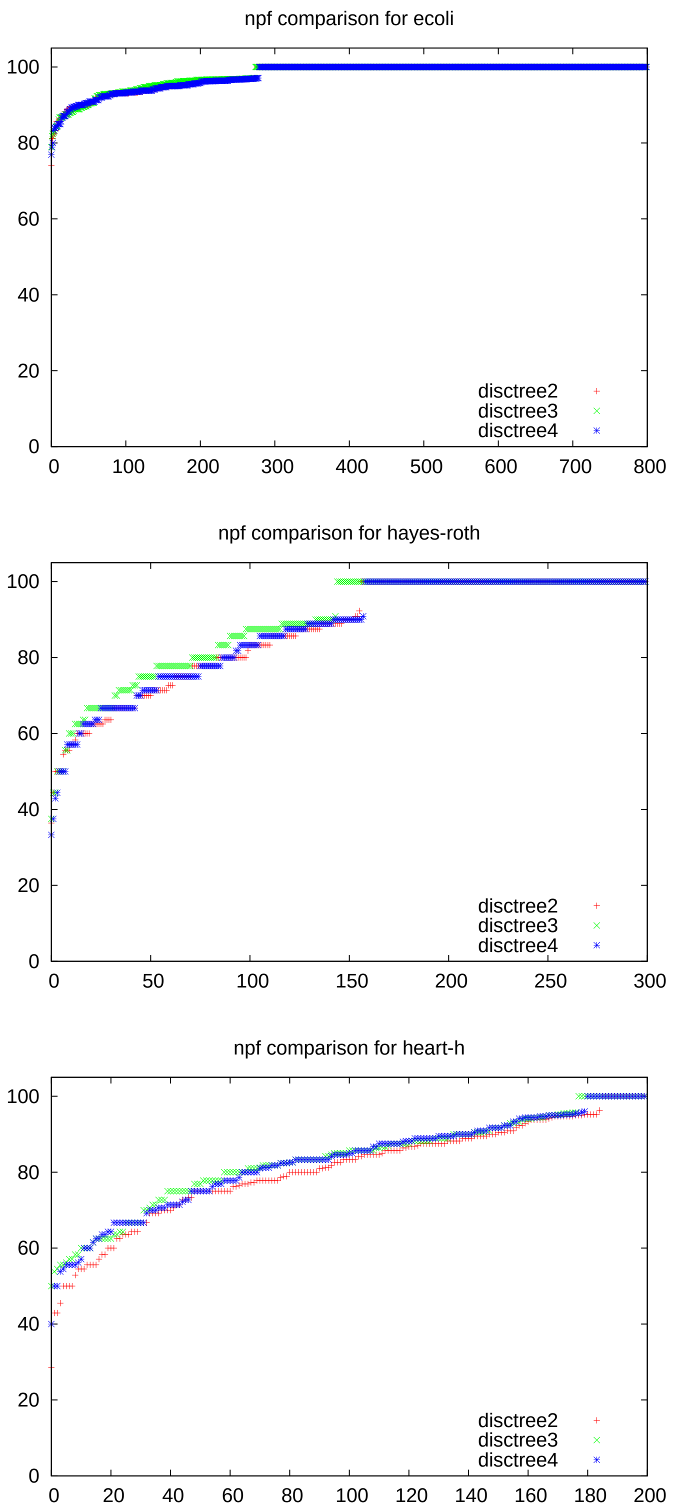


Figure 5.22: Plots of Probability of not False Alarm Scores, Sorted by Value



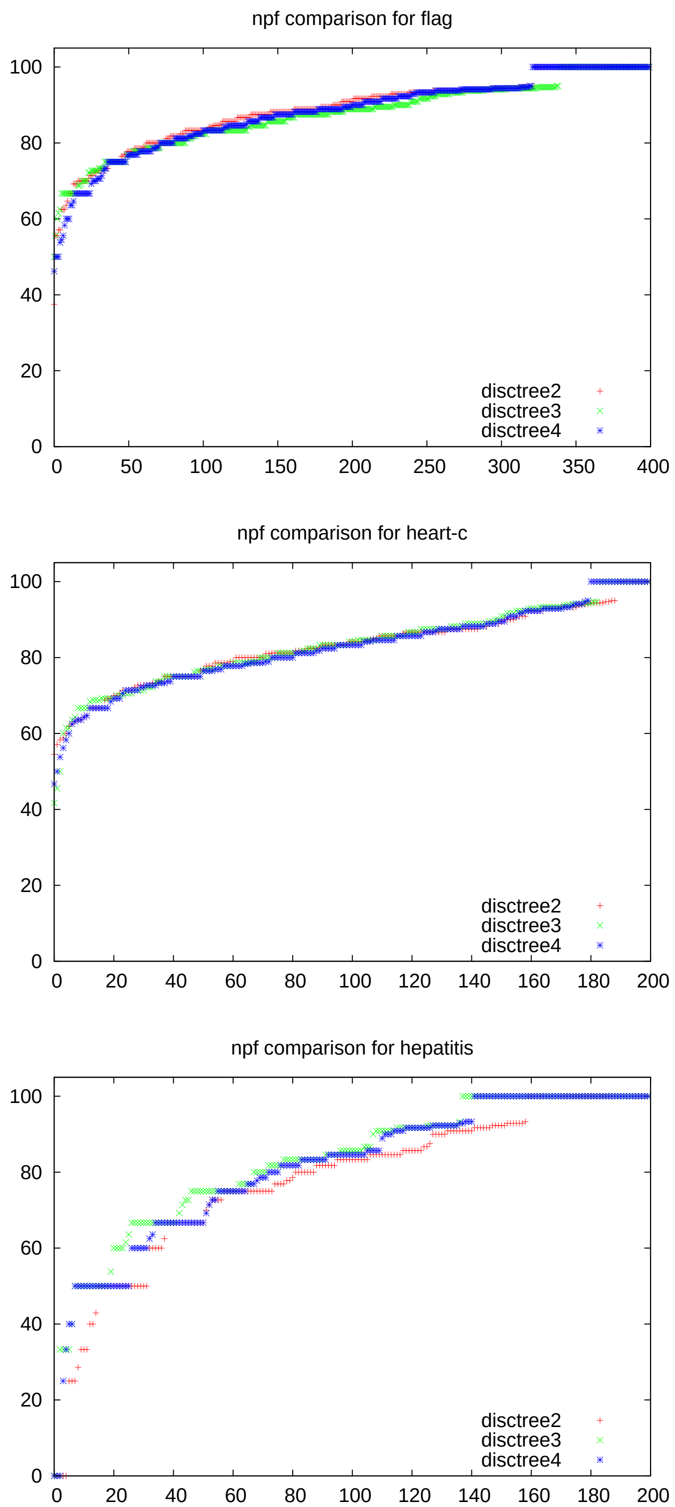


Figure 5.23: Plots of Probability of not False Alarm Scores, Sorted by Value

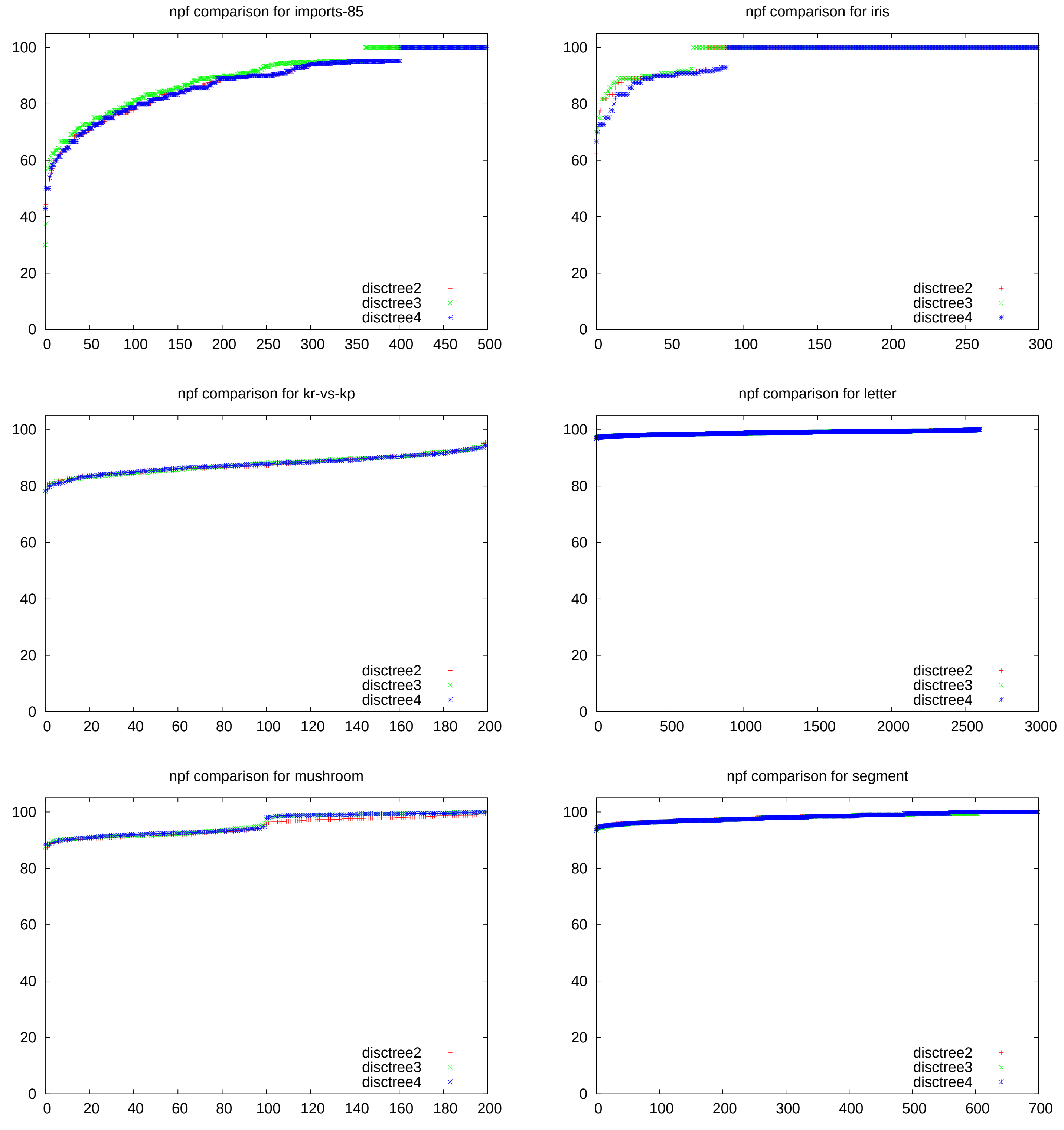


Figure 5.24: Plots of Probability of not False Alarm Scores, Sorted by Value

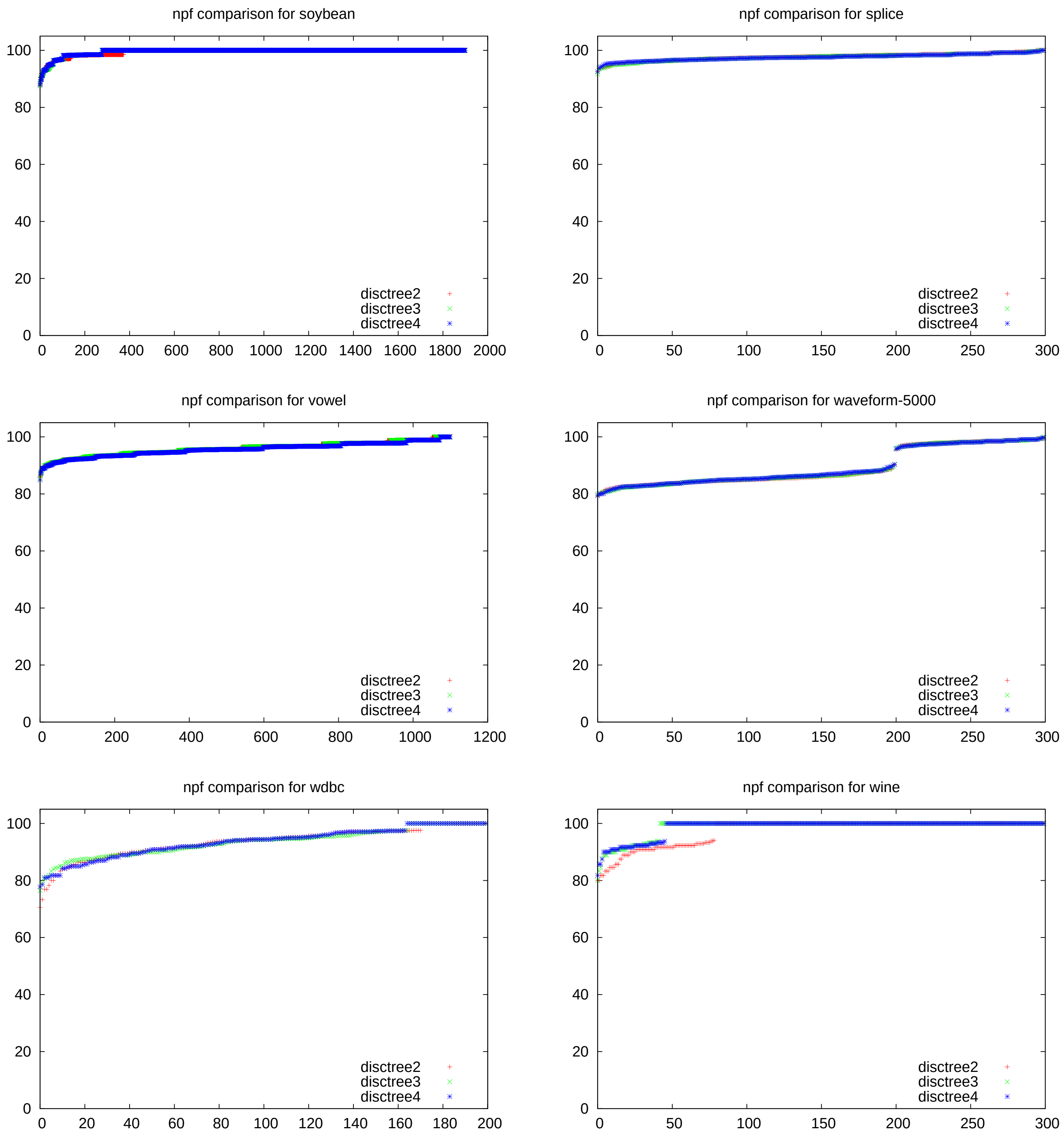


Figure 5.25: Plots of Probability of not False Alarm Scores, Sorted by Value

5.1.6 Decision Tree Method Selection

The results above show us that, overall, the *disctree3* method clearly obtains the most wins. In regard to accuracy, it is the only method to achieve a win in the U-test, and wins in several of the data plots. In regard to balance, precision, and probability of detection, it performs as well as the other methods in regard to the U-test but clearly wins in more of the data plots. Not Probability of Failure gives us two options in regards to methods that achieve U-test wins, but clearly *disctree3* wins in more data plot comparisons. Another interesting note from that measure is that clearly the method with nominal attribute discretization, *disctree2* would lead to higher a probability of false alarm, which, when paired with its result of one loss and one tie in the U-test for accuracy, would lead to a decision that it is certainly not the best method for us to use.

Based on the results above, we selected the implementation that used the Garbage Collection but did not perform nominal attribute discretization. This method, *disctree3*, will be used as the DiscTree algorithm in the comparisons against other discretization methods.

5.2 Discretization Method Comparison

5.2.1 Accuracy Results

The overall results for the accuracy comparison between the discretization methods discussed in Chapter 3, across all the data generated, can be found in Figure 5.26

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	1	1	2	0
pkid	1	1	2	0
disctree3	1	1	2	0
cat	0	4	0	-4

Figure 5.26: overall for acc

The results of this overview shows that in comparison with all the other discretization methods, the entropy-minimization method, called fayyadIrani in this comparison, won over each of the other methods. The tenbins (tbin), PKID (pkid), and DiscTree (disctree3) methods each performed comparably well, each winning a comparison, losing a comparison, and reaching a tie twice. The pure data set, provided by the "cat" method, had the worst results, losing out to each other method.

These results provide a great general summary of the results. However, each data set has also had the win/loss/tie record created for that data set. Those results can be found in Figure F.1 to Figure F.22. Figure 5.27 displays the winning method(s) for each data set where all the discretization methods did not tie. The winner or winners for each dataset are displayed, along with the number of degrees by which it wins. The counts for total wins are found in Figure 5.28. Clearly, the entropy-minimization heuristic, fayyadIrani, had more accuracy wins than any other method with ten, while DiscTree and tenbins each receiving one third of that number of wins. Interestingly, there is no clear manner by which to describe the conditions under which the DiscTree algorithm wins, other than that the data sets where it is recorded as a winner tend to have fewer classes (less than 4). However, it also loses to the entropy-minimization heuristic in similar data sets.

Another interesting result is the sorted data plots for each data set. These plots, found in Figure 5.2.1 to Figure 5.2.1, show that in most of the data sets where the fayyadIrani method wins, there is so very little difference in the accuracy scores that the plotted points for fayyadIrani, generate on top of or very close to the scores for the other methods. This should mean that, even when the entropy-minimization method wins, it is doing so by very little, unlike the DiscTree win in the auto-mpg data set, where the DiscTree accuracy plot quite clearly shows it winning across most of the data plot.

Data Set	*Instances*	*Winner(s)*	*Degree*	*Numeric*	*Classes*
hayes-roth	132	cat/fayyadIrani	1(1)	1	3
hepatitis	155	fayyadIrani	2(3)	6	2
flag	194	tbin/pkid/fayyadIrani/disctree3	1	10	4
imports-85	205	fayyadIrani	3(4)	15	5
heart-c	303	tbin	1(1)	6	5
auto-mpg	398	disctree3	3	5	3
wdbc	569	fayyadIrani	3(3)	30	2
credit-a	690	disctree3	1	0	2
diabetes	768	fayyadIrani	2(4)	8	2
vowel	990	tbin/fayyadIrani	1(3)	10	11
segment	2310	fayyadIrani/tbin	2(2)	19	7
waveform-5000	5000	fayyadIrani	2(4)	40	3
letter	20000	fayyadIrani	1(1)	16	26

Figure 5.27: These data sets had a particular winner(s) for their Accuracy comparison. In all cases, degree measures the number of wins over the next closest method. In the event that disctree3 did not win, the number in parenthesis represents its win difference from the lead method.

Method	*Wins*
fayyadIrani	10
disctree3	3
tbin	3
pkid	1
cat	1

Figure 5.28: Total Wins Per Method Based on Mann-Whitney U-Test Wins on each Data Set's Accuracy Scores

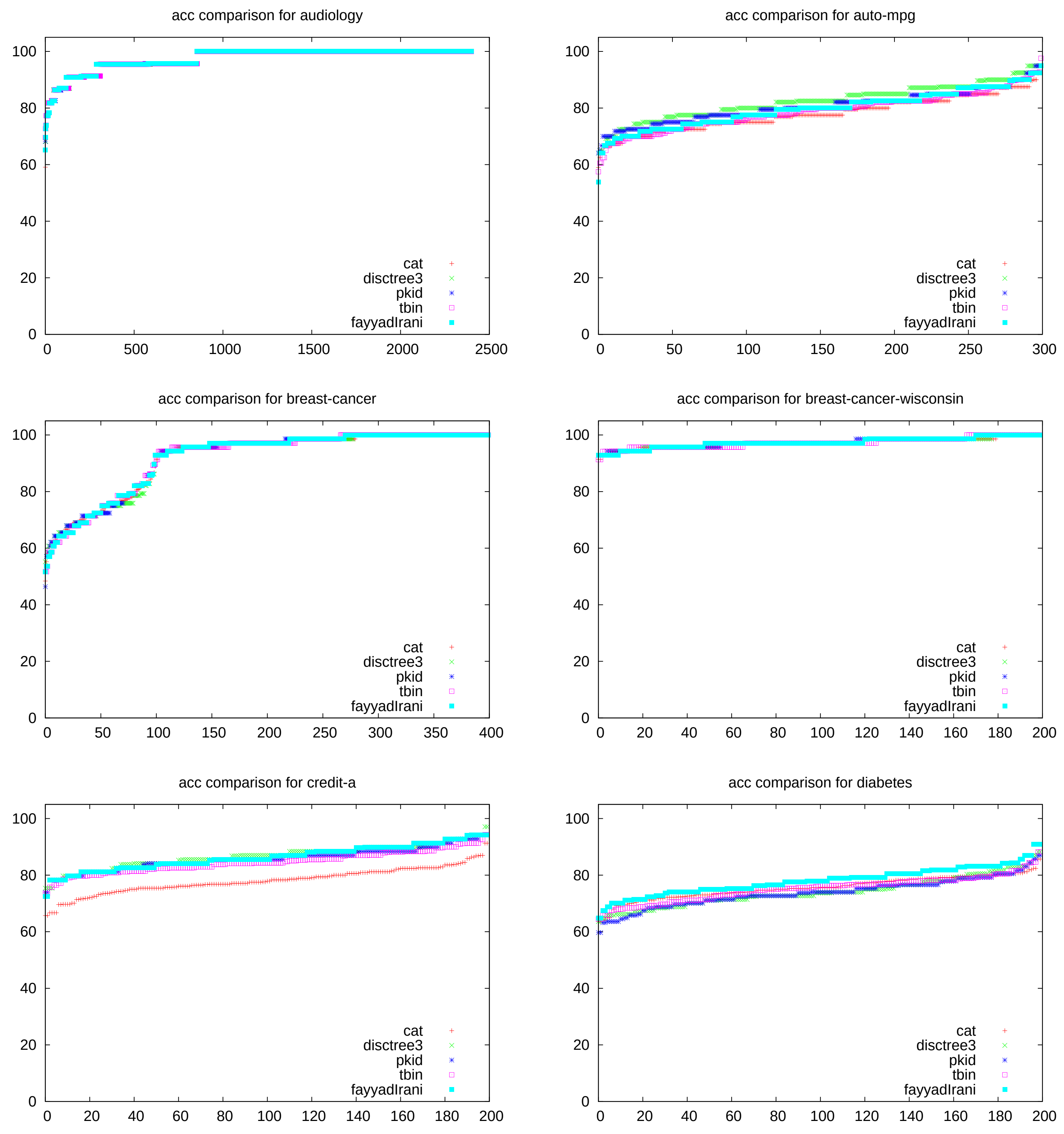


Figure 5.29: Plots of Accuracy Scores, Sorted by Value

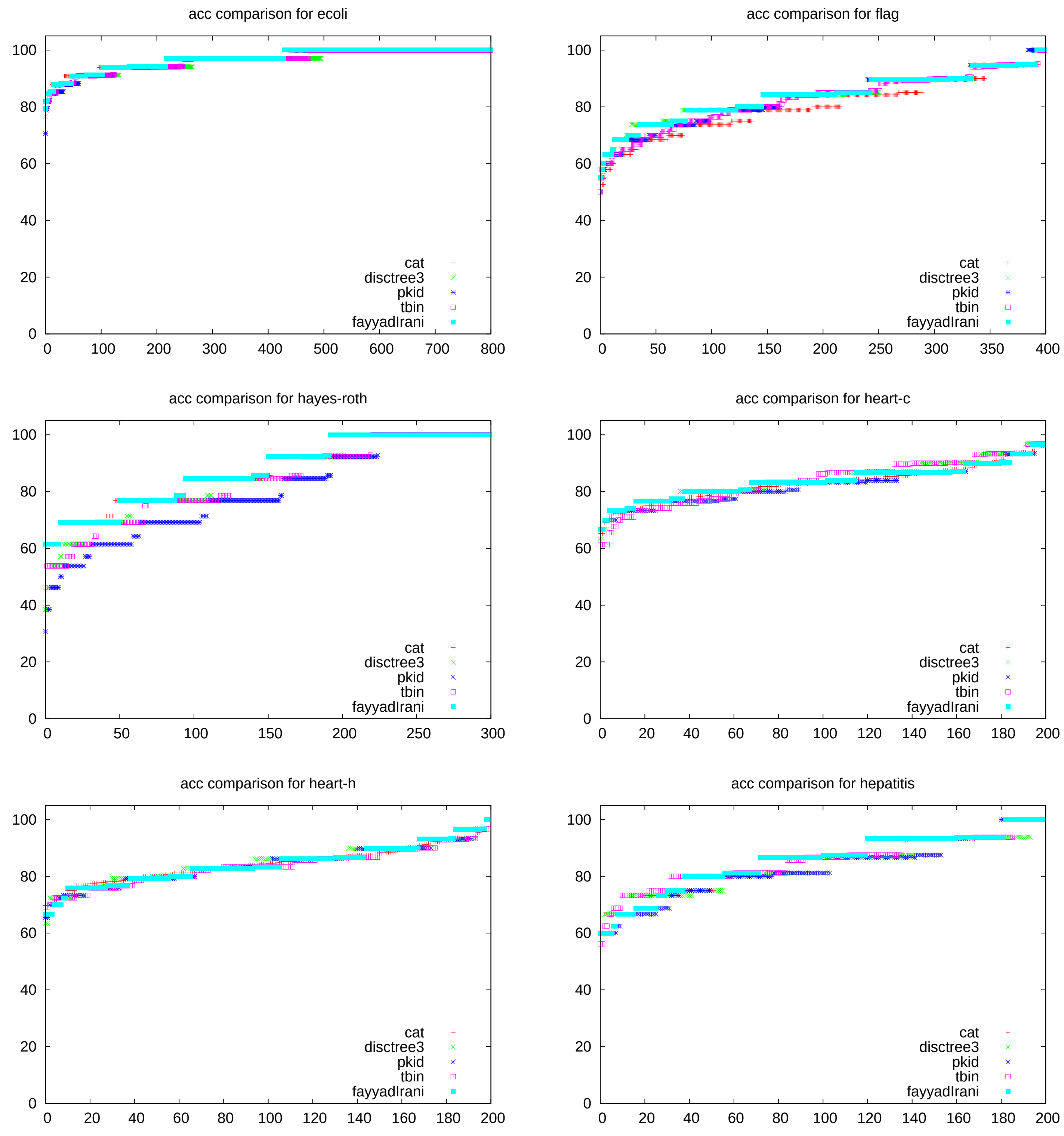


Figure 5.30: Plots of Accuracy Scores, Sorted by Value

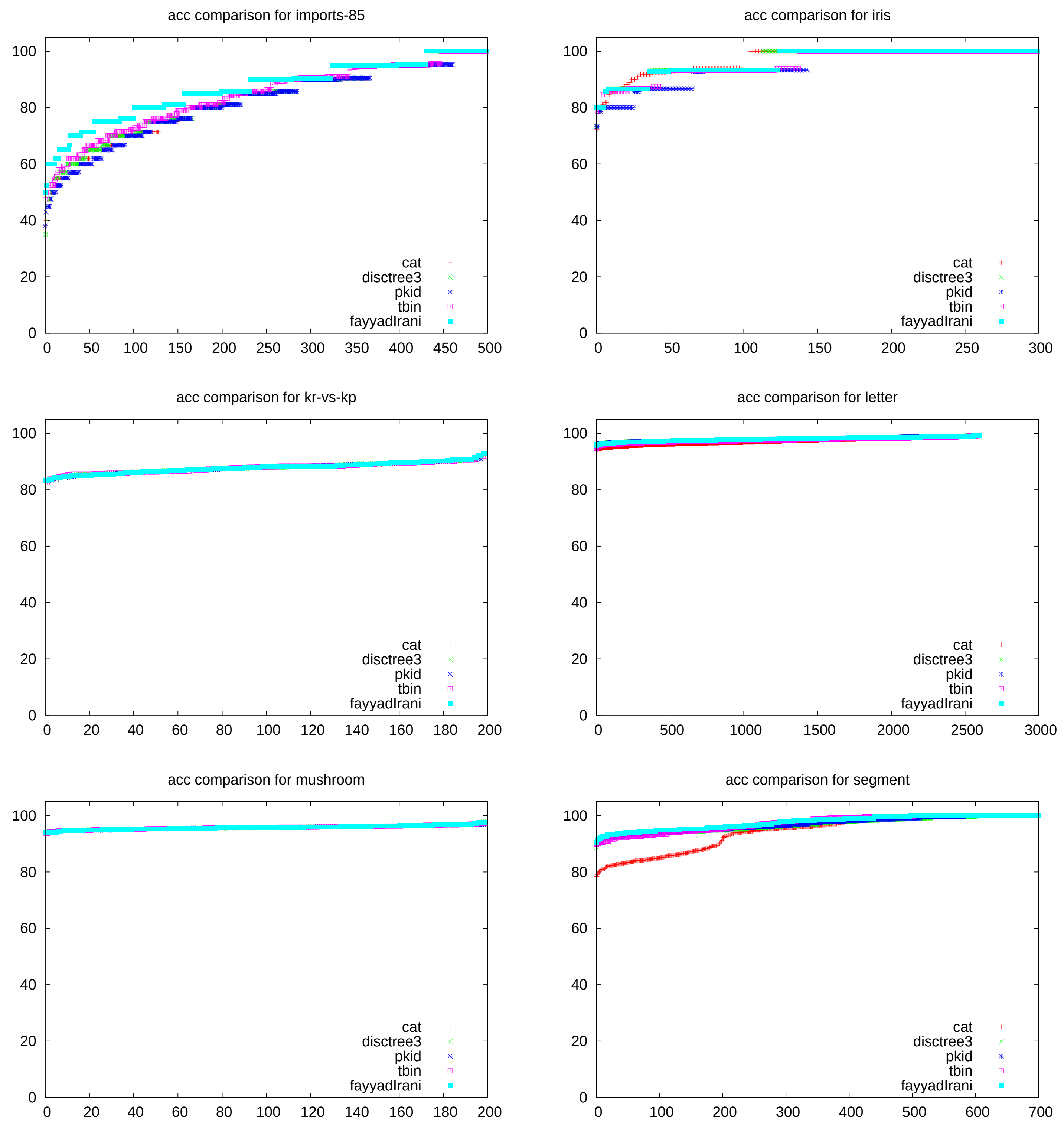


Figure 5.31: Plots of Accuracy Scores, Sorted by Value

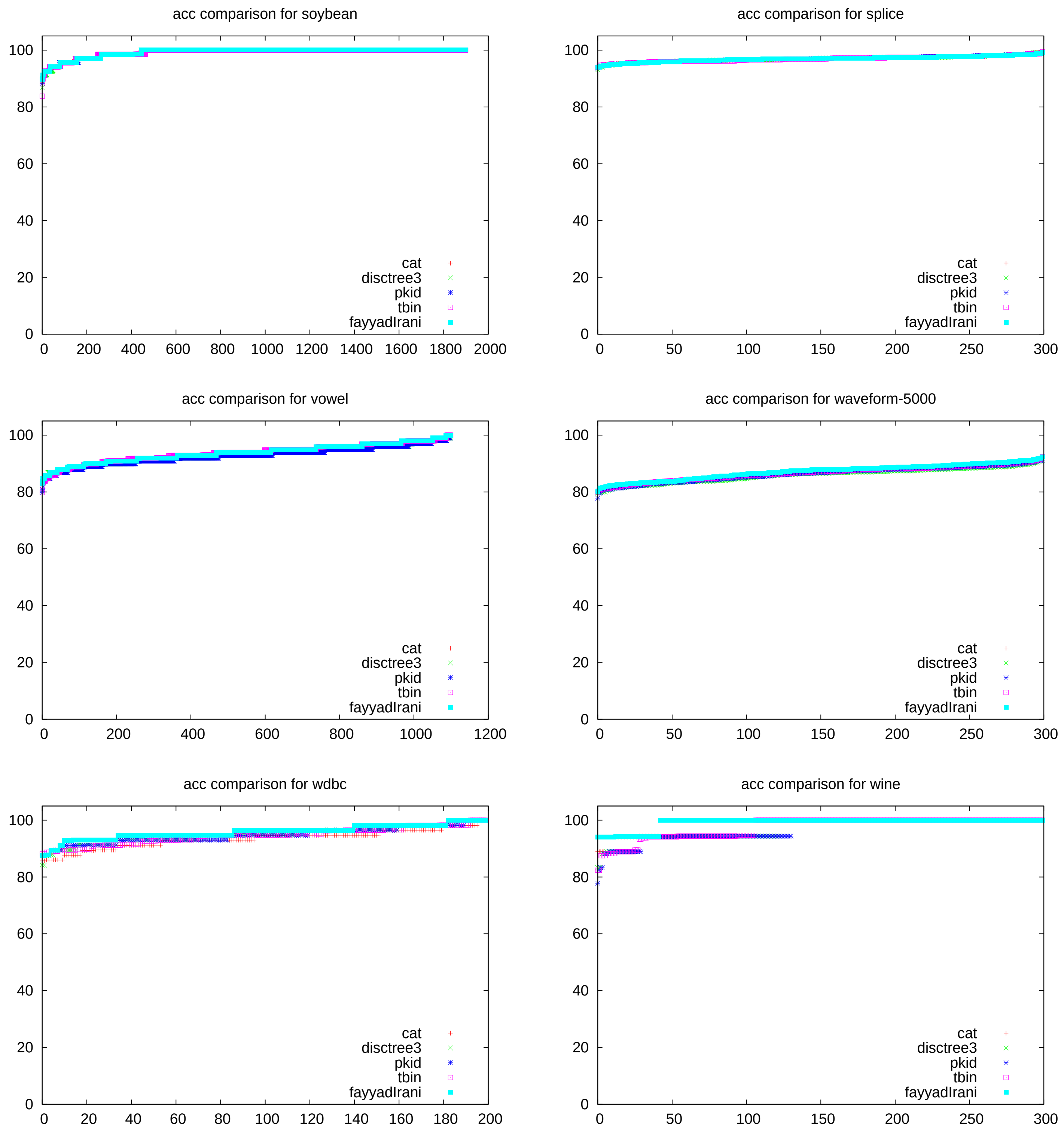


Figure 5.32: Plots of Accuracy Scores, Sorted by Value

5.2.2 Balance Results

The overall results for the probability of not false alarm comparison between the discretization methods discussed in Chapter 3, across all the data generated, can be found in Figure 5.33

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
pkid	2	1	1	1
disctree3	2	1	1	1
tbin	1	3	0	-2
cat	0	4	0	-4

Figure 5.33: overall for bal

The results of this overview shows that in comparison with all the other discretization methods, the entropy-minimization method, won over each of the other methods. The PKID (pkid), and DiscTree (disctree3) methods each performed comparably well, each winning two comparisons, losing a comparison, and reaching a tie once. The pure data set, provided by the ”cat” method, had the worst results, losing out to each other method, while tenbins performed slightly better winning once against other methods but losing to the other three.

These results provide a general summary of the results. However, each data set has also had the win/loss/tie record created for that data set. Those results can be found in Figure F.25 to Figure F.46. Figure 5.34 displays the winning method(s) for each data set where all the discretization methods did not tie. The winner or winners for each dataset are displayed, along with the number of degrees by which it wins. The counts for total wins are found in Figure 5.35. Clearly, the entropy-minimization heuristic, fayyadIrani, had more balance wins than any other method. DiscTree won five times, more than each of the other methods. Interestingly, there is no clear manner by which to describe the conditions under which the DiscTree algorithm wins, other than that the data sets where it is recorded as a winner tend to have fewer classes (less than 5). However, it also loses to the entropy-minimization heuristic in similar data sets.

Another interesting result is the sorted data plots for each data set. These plots, found in

Figure 5.2.2 to Figure 5.2.2, show that in most of the data sets where the fayyadIrani method wins there is so very little difference in the balance scores that the plotted points for fayyadIrani generate on top of or very close to the scores for the other methods. This should mean that, even when the entropy-minimization method wins, in most cases it is doing so by very little, unlike the DiscTree win in the auto-mpg data set or the fayyadIrani win in the imports-85 data set, where the respective method's plot quite clearly shows it winning across most of the data plot.

Data Set	*Instances*	*Winner(s)*	*Degree*	*Numeric*	*Classes*
hayes-roth	132	cat/disctree3/fayyadIrani	1	1	3
hepatitis	155	fayyadIrani/tbin	1(1)	6	2
flag	194	fayyadIrani	1(1)	10	4
imports-85	205	fayyadIrani/pkid	1(1)	15	5
heart-c	303	disctree3/tbin	1	6	5
ecoli	336	cat/fayyadIrani/pkid	2(2)	7	8
auto-mpg	398	disctree3	3	5	3
wdbc	569	fayyadIrani	3(3)	30	2
credit-a	690	disctree3	1	0	2
diabetes	768	fayyadIrani	3(3)	8	2
vowel	990	fayyadIrani/tbin	1(3)	10	11
segment	2310	fayyadIrani/tbin	2(2)	19	7
waveform-5000	5000	fayyadIrani	2(3)	40	3
letter	20000	disctree3/fayyadIrani/pkid	1	16	26

Figure 5.34: These data sets had a particular winner(s) for their Balance comparison. In all cases, degree measures the number of wins over the next closest method. In the event that disctree3 did not win, the number in parenthesis represents its win difference from the lead method.

Method	*Wins*
fayyadIrani	11
disctree3	5
tbin	4
pkid	3
cat	2

Figure 5.35: Total Wins Per Method Based on Mann-Whitney U-Test Wins on Each Data Set's Balance Scores

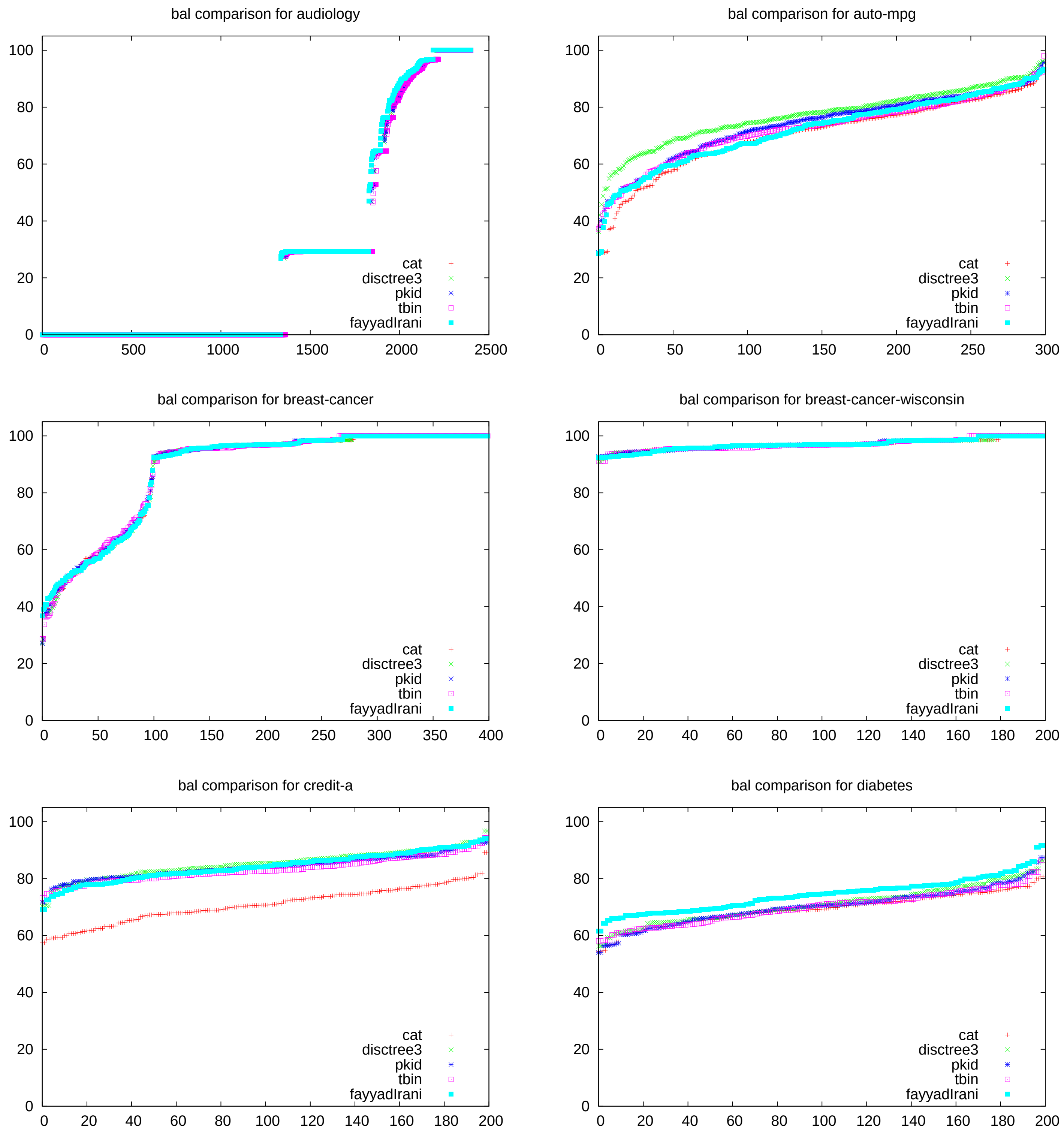


Figure 5.36: Plots of Balance Scores, Sorted by Value

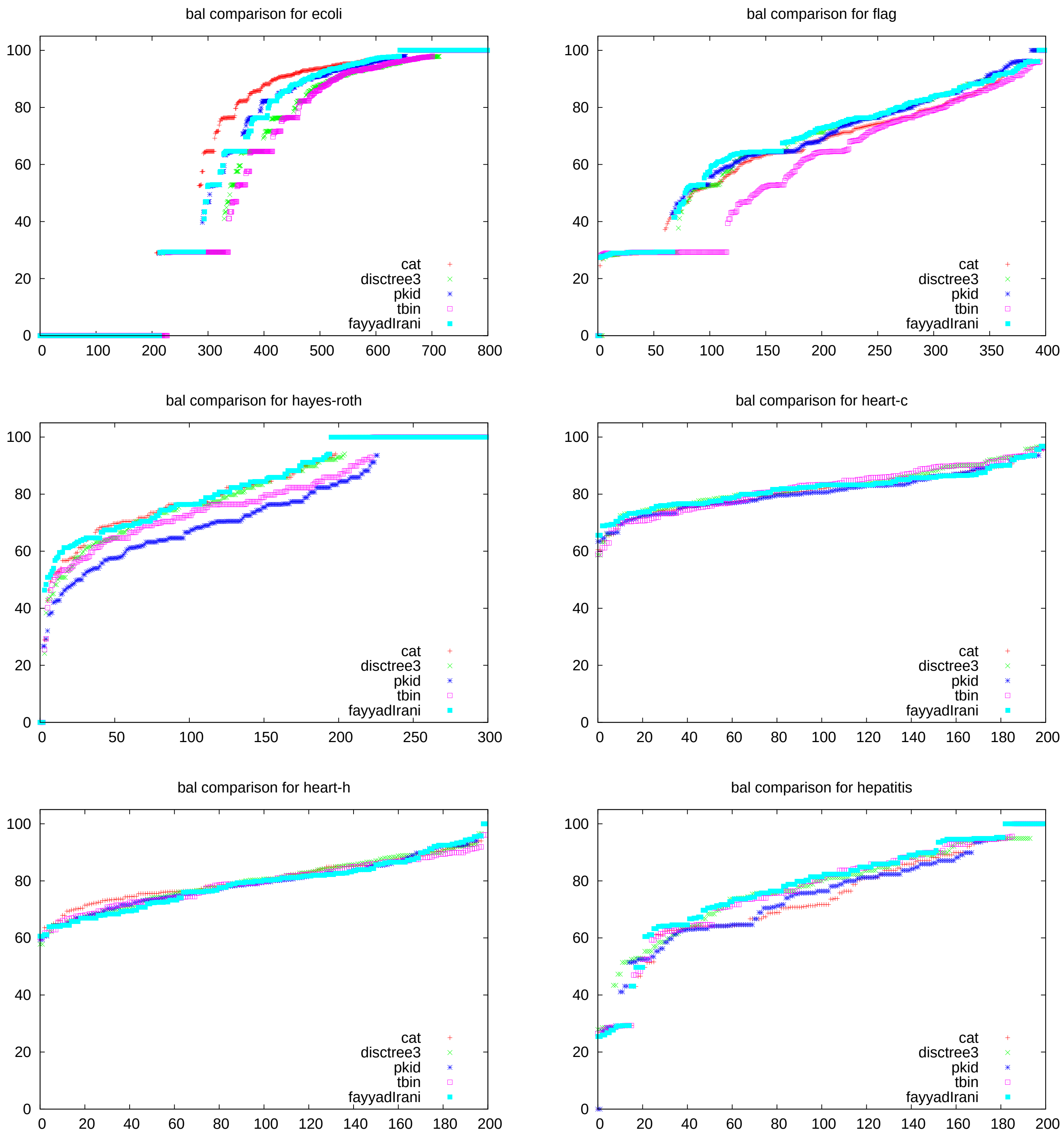


Figure 5.37: Plots of Balance Scores, Sorted by Value

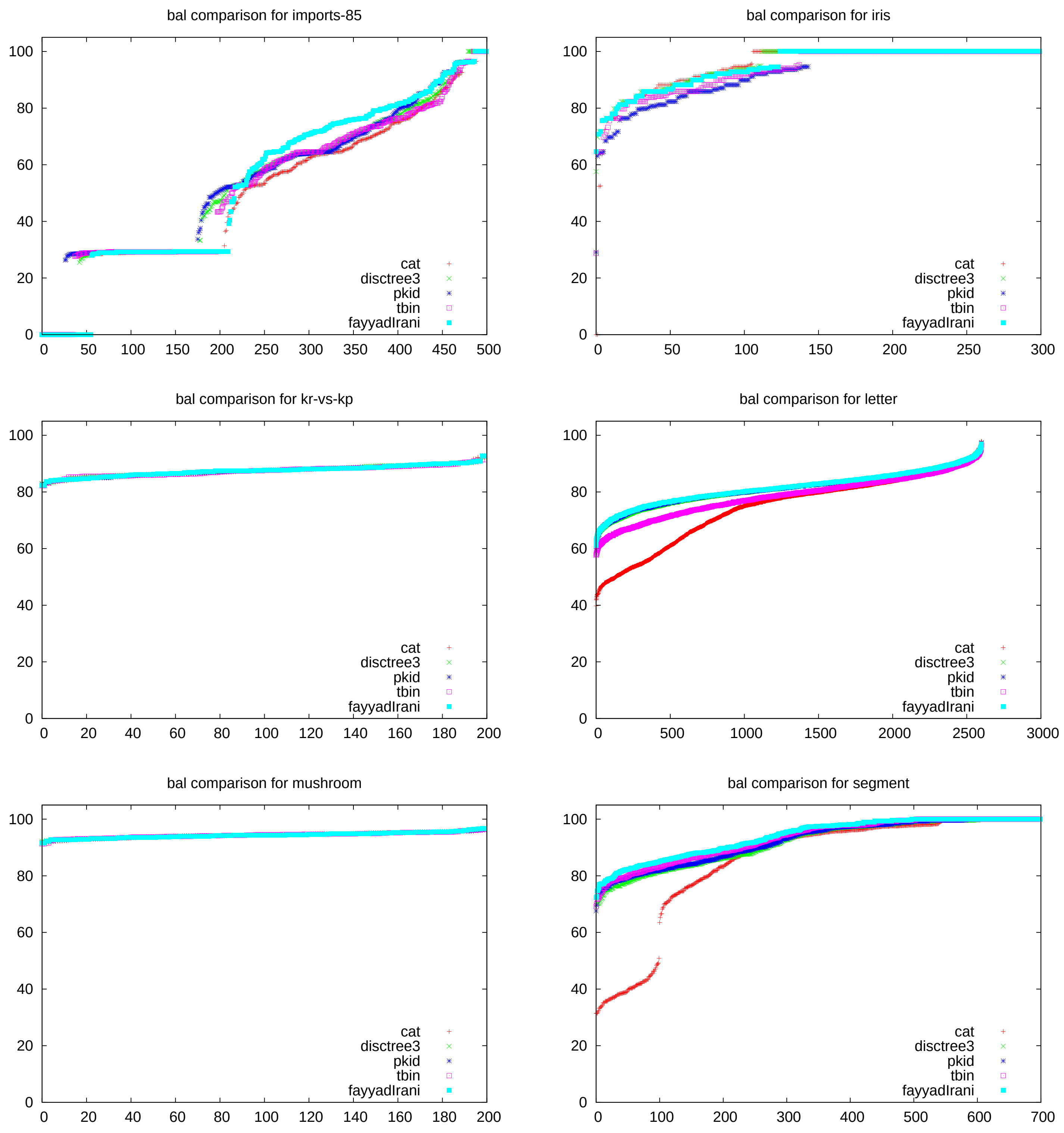


Figure 5.38: Plots of Balance Scores, Sorted by Value

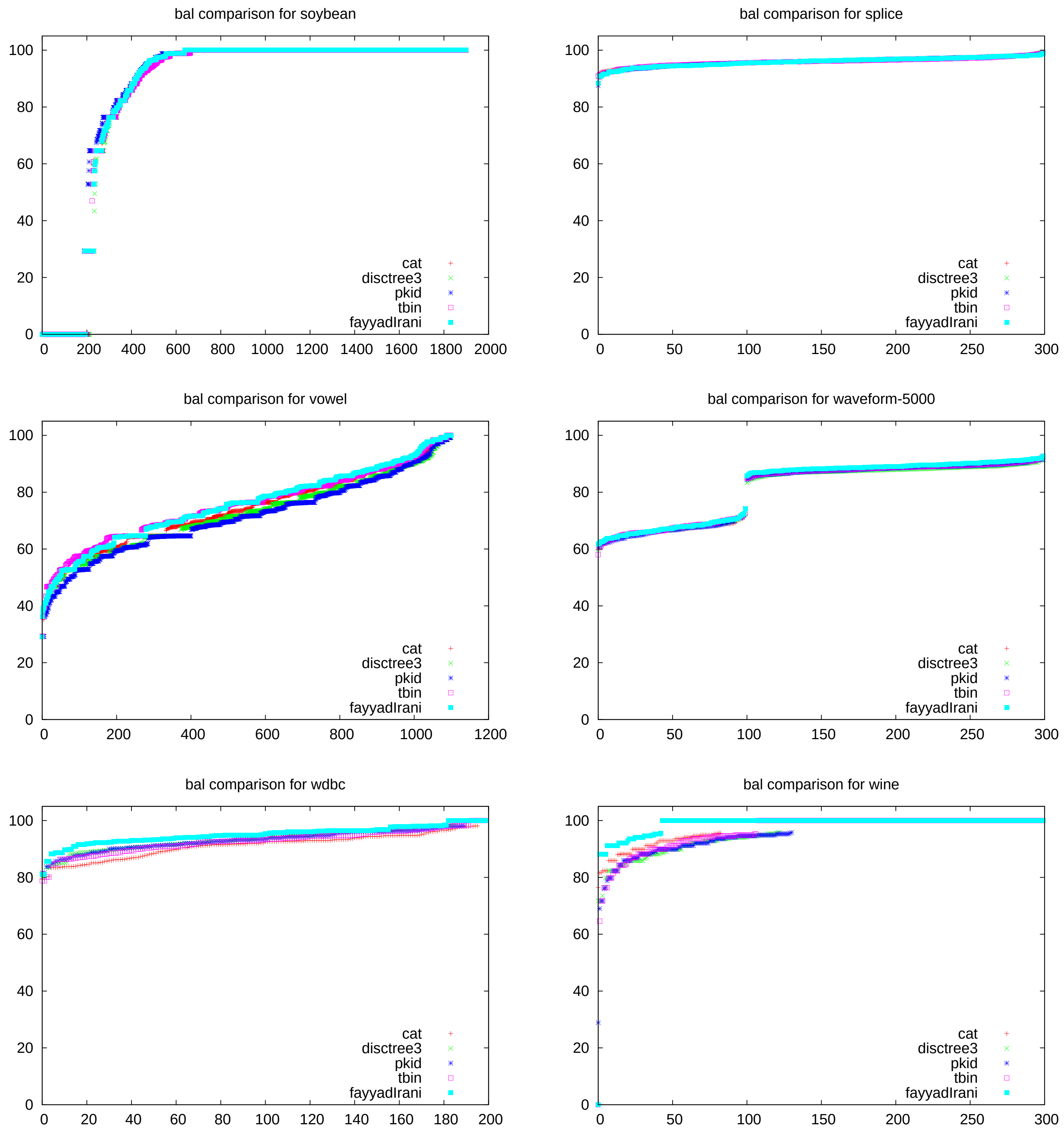


Figure 5.39: Plots of Balance Scores, Sorted by Value

5.2.3 Precision Results

The overall results for the precision comparison between the discretization methods discussed in Chapter 3, across all the data generated, can be found in Figure 5.40

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	1	1	2	0
pkid	1	1	2	0
disctree3	1	1	2	0
cat	0	4	0	-4

Figure 5.40: overall for prec

The results of this overview shows that in comparison with all the other discretization methods, the entropy-minimization method, fayyadIrani, won over each of the other methods. The tenbins (tbin), PKID (pkid), and DiscTree (disctree3) methods each performed comparably well, each winning a comparison, losing a comparison, and reaching a tie twice. The pure data set, provided by the ”cat” method, had the worst results, losing out to each other method.

These results provide a general summary of the results. However, each data set has also had the win/loss/tie record created for that data set. Those results can be found in Figure F.49 to Figure F.70. Figure 5.41 displays the method for each data set where all the discretization methods did not tie. The winner or winners for each dataset are displayed, along with the number of degrees by which it wins. The counts for total wins are found in Figure 5.42. The entropy-minimization heuristic, fayyadIrani, had more precision wins than any other method, with DiscTree winning half as many times. Interestingly, there is no clear manner by which to describe the conditions under which the DiscTree algorithm wins, other than that the data sets where it is recorded as a winner tend to have fewer classes (less than 5). However, it also loses to the entropy-minimization heuristic in similar data sets.

Another interesting result is the sorted data plots for each data set. These plots, found in Figure 5.2.1 to Figure 5.2.1, show that in most of the data sets where the fayyadIrani method wins there is so very little difference in the precision scores that the plotted points for fayyadIrani generate on top of or very close to the scores for the other methods. While in some cases (DiscTree with the auto-mpg data set, entropy in the ecoli data set), in most cases the plots are very, very similar and very little difference appears between the methods.

Data Set	*Instances*	*Winner(s)*	*Degree*	*Numeric*	*Classes*
hayes-roth	132	cat/disctree3/fayyadIrani/tbin	1	1	3
flag	194	cat/disctree3/fayyadIrani/pkid	1	10	4
heart-c	303	disctree3	1	6	5
ecoli	336	fayyadIrani	1(2)	7	8
auto-mpg	398	disctree3	3	5	3
wdbc	569	fayyadIrani	3(3)	30	2
credit-a	690	disctree3/fayyadIrani/pkid	1	0	2
diabetes	768	fayyadIrani	4(4)	8	2
vowel	990	cat/fayyadIrani/tbin	2(2)	10	11
segment	2310	fayyadIrani/tbin	2(3)	19	7
waveform-5000	5000	fayyadIrani	1(2)	40	3
letter	20000	fayyadIrani	1(1)	16	26

Figure 5.41: These data sets had a particular winner(s) for their Precision comparison. In all cases, degree measures the number of wins over the next closest method. In the event that disctree3 did not win, the number in parenthesis represents its win difference from the lead method.

Method	*Wins*
fayyadIrani	10
disctree3	5
tbin	3
pkid	2
cat	3

Figure 5.42: Total Wins Per Method Based on Mann-Whitney U-Test Wins on Each Data Set's Precision Scores

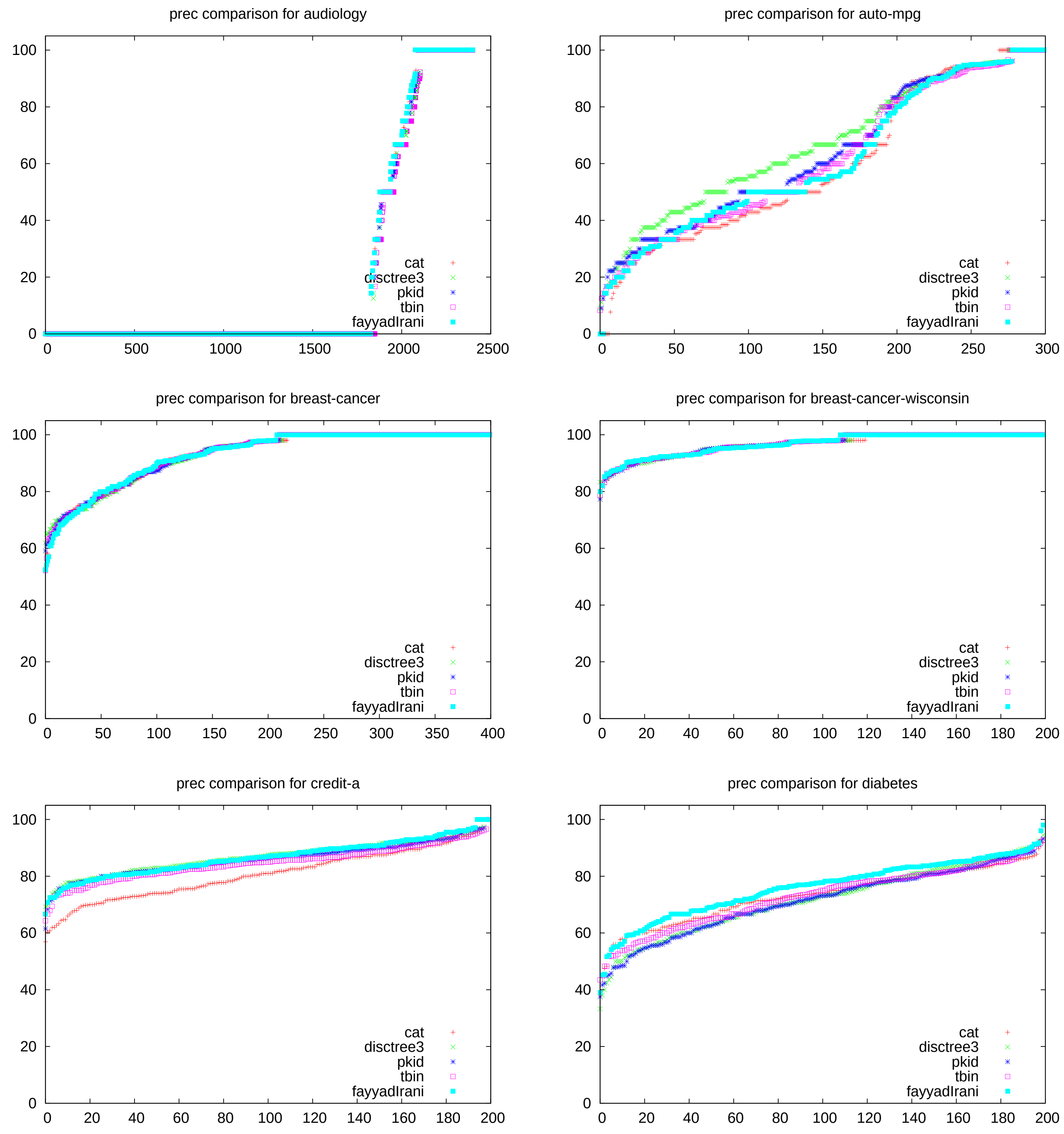


Figure 5.43: Plots of Precision Scores, Sorted by Value

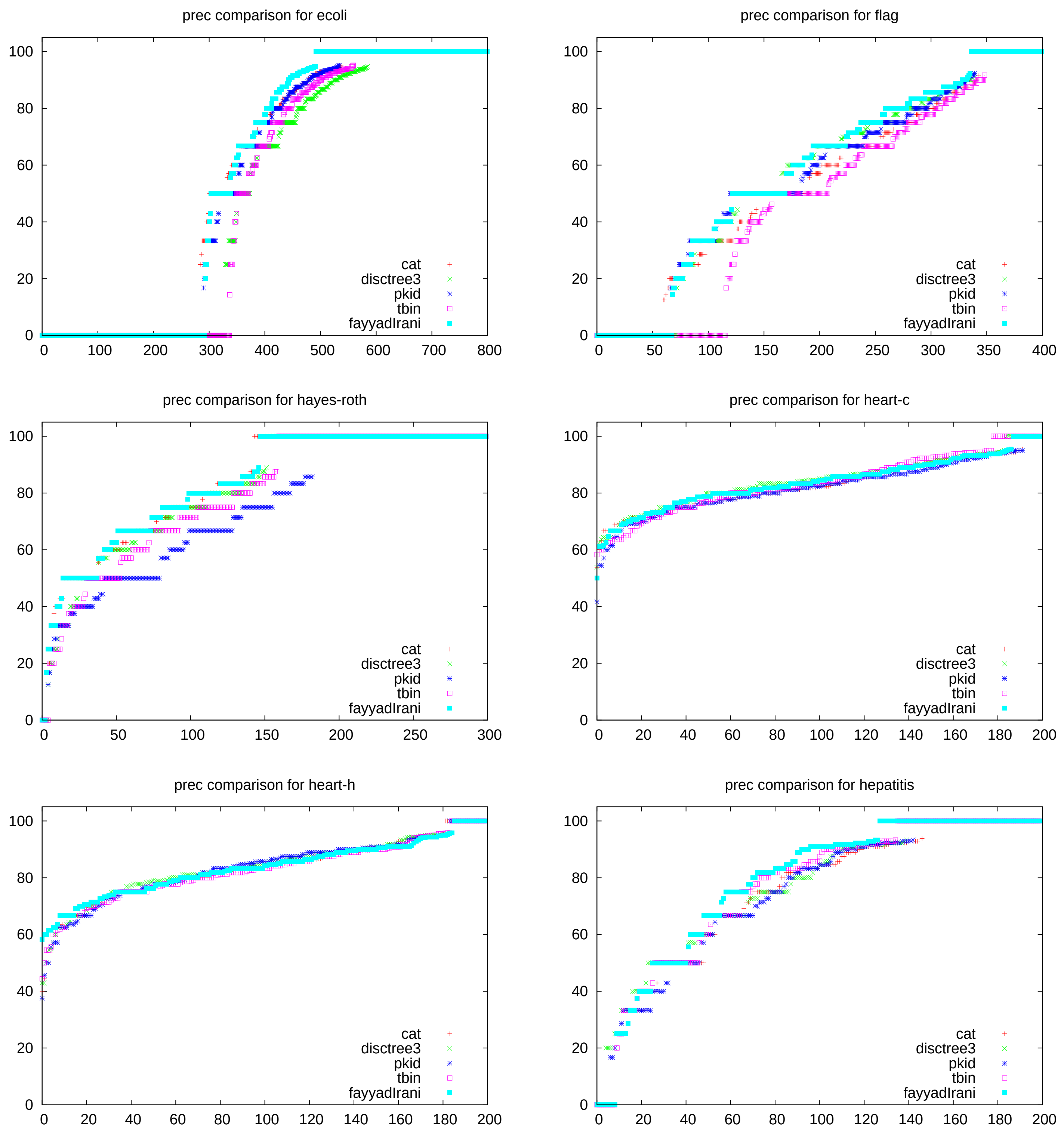


Figure 5.44: Plots of Precision Scores, Sorted by Value

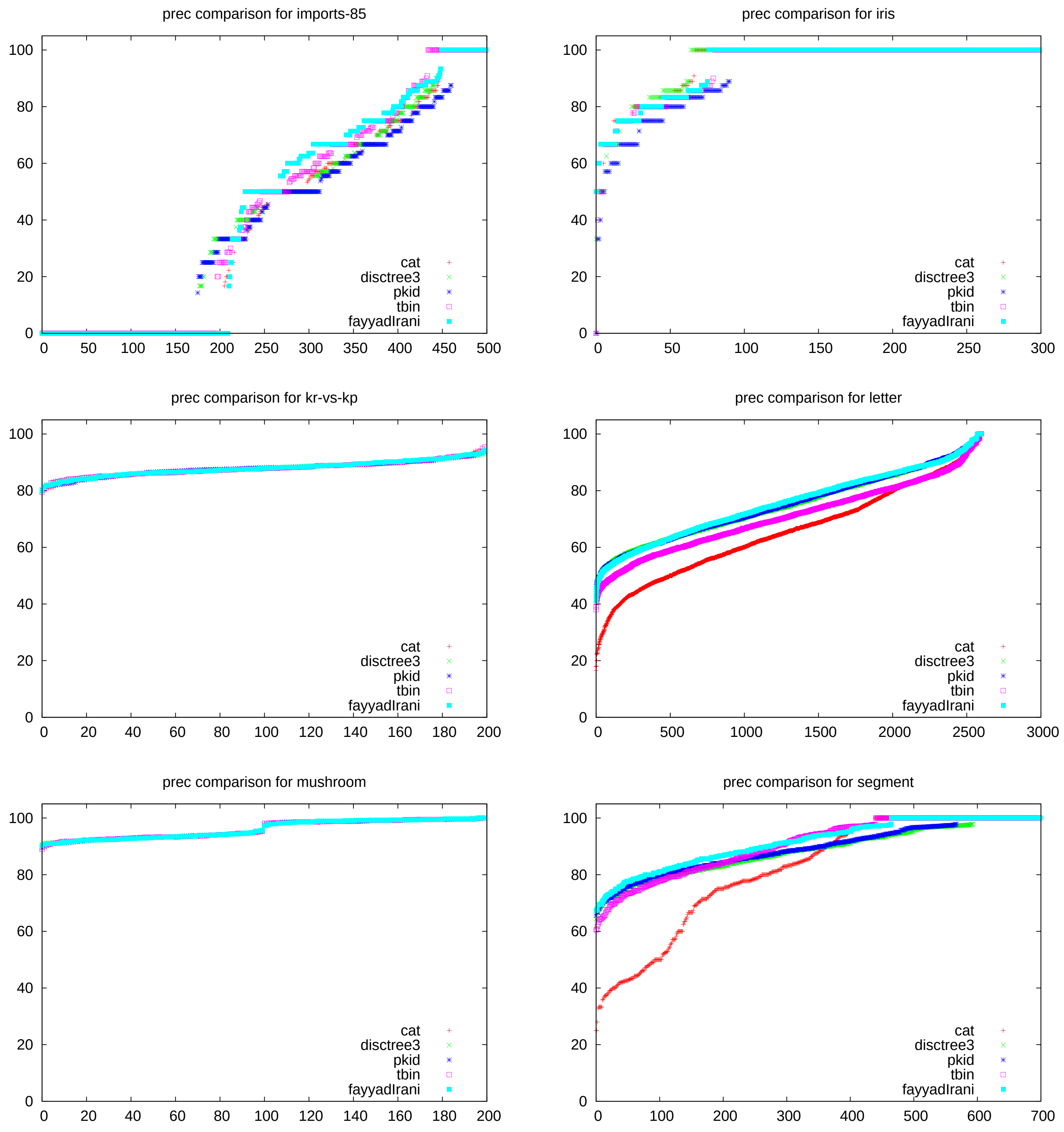


Figure 5.45: Plots of Precision Scores, Sorted by Value

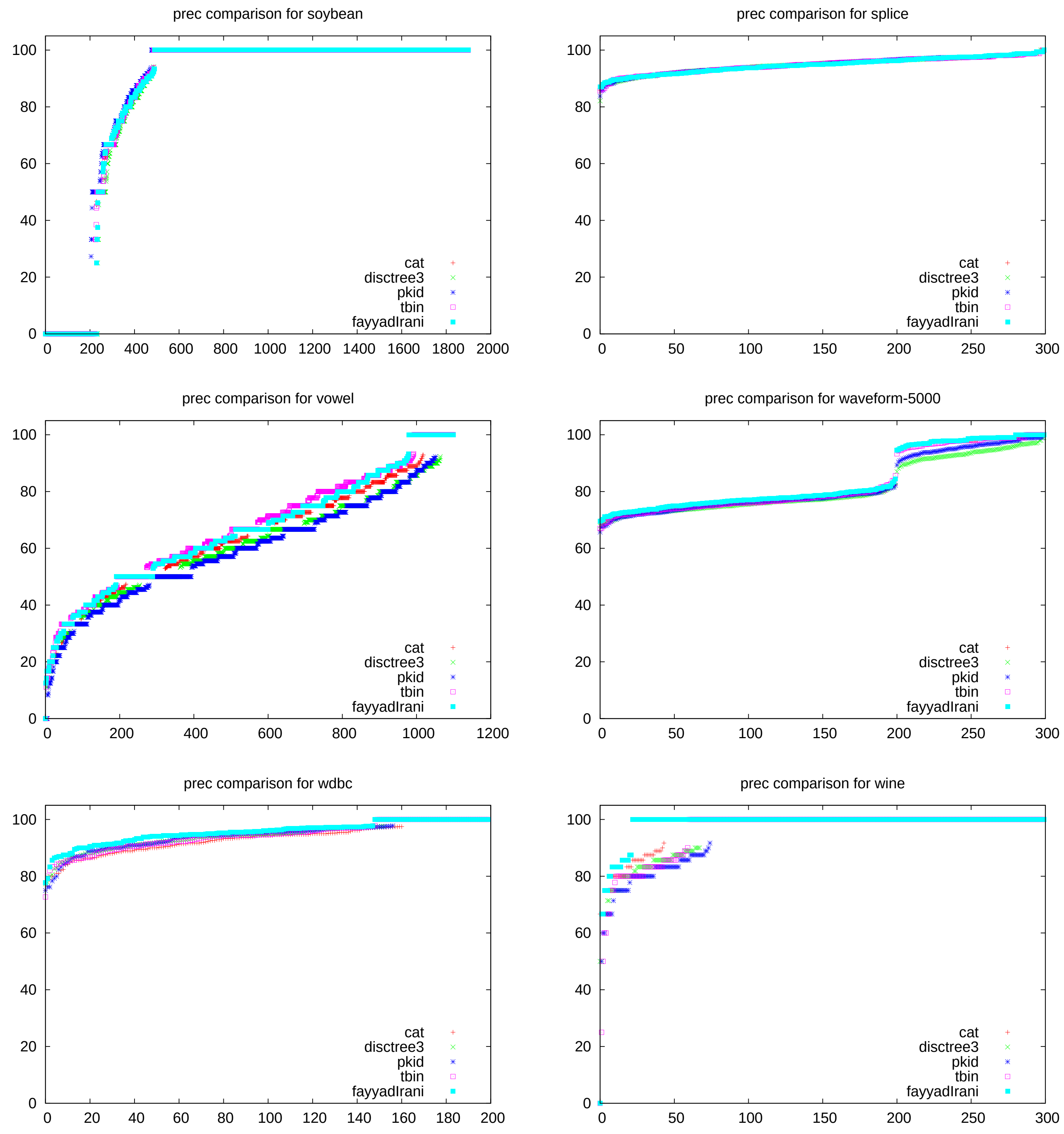


Figure 5.46: Plots of Precision Scores, Sorted by Value

5.2.4 Probability of Detection Results

The overall results for the probability of detection comparison between the discretization methods discussed in Chapter 3, across all the data generated, can be found in Figure 5.47

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
pkid	2	1	1	1
disctree3	2	1	1	1
tbin	0	3	1	-3
cat	0	3	1	-3

Figure 5.47: overall for pd

The results of this overview shows that in comparison with all the other discretization methods, the entropy-minimization method, called fayyadIrani in this comparison, won over each of the other methods. The PKID (pkid), and DiscTree (disctree3) methods each performed comparably well, each winning two comparisons, losing a comparison, and reaching a tie once. The pure data set, provided by the ”cat” method, had the worst results, losing out to each other method. The tenbins method (tbin) performed just as poorly.

These results provide a general summary of the results. However, each data set has also had the win/loss/tie record created for that data set. Those results can be found in Figure F.73 to Figure F.94. Figure 5.48 displays the winning method(s) for each data set where all the discretization methods did not tie. The winner or winners for each dataset are displayed, along with the number of degrees by which it wins. The counts for total wins are found in Figure 5.49. The entropy-minimization heuristic, fayyadIrani, had more probability of detection wins than any other method, with DiscTree having the next highest number of wins. Interestingly, there is no clear manner by which to describe the conditions under which the DiscTree algorithm wins, other than that the data sets where it is recorded as a winner tend to have fewer classes (typically less than 5). However, it also loses to the entropy-minimization heuristic in similar data sets.

Another interesting result is the sorted data plots for each data set. These plots, found in

Figure 5.2.4 to Figure 5.2.4, show that in most of the data sets where the fayyadIrani method wins there is so very little difference in the probability of detection scores that the plotted points for fayyadIrani generate on top of or very close to the scores for the other methods. The DiscTree algorithms still performs visibly well in the auto-mpg data set, and other methods visibly win in one or two other data sets, but in most cases there is very little difference between the plots.

Data Set	*Instances*	*Winner(s)*	*Degree*	*Numeric*	*Classes*
hayes-roth	132	cat/fayyadIrani	1(1)	1	3
hepatitis	155	fayyadIrani	1(1)	6	2
flag	194	cat/disctree3/fayyadIrani/pkid	1	10	4
heart-c	303	disctree3	1	6	5
ecoli	336	cat	2(4)	7	8
auto-mpg	398	disctree3	3	5	3
wdbc	569	fayyadIrani	2(2)	30	2
credit-a	690	disctree3/fayyadIrani	1	0	2
diabetes	768	fayyadIrani	4(4)	8	2
vowel	990	fayyadIrani/tbin	1(3)	10	11
segment	2310	fayyadIrani	1(2)	19	7
waveform-5000	5000	fayyadIrani	2(4)	40	3
letter	20000	disctree3/fayyadIrani/pkid	1	16	26

Figure 5.48: These data sets had a particular winner(s) for their Probability of Detection comparison. In all cases, degree measures the number of wins over the next closest method. In the event that disctree3 did not win, the number in parenthesis represents its win difference from the lead method.

Method	*Wins*
fayyadIrani	10
disctree3	5
tbin	1
pkid	2
cat	3

Figure 5.49: Total Wins Per Method Based on Mann-Whitney U-Test Wins on Each Data Set's Probability of Detection Scores

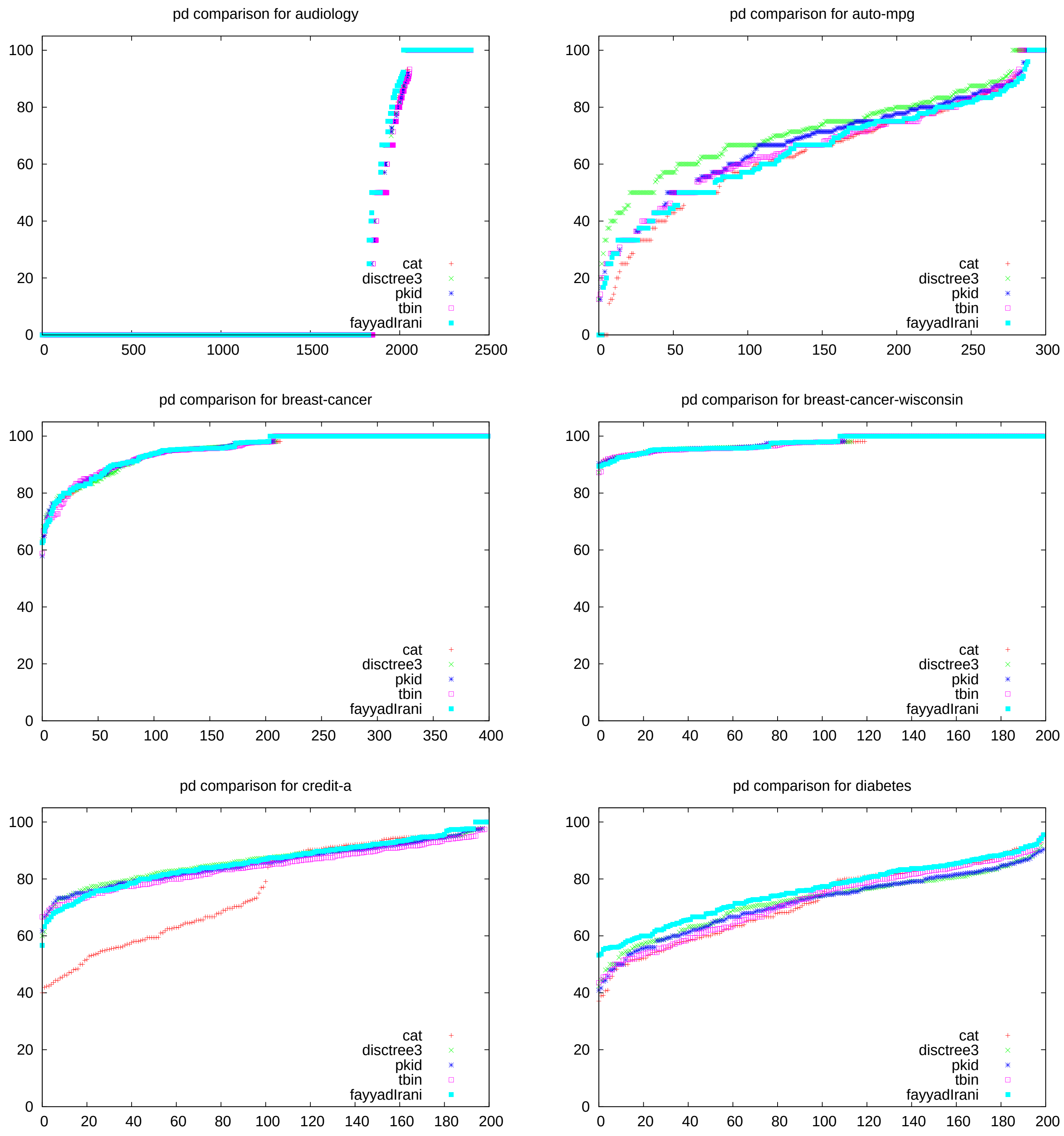


Figure 5.50: Plots of Probability of Detection Scores, Sorted by Value

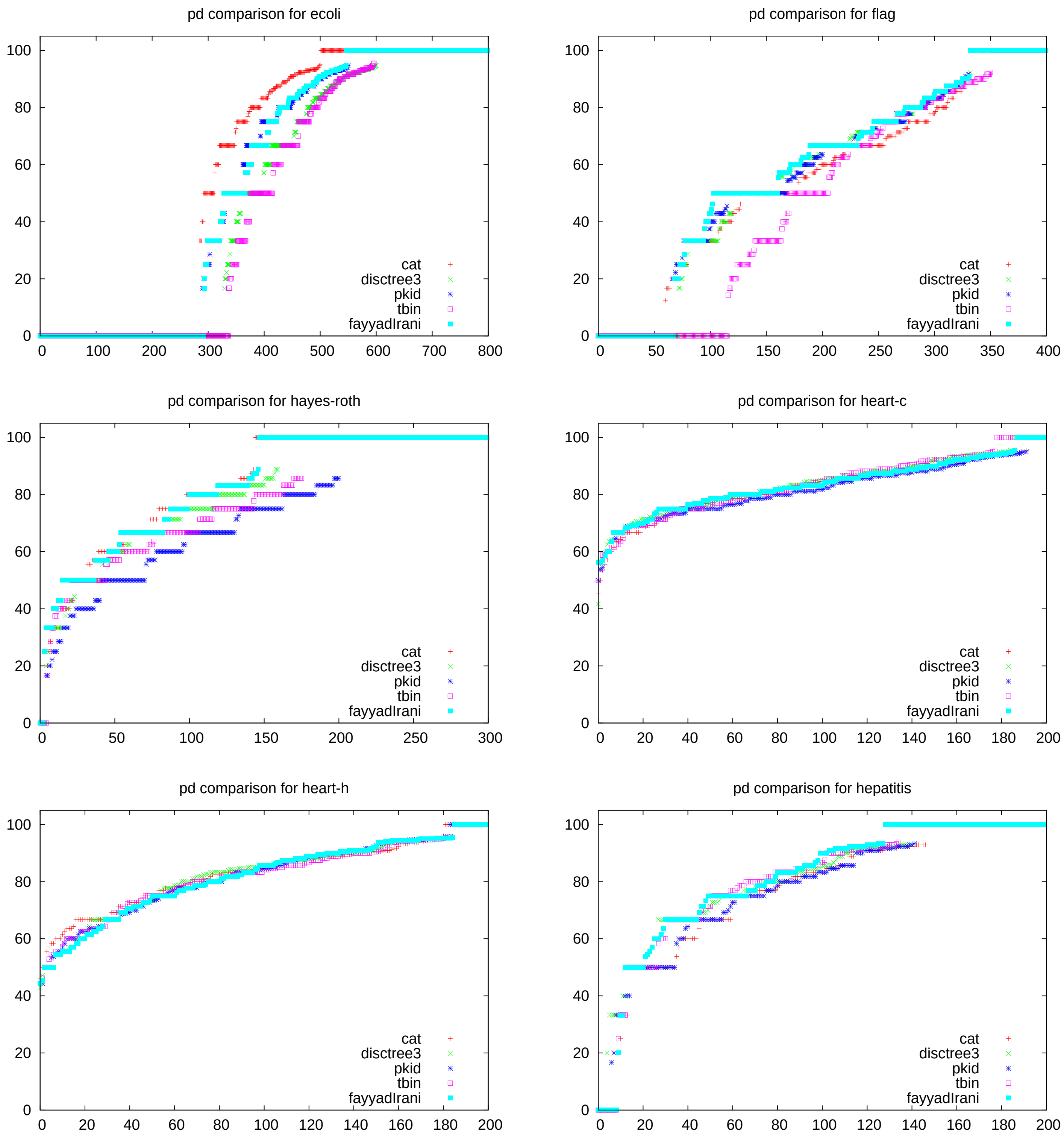


Figure 5.51: Plots of Probability of Detection Scores, Sorted by Value

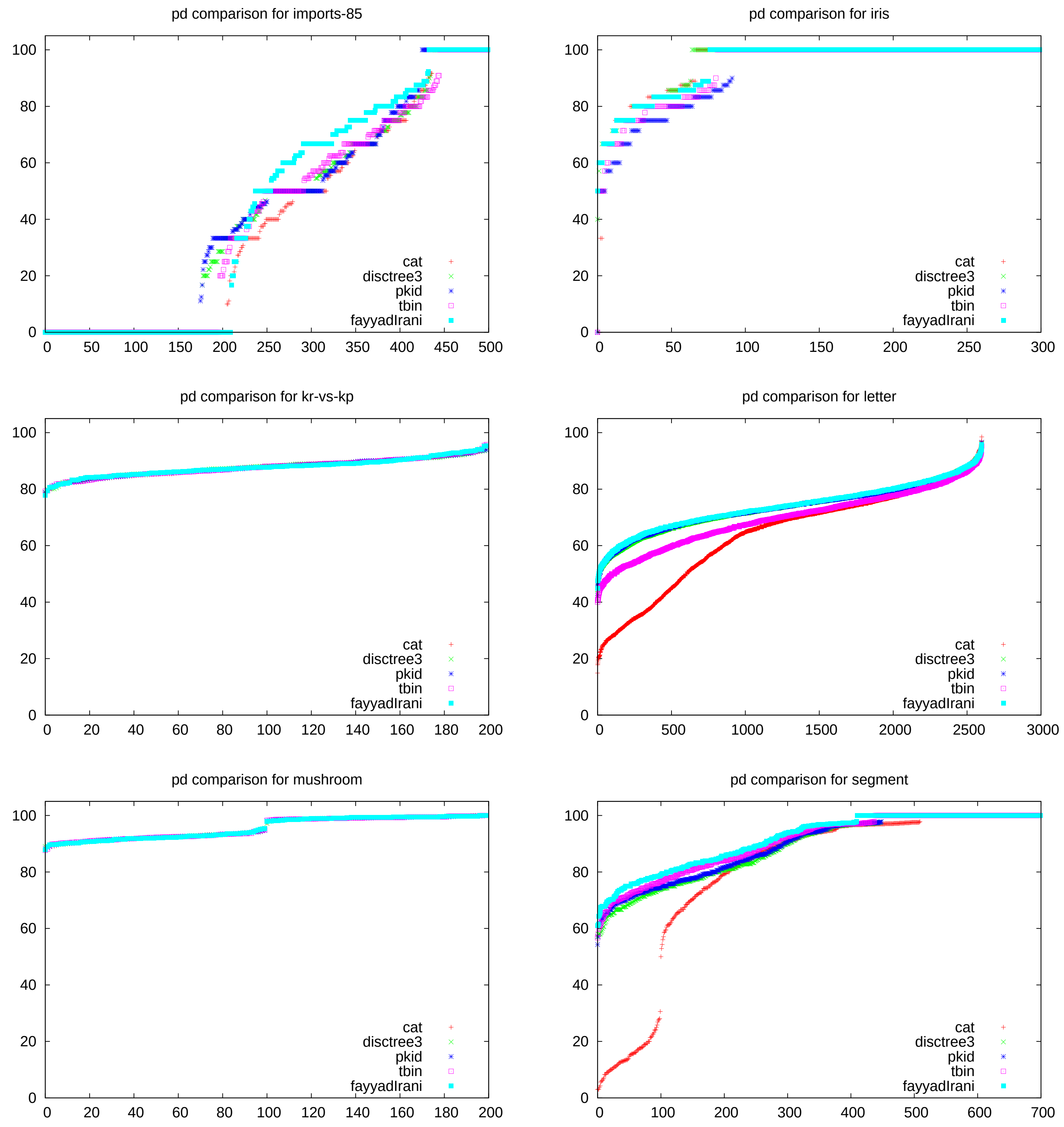


Figure 5.52: Plots of Probability of Detection Scores, Sorted by Value

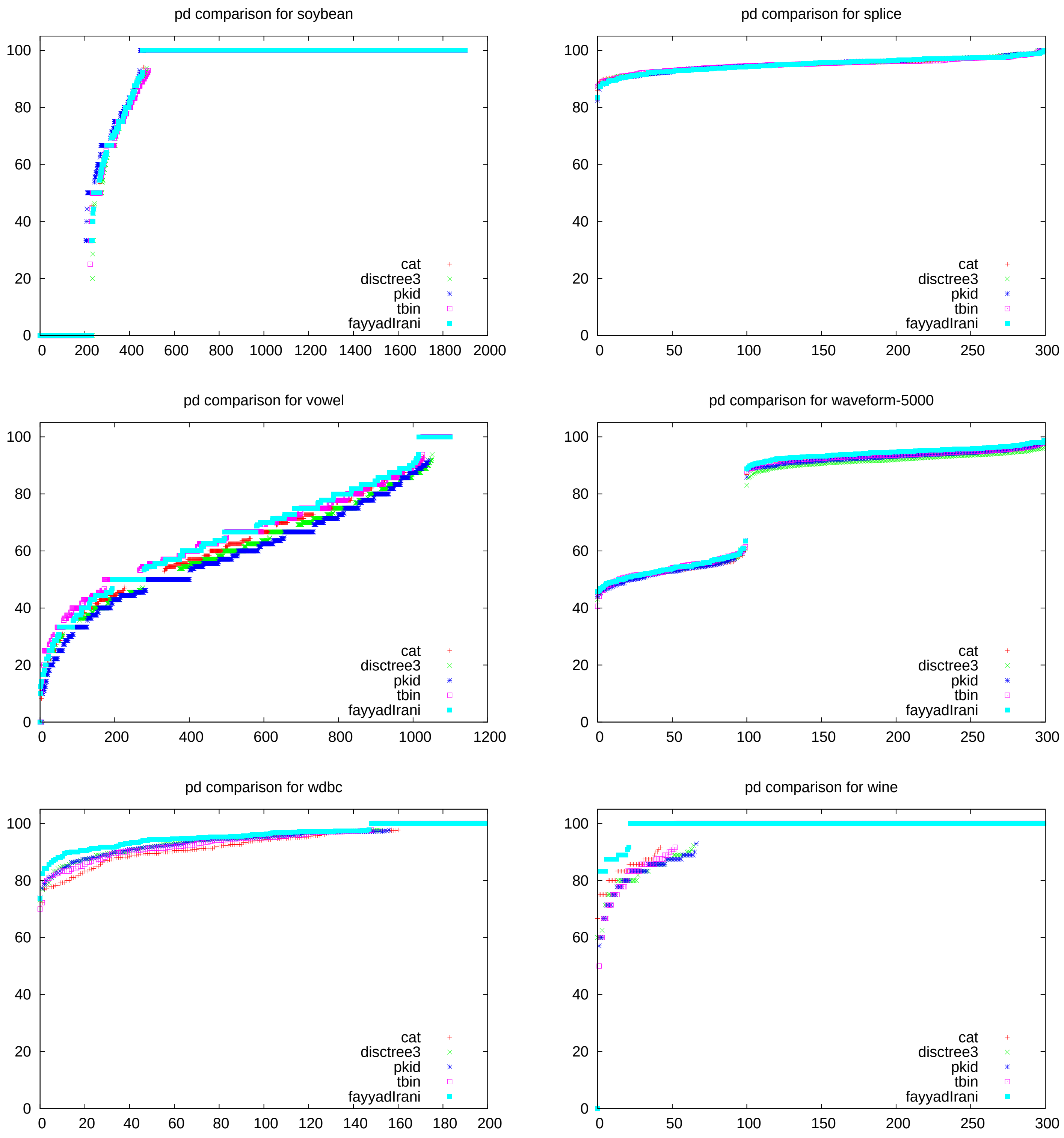


Figure 5.53: Plots of Probability of Detection Scores, Sorted by Value

5.2.5 Probability of Not False Alarm

The overall results for the probability of not false alarm comparison between the discretization methods discussed in Chapter 3, across all the data generated, can be found in Figure 5.54

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	3	1	0	2
pkid	0	2	2	-2
disctree3	0	2	2	-2
cat	0	2	2	-2

Figure 5.54: overall for npf

The results of this overview shows that in comparison with all the other discretization methods, the entropy-minimization method won over each other methods. The tenbins algorithm performed next best, winning a comparisons against each method besides fayyadIrani and losing just to that method. The pure data set, DiscTree, and pkid each performed poorly in this comparison, losing out to each other method.

Figure 5.54 provides a general summary of the results. However, each data set has also had the win/loss/tie record created. Those results can be found in Figure F.97 to Figure F.118. Figure 5.55 displays the winning method(s) for each data set where all the discretization methods did not tie. The winner or winners for each dataset are displayed, along with the number of degrees by which they win. The counts for total wins are found in Figure 5.56. Again, the entropy-minimization heuristic had the most wins, while the DiscTree method won about half as often. Interestingly, there is no clear manner by which to describe the conditions under which the DiscTree algorithm wins, other than that the data sets where it is recorded as a winner tend to have fewer classes (typically less than 5). However, it also loses to the fayyadIrani method in similar data sets.

Another interesting result is the sorted data plots for each data set. These plots, found in Figure 5.2.5 to Figure 5.2.5, show that in most of the data sets where the fayyadIrani method wins, there is so very little difference in the not probability of failure scores that the plotted points for fayyadIrani, generate on top of or very close to the scores for the other methods.

Data Set	*Instances*	*Winner(s)*	*Degree*	*Numeric*	*Classes*
hayes-roth	132	cat	1(1)	1	3
hepatitis	155	fayyadIrani	1(1)	6	2
flag	194	disctree3/fayyadIrani/pkid/tbin	1	10	4
imports-85	205	fayyadIrani	2(3)	15	5
heart-c	303	disctree3	1	6	5
auto-mpg	398	disctree3	3	5	3
wdbc	569	fayyadIrani	2(2)	30	2
credit-a	690	disctree3/fayyadIrani	1	0	2
diabetes	768	fayyadIrani	4(4)	8	2
vowel	990	cat/fayyadIrani/tbin	2(2)	10	11
segment	2310	fayyadIrani/tbin	2(2)	19	7
waveform-5000	5000	fayyadIrani	1(2)	40	3
letter	20000	disctree3/fayyadIrani/pkid	1	16	26

Figure 5.55: These data sets had a particular winner(s) for their not Probability of Failure comparison. In all cases, degree measures the number of wins over the next closest method. In the event that disctree3 did not win, the number in parenthesis represents its win difference from the lead method.

Method	*Wins*
fayyadIrani	10
disctree3	5
tbin	3
pkid	2
cat	2

Figure 5.56: Total Wins Per Method Based on Mann-Whitney U-Test Wins on Each Data Set's not Probability of Failure Scores

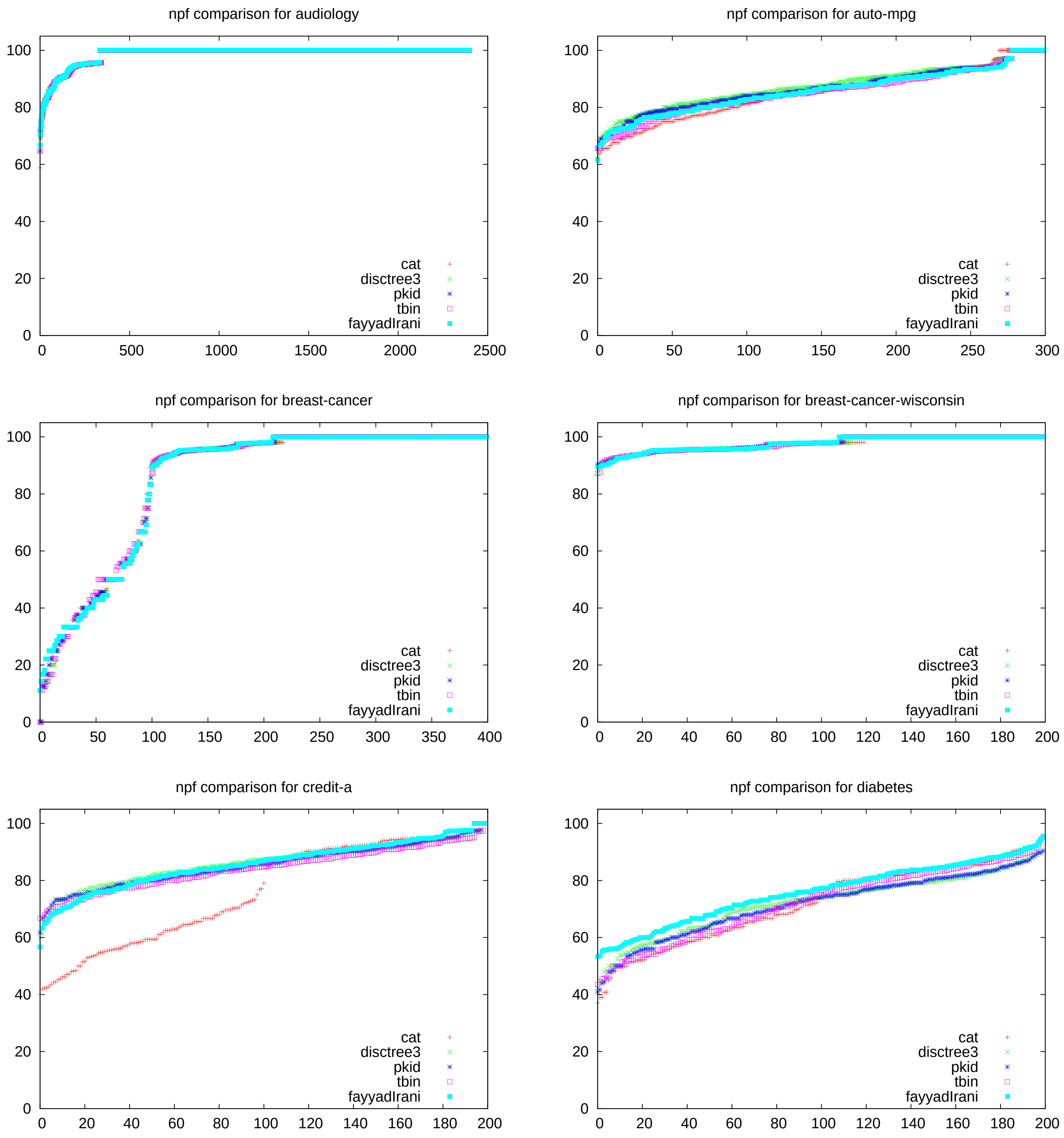


Figure 5.57: Plots of Probability of not False Alarm Scores, Sorted by Value

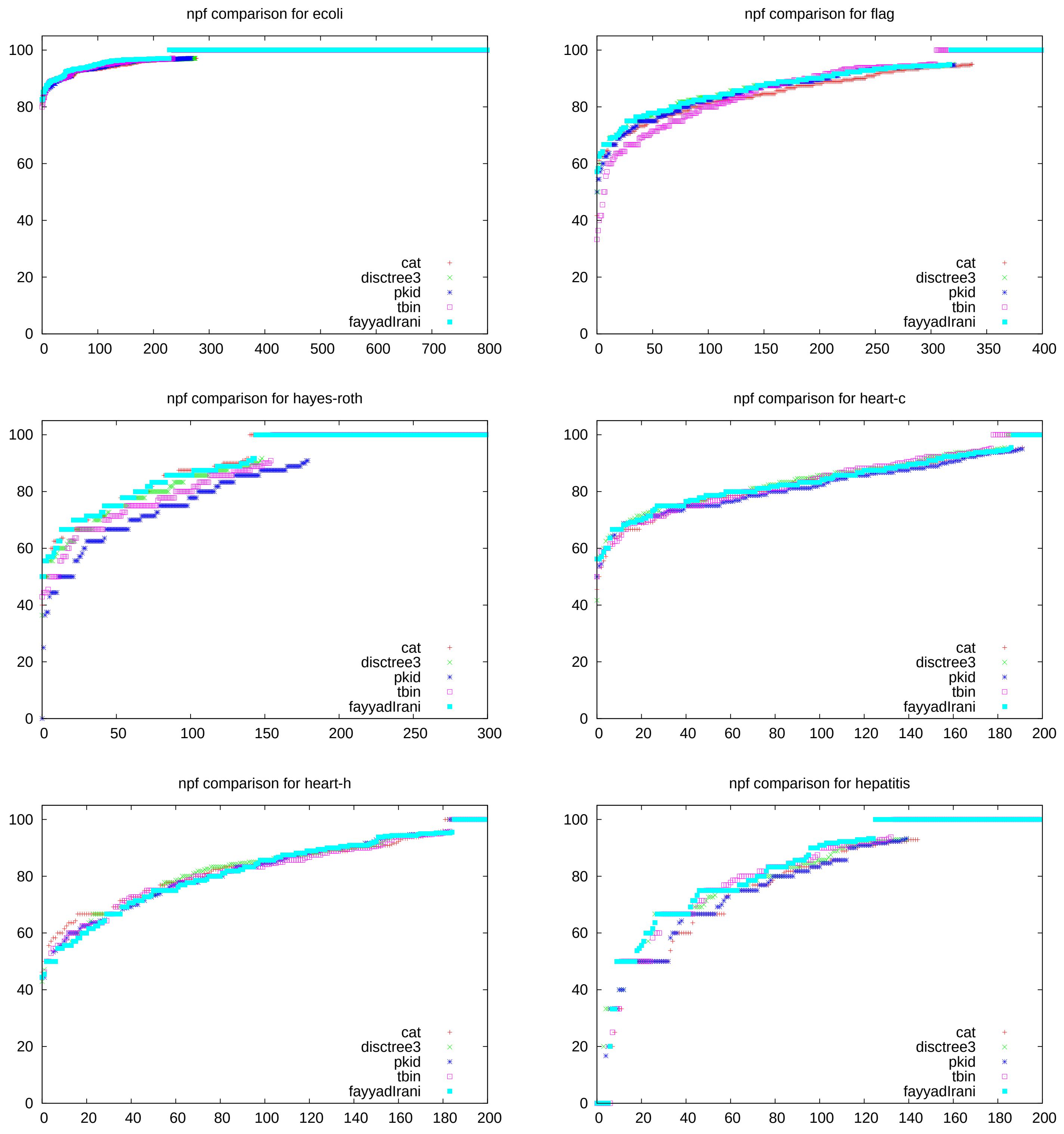


Figure 5.58: Plots of Probability of not False Alarm Scores, Sorted by Value

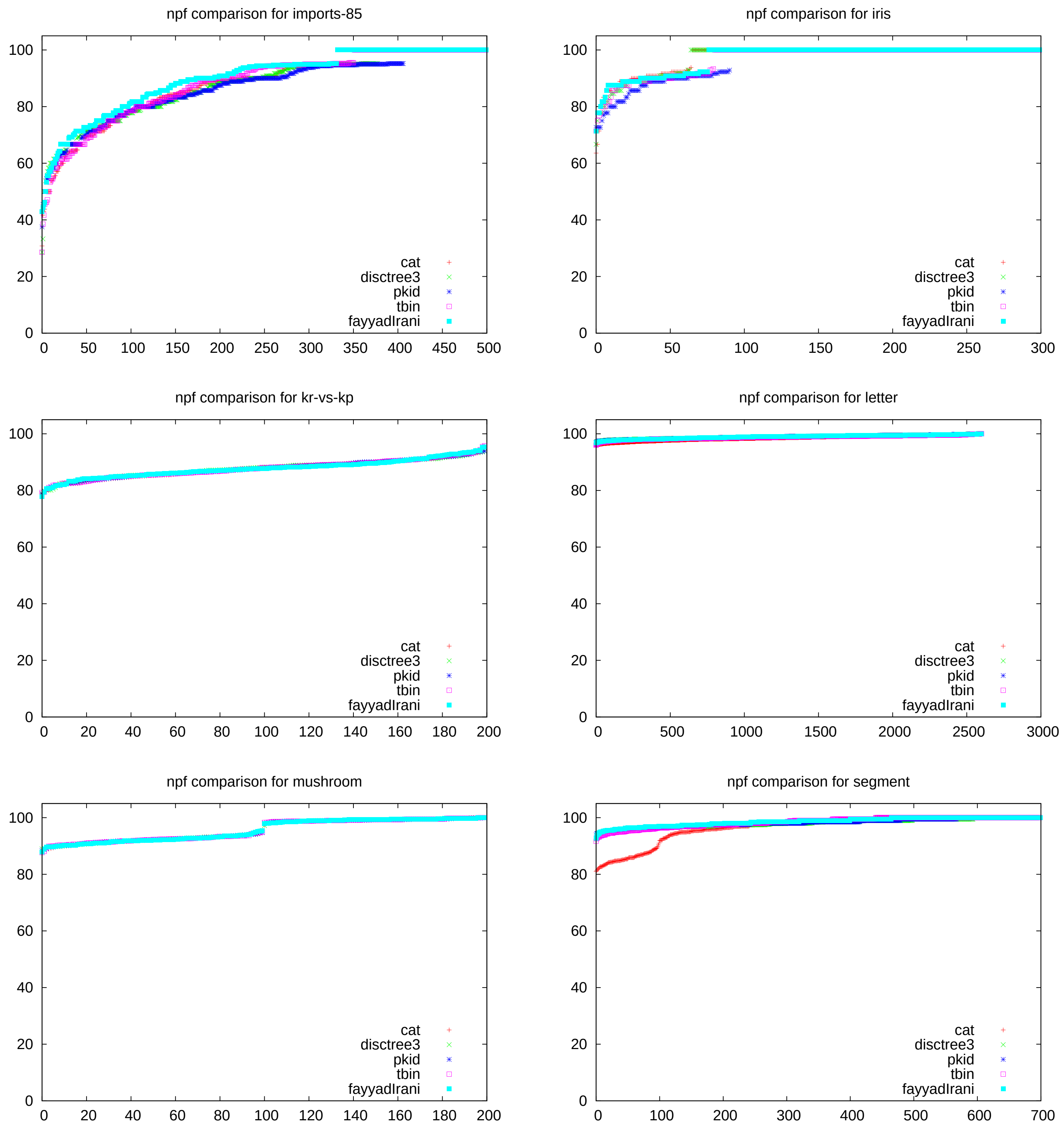
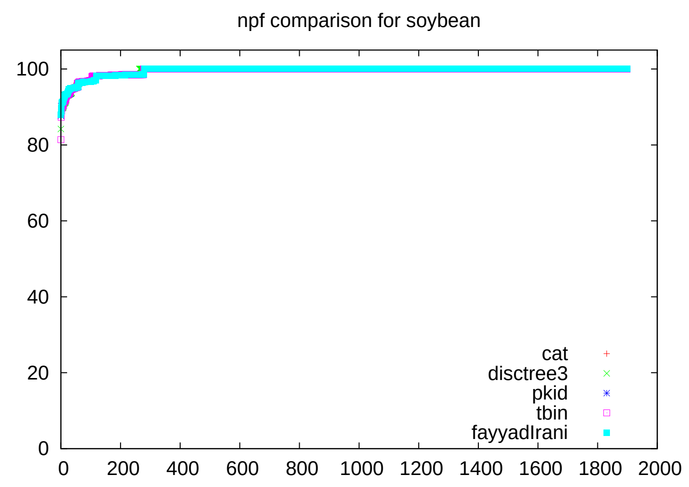
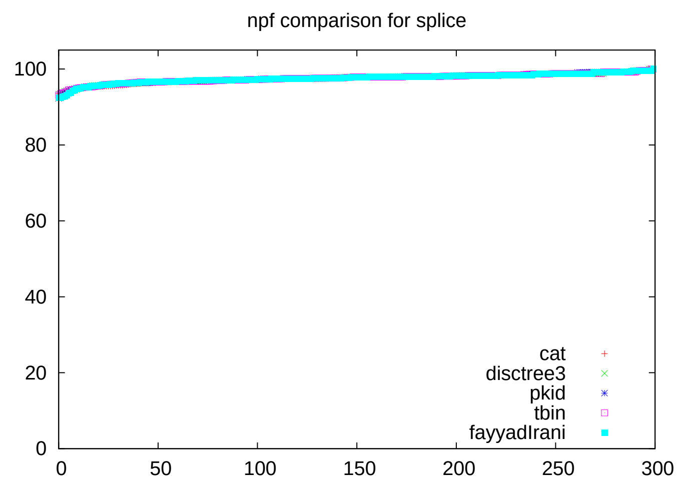
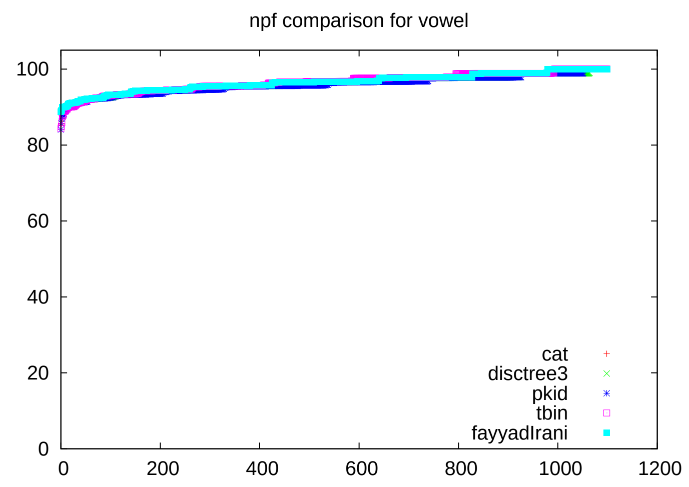
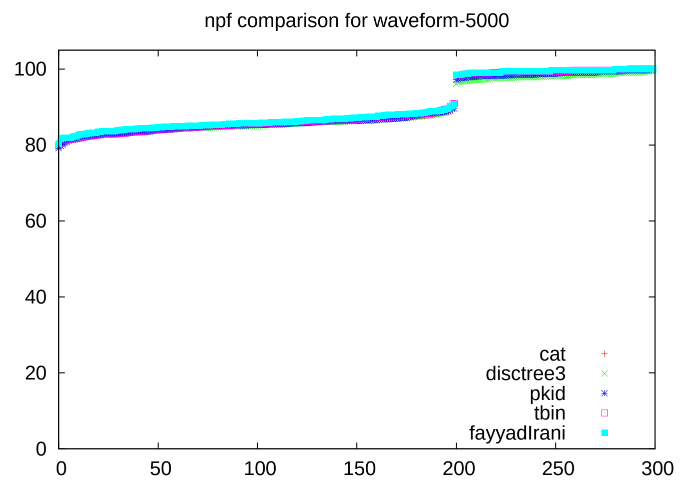


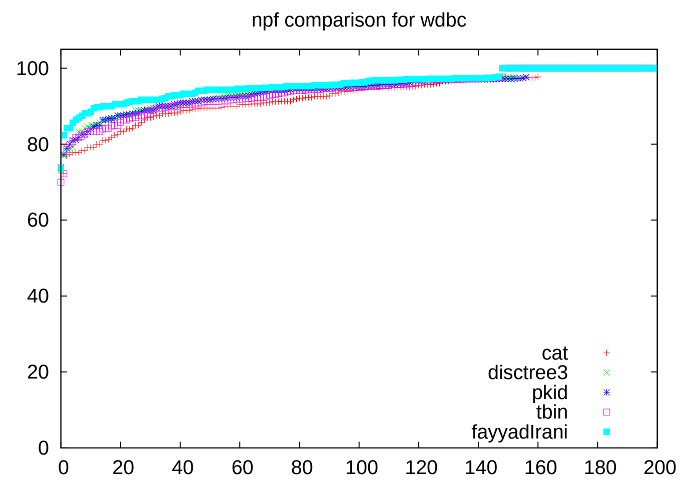
Figure 5.59: Plots of Probability of not False Alarm Scores, Sorted by Value











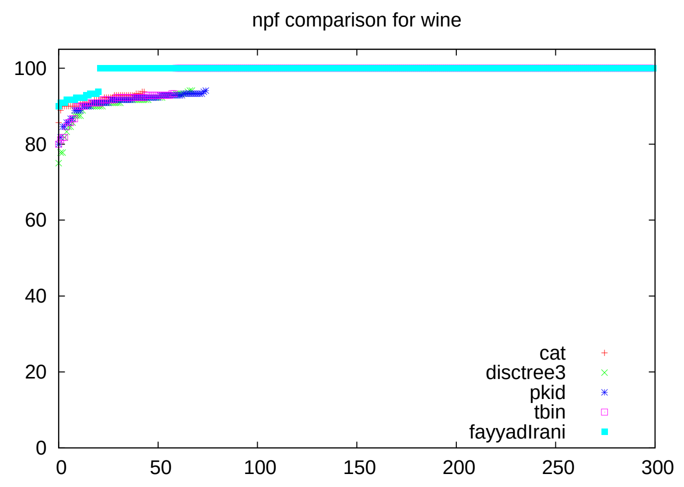


Figure 5.60: Plots of Probability of not False Alarm Scores, Sorted by Value

5.3 Summary

Our results for the discretization method comparison are what could be called a mixed bag. For each method used, the entropy-minimization heuristic won most often when just viewing the results of the Mann-Whitney U-test scores. This is consistent with results from [12, 14], which both found the method proposed in [10] to have the best effect on the Naïve Bayes classifier.

However, it is notable that for each method, most of the plots of the data points appear very similar, with small differences between the data points plotted for the fayyadIrani method and the other methods it was compared against.

Also notable is the considerable win that the DiscTree algorithm had for each measure in the auto-mpg data set. That data set, described in Figure 5.61, consistently had DiscTree winning by a degree of 3. DiscTree also performed well against the PKID method proposed in [25]. In each measure, DiscTree consistently had more wins than PKID (and in most cases the other methods). This is notable because of the methods compared, PKID is the method proposed most recently and provides the basis for WPKID, NDD, and WNDD that could not be tested in this paper.

Data Set	*Instances*	*Attributes*	*Numeric*	*Nominal*	*Classes*
auto-mpg	398	7	5	2	3

Figure 5.61: Data Set Information for auto-mpg

The results in this section will be reviewed in Chapter 6.

Chapter 6

Conclusion

Chapter 6 begins with Section 6.1 which is a broad overview of the previous content of this document. Section 6.2 describes conclusions reached based on the results of the experiments described previously in this document. Section 6.3 proposes additional open research questions which have been brought to light by the work and experimental results of this thesis.

6.1 Overview

In this thesis we have reviewed the broad field of data mining. We identified classification as a useful tool to predict future occurrences based on current evidence and previous information. We reviewed classification methods, including decision tree learners and Naïve Bayes classifiersin order to discuss the differences between available methods of performing this useful task. We then identified Naïve Bayes classifiersas the ideal method for comparing discretization methods, as previous results from numerous sources have shown them to be prone to vast improvement when used with discretization.

We discussed numerous methods of discretization, identifying their strengths and weaknesses and selecting a few of them for comparison against our newly proposed method, the DiscTree algorithm. We introduced this algorithm in this paper and have implemented it. We proposed comparing it to the previously described and selected methods in order to determine how it performed relative to them. We discussed our experimental method, specifically using each of the selected discretization methods and the raw data from 25 data sets in a ten by ten-fold stratified cross-valdiation to obtain the classifier performance measures of *acc*, *bal*, *pd*, *!pf*, and *prec*. We then compared the results of each of these performance measures across each classifier to obtain the results provided in the form of win-loss tables.

We will now discuss our conclusions based on the results collected and provide possible future avenues to continue this research.

6.2 Conclusions

The results of our discretization method comparison appear to align with the results from [14] and [12]. The entropy-minimization heuristic, encapsulated in the script *fayyadIrani* (see Appendix E) clearly wins when the U-test statistics are reviewed. For each measure, it does at least twice as many times as any of the other methods to which it was compared.

However, despite the results of the U-test statics, it is very interesting to note the data plots provided for each measure for each data set. These plots show that, even in most of the cases where fayyadIrani wins, it is doing so almost marginally, with other methods plotting very close to it - so close the that fayyadIrani data points, plotted last, often plot over the other methods. There are some clear instances where this isn't true - for example, the auto-mpg results which always show DiscTree as a clear winner and various results for each measure where fayyadIrani or another method clearly out pace the other methods - but in the majority of cases it appears that the differences between the scores for each method or just slight enough to qualify as different when compared using the U-test statistic.

Of interest to this particular thesis is the results for the DiscTree Algorithm. For each measure, DiscTree performs second-best behind fayyadIrani and almost always better than all the other methods. This is interesting because it often wins out over the PKID method, the most recently proposed of its competition. That method had been shown to perform better in terms of minimized error than the fayyadIrani method in other cases [25]. We did not have similar results; PKID obtains fewer accuracy wins than fayyadIrani and does not even perform as well as DiscTree. This leads us to believe that our method, while clearly not most superior in this comparison, is clearly not an idea that should just be discarded.

The data plots lead us to question whether there exists a continuing need to create better discretization methods. While there may be ways yet unidentified to increase accuracy beyond the entropy-minimization heuristic, the plots of the data values for each performance measure are so close to one another in most cases that perhaps this goal of finding a ”better” discretization method can end with a statement that, so long as a discretization method is used it should better the performance of the Naïve Bayes classifier in most cases. There are a few cases where one discretization method performs better than another, but overall, they perform very similarly. We therefore conclude that discretization is a useful technique for aiding Naïve Bayes classifiers, but we can not put forth one particular method that we would always recommend, except to say that we believe that the methods compared herein perform very similarly in most cases and would recommend using simple methods when possible for easy of explanation.

6.3 Future Work

Our research has led us to a few possible future paths for research in this area.

An immediate possibility from our work would be the modification of DiscTree discretization method to create an incremental method of discretization. Incremental discretization involves data being discretized as each new instance (or a very small group of instances) arrive, rather than requiring all the data as in the batch approach currently used. Incremental discretization methods are very useful because they can be used on streams of data as well as on the files of instances used with the current batch method. For instance, incremental discretization methods could be used as part of a learning mechanism in a problem that involves collecting data online or from a real time source and analyzing it as it arrives, rather than performing the learning operations on static, previously created files stored on a disk.

A second possible area of future research would be in expanding the DiscTree algorithm to examine other possible tree structures that may improve some aspect of the DiscTree algorithm's performance. Specifically, we have identified splay trees and AVL trees as possible alternatives for the base data structure in the DiscTree algorithm. The implementation of a Randomized Binary Search Tree was the easiest to implement at the inception of this idea of a base sorting data structure behind the discretization and performed fairly well when compared to some recommended methods. It is likely that such other data structures would have similar or better performance. Splay trees may be most interesting because their re-balancing of the tree in accordance with the most accessed values may help to prevent any useful nodes from being pruned during the garbage collection activities of the DiscTree algorithm.

A third possible area of future research would be combining the DiscTree algorithm with some additional forms of data preprocessing, such as attribute subset selection. This technique involves using machine learning algorithm to determine the best combinations of attributes and pruning the data to only contain those attributes. This might be useful because experimental results have shown that learners may become confused or distracted in their classification schemes by irrelevant attributes [22]. Pruning the data prior to running it through the DiscTree algorithm would not increase the method's performance in terms of classification results. It would, however, save the algorithm time spent in building trees for the unused attributes. Another option for preprocessing prior to use of the DiscTree algorithm would be a review of over-sampling and under-sampling techniques for the data provided to the discretization method. In either case the data is examined for notable instances. In over-sampling, those instances are then repeated numerous times in the data used by the discretization method/classifier. In under-sampling, the instances of interest are used to prune instances from the data set to avoid or minimize their affect on the discretization method/classifier. Both could be potential boons for DiscTree if they could be harnessed to remove the worst or multiply the best examples that DiscTree should learn from. Thus, improvements with additional preprocessing might be made to make DiscTree perform better.

A final possible area of future research would be examining the DiscTree Algorithm to determine if there is a way to best decide the "cludged" values of that algorithm, including:

- The number of instances required between garbage collects
- The maximum allowable tree size
- The number of instances required between increments of maximum allowable tree size
- The number of instances per node/subtree required to make it worthy of being used in the

discretization.

It is possible that finding optimum values for these items could improve the performance of Disc-Tree against its competition.

Appendix A

disctree Source Code

```
#!/usr/bin/gawk -f
#/* vim: set filteype=awk : */ -*- awk -*-

# A Design for A Randomized Binary
# Search Tree Discretization Method
# By DJ Boland

# Multipurpose implementation.  Allows dynamic tree sizing, tree size, and
# variable node size requirement for substitution

#--------------------------------------------------------------------------------

BEGIN{
        FS=" *, *";             #Fields seperated by a comma and its preceeding
        OFS=",";                # and trailing spaces.
        SUBSEP="#"              #An pound sign will be used as the default seperator
        IGNORECASE=1;           #Set interpreter to be case-insensitive
        Val="val";              #Substitute the string "val" where Val used
        Left="left";            #Substitute the string "left" where Left used
        Right="right";          #Substitute the string "right" where Right used
        Here="here";            #Substitute the string "here" where Here used
        All="all";              #Substitute the string "all" where All used
        Min="min";              #Substitute the string "min" where Min used
        Max="max";              #Substitute teh string "max" where Max used
        srand();                #Randomize the seed used to generate random numbers
        BuildTree=0;            #By default, don't add records to the trees
                                #0 - don't add; 1 - add.
        CurrentRecord=0;        #Record currently being accessed
        Pass=0;                 #Contains the current pass being made through data
        LastLine="";            #Contains last line seen from the file being read
        MaxSize=7;              #The maximum size to allow the tree to grow to
        ReqRecs=35;             #The minimum number of records that must be seen
                                #between garbage collections
        Csize=0;                #Size fo the tree currently being manipulate
        Mode=0;
        RecordsIn=0;            #Number of records seen thus far
        IncSize=75;             #Number of records required before tree size grows
        GrowthMult=1;
#       Special                 #If special equals 1, print Trees on discretization finish
        DynTree=0;              #DynTree = 1 allows dynamic tree size; else static
        SizeChoice=0;           #SizeChoice = 1 use size req of n/3; else sqrt(n)
}

#Skip blank and comment lines
           {sub(/\%.*/,"")}         #Substitute comments with blanks
           {gsub(/[\'\"\`]/,"",$0)} #Remove any single or double quotes from this line
           {gsub("'", "", $0)}      #Remove any single quotes
/^[ \t]*$/ {next}                   #When blank line observed, go to the next line


#PROCESS RELATION LINES
/@RELATION/{sub(/@RELATION/, "@relation")}
/@relation/{
   BuildTree=0; #Set flag for parsing lines to remember (add to tree) to false
   Attr=0;      #Set attribute counter to zero
   LastLine=""; #Set the LastLine seen to nothing
   RecordsIn=0;
   if(Pass==2){
     for (I in Trees)
       {
         if (CurrentSize[I] > MaxSize)
```

```
          {
            garbageCollect(CurrentRoot[I], Rbst, MaxSize)
          }
        }
      print $0
    }
    #COMMENTED CODE can be used to verify that the first pass created the BSTs for the numeric
    #attributes and that it maintained class counts.  It prints the tree to the screen for review
    #if (Pass==2){
    #     for (I in Numeric)
    #     {
    #         allvalues=""
    #         for (J in Classes){allvalues = allvalues " " Classes[J]}
    #         print "Classes are: [" allvalues "]"
    #         printtree(CurrentRoot[I], Rbst,"","", Classes)
    #         print ""
    #     }
    #}
}

#PROCESS ATTRIBUTE LINES
/@ATTRIBUTE/ {sub(/@ATTRIBUTE/, "@attribute")}
/@attribute/ {
  #On the first pass, determine which attributes should be put into trees
  #while preventing the class attribute from being put into the tree.
  if(Pass==1)
  {
    if (LastLine != "")
    {
      Attr++;     #For each new attribute, increase attribute counter

      # Obtain the name of each attribute and
      # store to the Names array
      gsub("_", "", LastLine)
      split(LastLine,a," ")
      Names[Attr]=a[2]

      #Create a list of the fields that should be put into trees
      Attrs[Attr]=1
      if ((match(LastLine, /numeric/) > 0) || (match(LastLine, /real/) > 0) || (match(LastLine,/integer/) > 0))
      {
        Trees[Attr]=1
      }
    }
    LastLine=$0  #Maintain a copy of the last attribute read, as if the @data
                 #symbol is seen then the LastLine attribute is the class
                 #attribute
    next
  }

  #On the second pass, replace the original attribute values with an
  #accurate list of the discretized classes
  if(Pass==2)
  {
    if(LastLine != "")
    {
      Attr++
      if (Attr in Trees)
      {
        outclasses = ""
        if (SizeChoice==1)
        {
          outclasses = discClasses(Rbst, CurrentRoot[Attr], int((Rbst[CurrentRoot[Attr], All])/3), outclasses)
        }
        else
        {
          outclasses = discClasses(Rbst, CurrentRoot[Attr], int(sqrt(Rbst[CurrentRoot[Attr], All])), outclasses)
        }
        sub(LastLine, "@attribute " Names[Attr] " {"outclasses"}", LastLine)
      }
      print LastLine
    }
    LastLine=$0
    next
  }
}


#PREPARE TO PROCESS DATA
/@DATA/{sub(/@DATA/, "@data")}
/@data/{
  BuildTree=1;       #The point has been reached where data should be put in tree
  CurrentRecord=1;  #The next nonblank line read will be the first record
  #On first pass, determine the the classes and set up the trees
  if (Pass==1)
  {
    #Parse last line to obtain classess
    sub(/\}$/, "", LastLine)                     #Prune closing bracket from class
    sub(/@attribute /,"", LastLine)              #Prune @attribute sybmol from class
    sub(/^[[:alnum:][:blank:]]*\{/, "", LastLine) #Prune class name and opening bracket from classes
    gsub(/[ \t\n]*/, "", LastLine)               #Prune any extraneous spaces from classes
    split(LastLine, myClasses, ",")              #Split remaining information into different classes

#    COMMENTED CODE prints each class and creates a list of all the classes and prints it
#     for (J in myClasses){print myClasses[J]}
#     allvals = ""
#     for(J in myClasses){if(allvals==""){allvals=myClasses[J]}else{allvals = allvals " " myClasses[J]}}
#     print allvals

    #Initialize the roots of each tree to sybolize an empty tree, and new records for each tree to 0
    for(I in Trees){CurrentRoot[I] = "?"; CurrentSize[I]=0; NewRecords[I]=0}

  }

  #When outputting pruned dataset, ensure keeping the @data line
  if (Pass==2) {print LastLine; print $0}

  #Do not want to proceed to put the @data symbol in the tree, so go to the next line
  next
}

#PROCESS DATA / BUILD TREES FROM DATA
#PASS ONE creates the binary search tree for each (numeric) attribute
Pass==1 && BuildTree && NF >1{
  for(I in Trees)              #For each non-class attribute
  {
    NewRecords[I]++            #Add one to the number of records seen since last garbage collect
    field=$I;                  #field set to the value of the attribute in the row currently being processed
    CSize=CurrentSize[I]       #set CSize to the size of the current tree

    #If missing data, decrease new record count and go to next field
    if(field ~ /?/) {NewRecords[I]--; continue}

    #Otherwise, data is valid, insert into the tree
    else returnvalues=rbstinsert(CurrentRoot[I], Rbst, field, Names[I], CurrentRecord,$NF, myClasses, CSize)

    #Split the returned value into the current root and size of the tree
    #that was just processed
    split (returnvalues, vals, SUBSEP)
    CurrentRoot[I]=vals[1] SUBSEP vals[2]
    CurrentSize[I]=vals[3]

    #If the tree exceeds the Maximum size and the minumum numer of records
    #between garbage collections has been seen, Garbage Collect!
    if ((CurrentSize[I] > MaxSize)  && (NewRecords[I] >=ReqRecs))
    {
      garbageCollect(CurrentRoot[I], Rbst, MaxSize)
      NewRecords[I]=0
    }
    RecordsIn++;
    if(DynTree==1)
    {
      if(RecordsIn >= (IncSize * GrowthMult))
      {
        GrowthMult++;
        MaxSize= MaxSize + 2^(2+GrowthMulti)
      }
    }
  }

  #Increase the counter of the number of records that have been processed
  CurrentRecord++;
}

#PASS TWO substitutes the discretized value from the search tree for each value/attribute
Pass==2 && BuildTree && NF >1{
  for(I in Trees)                 #for each non-class attribute
  {

    field=$I                      #field set to the value of the attribute in the row currently being processed
    RootInstance=CurrentRoot[I]   #Set RootInstance to the root instance of the attribute currently being processed
    if(field ~ /?/) continue      #If missing data (symbolized by ?), go to the next field

    #Otherwise, data is valid, determine what discrete value to substitute with
    else
    {
      if(SizeChoice==1)
      {
        $I=subwith(int((Rbst[RootInstance,All])/3),field, Rbst,CurrentRoot[I])
      }
      else
      {
        $I=subwith(int(sqrt(Rbst[RootInstance,All])),field, Rbst,CurrentRoot[I])
      }
    }
  }

  #Print the parsed lines
  print $0

}


END{
  if (Special==1)
  {
    for (I in Trees)
    {
      print ""
      allvals = ""
      for(J in myClasses){if(allvals==""){allvals=myClasses[J]}else{allvals = allvals " " myClasses[J]}}
      print "Classes are: [" allvals "]"
      printtree(CurrentRoot[I], Rbst,"","", myClasses)
      print ""
    }
  }
}

#****************************************************************************
#bst.awk
#DJ's Binary Search Tree Functions

#Insert a a new node at the root of the tree
function insertR(RootInstance, Tree, Value, Attr, Rec, Class, Classes, CurrentSize)
{
  if (RootInstance == "?")
  {
    return newtree(Attr SUBSEP Rec, Tree, Value, Class, Classes, CurrentSize)
  }
  Tree[RootInstance,All]++
  Tree[RootInstance,"A" SUBSEP Class]++

  if (Value == Tree[RootInstance,Val])
  {
    Tree[RootInstance,Here]++
    Tree[RootInstance,"h" SUBSEP Class]++
    return RootInstance SUBSEP CurrentSize
  }

  if (Value < Tree[RootInstance,Val])
  {
    returnvals=insertR(Tree[RootInstance,Left], Tree, Value, Attr, Rec, Class, Classes, CurrentSize)
    split(returnvals, vals, SUBSEP)
    Tree[RootInstance,Left]=vals[1] SUBSEP vals[2]
    CurrentSize=vals[3]
    RootInstance = rotateR(RootInstance,Tree,Classes)
  }
  else
  {
    returnvals=insertR(Tree[RootInstance,Right], Tree, Value, Attr, Rec,Class, Classes, CurrentSize)
    split(returnvals,vals, SUBSEP)
    Tree[RootInstance,Right]=vals[1] SUBSEP vals[2]
    CurrentSize=vals[3]
    RootInstance = rotateL(RootInstance,Tree,Classes)
  }
  return RootInstance SUBSEP CurrentSize
}


#Rotate the current root's left subchild to right to make it become the current root
function rotateR(RootInstance, Tree, Classes,     temp)
{
  temp = Tree[RootInstance,Left]
  Tree[RootInstance,Left] = Tree[temp,Right]
  Tree[temp,Right] = RootInstance
  Tree[temp,All] = Tree[RootInstance,All]
  for(M in Classes)
  {
    Tree[temp, "A" SUBSEP Classes[M]] = Tree[RootInstance, "A" SUBSEP Classes[M]]
  }
  if((Tree[RootInstance,Left] != "?") && (Tree[RootInstance,Right] != "?"))
  {
    Tree[RootInstance,All]=\
    Tree[Tree[RootInstance,Left],All]+Tree[Tree[RootInstance,Right],All]+Tree[RootInstance,Here]
    for(M in Classes)
    {
      Tree[RootInstance,"A" SUBSEP Classes[M]]=\
                  Tree[Tree[RootInstance,Left],"A" SUBSEP Classes[M]]+\
                  Tree[Tree[RootInstance,Right],"A" SUBSEP Classes[M]]+\
                  Tree[RootInstance,"h" SUBSEP Classes[M]]
    }
```

```
  }
  else
  {
    if((Tree[RootInstance,Left] != "?"))
    {
      Tree[RootInstance,All] = Tree[RootInstance,Here]+Tree[Tree[RootInstance,Left],All]
      for(M in Classes)
      {
        Tree[RootInstance, "A" SUBSEP Classes[M]] =\
                  Tree[RootInstance,"h" SUBSEP Classes[M]]+\
                  Tree[Tree[RootInstance,Left],"A" SUBSEP Classes[M]]
      }
    }
    else
    {
      if((Tree[RootInstance,Right] != "?"))
      {
         Tree[RootInstance,All] =\
                        Tree[RootInstance,Here]+\
                        Tree[Tree[RootInstance,Right],All]
        for(M in Classes)
                        {
           Tree[RootInstance,"A" SUBSEP Classes[M]]=\
                          Tree[RootInstance,"h" SUBSEP Classes[M]]+\
                          Tree[Tree[RootInstance,Right],"A" SUBSEP Classes[M]]
                        }
      }
      else
      {
        Tree[RootInstance,All] = Tree[RootInstance,Here]
        for(M in Classes)
        {
          Tree[RootInstance,"A" SUBSEP Classes[M]]=Tree[RootInstance,"h" SUBSEP Classes[M]]
        }
      }
    }
  }
  return temp
}



#Rotate the current roots right subchild up to the root (by rotating it to the left)
function rotateL(RootInstance, Tree, Classes,      temp)
{
  temp = Tree[RootInstance,Right]
  Tree[RootInstance,Right] = Tree[temp,Left]
  Tree[temp,Left] = RootInstance
  Tree[temp,All] = Tree[RootInstance,All]
  for(M in Classes)
  {
    Tree[temp,"A" SUBSEP Classes[M]]=Tree[RootInstance,"A" SUBSEP Classes[M]]
  }
  if((Tree[RootInstance,Left] != "?") && (Tree[RootInstance,Right] != "?"))
  {
    Tree[RootInstance,All]=\
    Tree[Tree[RootInstance,Left],All]+\
         Tree[Tree[RootInstance,Right],All]+\
         Tree[RootInstance,Here]
    for(M in Classes)
    {
      Tree[RootInstance,"A" SUBSEP Classes[M]] =\
                Tree[Tree[RootInstance,Left],"A" SUBSEP Classes[M]]+\
                Tree[Tree[RootInstance,Right],"A" SUBSEP Classes[M]]+\
                Tree[RootInstance,"h" SUBSEP Classes[M]]
    }
  }
  else
  {
    if((Tree[RootInstance,Left] != "?"))
    {
      Tree[RootInstance,All] = Tree[RootInstance,Here]+Tree[Tree[RootInstance,Left],All]
      for(M in Classes)
      {
         Tree[RootInstance,"A" SUBSEP Classes[M]]=\
                        Tree[RootInstance, "h" SUBSEP Classes[M]] +\
                        Tree[Tree[RootInstance,Left], "A" SUBSEP Classes[M]]
      }
    }
    else
    {
      if((Tree[RootInstance,Right] != "?"))
      {
        Tree[RootInstance,All] = Tree[RootInstance,Here]+Tree[Tree[RootInstance,Right],All]
        for(M in Classes)
                  {
          Tree[RootInstance,"A" SUBSEP Classes[M]]=\
                                Tree[RootInstance, "h" SUBSEP Classes[M]] +\
                                Tree[Tree[RootInstance,Right],"A" SUBSEP Classes[M]]
                         }
        }
        else
        {
           Tree[RootInstance,All] = Tree[RootInstance,Here]
           for(M in Classes)
                                {
              Tree[RootInstance, "A" SUBSEP Classes[M]] = Tree[RootInstance, "h" SUBSEP Classes[M]]
                                }
        }
     }
  }
  return temp
}


#Function to insert a new node into the tree
function newtree(RootInstance, Tree, Value, Class, Classes, CurrentSize)
{
  print RootInstance
  Tree[RootInstance,Left]="?"
  Tree[RootInstance,Right]="?"
  Tree[RootInstance,Val]=Value
  Tree[RootInstance,All]=1
  Tree[RootInstance,Here]=1

  #Initialize the here and All class counts for each class
  for(M in Classes)
  {
    Tree[RootInstance,"h" SUBSEP Classes[M]]=0
    Tree[RootInstance,"A" SUBSEP Classes[M]]=0
  }

  Tree[RootInstance,"A" SUBSEP Class]=1
  Tree[RootInstance,"h" SUBSEP Class]=1
  CurrentSize++
  return RootInstance SUBSEP CurrentSize
}


#Function to print the current tree
function printtree(RootInstance, Tree, b4, indent, Classes,      allclasses, treeclasses)
{

  allclasses=""
  treeclasses=""
  for(M in Classes)
  {
    if(allclasses==""){allclasses=Tree[RootInstance,"h" SUBSEP Classes[M]]}
    else allclasses=allclasses " " Tree[RootInstance,"h" SUBSEP Classes[M]]
    if(treeclasses==""){treeclasses=Tree[RootInstance,"A" SUBSEP Classes[M]]}
    else treeclasses=treeclasses " " Tree[RootInstance,"A" SUBSEP Classes[M]]
  }
  print indent b4 RootInstance " = " Tree[RootInstance,Val] " [here=" Tree[RootInstance,Here] "] [all=" Tree[RootInstance,All] "]"
  print indent b4 RootInstance " Classes here are: [" allclasses "], all: [" treeclasses "]"
  if(Tree[RootInstance,Left] != "?")
    printtree(Tree[RootInstance,Left], Tree, "< ", indent "|    ",Classes,x)
  if(Tree[RootInstance,Right] != "?")
    printtree(Tree[RootInstance,Right], Tree, "> ", indent "|    ",Classes,x)
}


#Function for inserting a value into the tree. If the value is not at the root,
#this function will, with a 1/n chance, insert the node at the current root.
function rbstinsert(RootInstance, Tree, Value, Attr, Rec, Class, Classes, CurrentSize)
{
  print RootInstance
  if (RootInstance == "?")
  {
    return newtree(Attr SUBSEP Rec, Tree, Value, Class, Classes, CurrentSize)
  }

  if (Value == Tree[RootInstance,Val])
  {
    Tree[RootInstance,Here]++
    Tree[RootInstance,All]++
    Tree[RootInstance, "A" SUBSEP Class]++
    Tree[RootInstance,"h" SUBSEP Class]++
    return RootInstance SUBSEP CurrentSize
  }

  else
  {
    if ((rand()*Tree[RootInstance,All]) < 1.0)
    {
      return insertR(RootInstance,Tree,Value, Attr, Rec, Class, Classes, CurrentSize)
    }
    if (Value < Tree[RootInstance,Val])
    {
      returnvals=\
                rbstinsert(Tree[RootInstance,Left], Tree, Value, Attr, Rec, Class, Classes, CurrentSize)
      split(returnvals, vals, SUBSEP)
      Tree[RootInstance,Left]=vals[1] SUBSEP vals[2]
      CurrentSize=vals[3]
    }
    else
    {
      returnvals=\
                rbstinsert(Tree[RootInstance,Right], Tree, Value, Attr, Rec, Class, Classes, CurrentSize)
      split(returnvals, vals, SUBSEP)
      Tree[RootInstance,Right]=vals[1] SUBSEP vals[2]
      CurrentSize=vals[3]
    }
    Tree[RootInstance,All]++
    Tree[RootInstance,"A" SUBSEP Class]++
    return RootInstance SUBSEP CurrentSize
  }
}


#Function that determines which tree node to substitute for a given value
#For a tree node to be chosen as a viable substitute, it must have at least
#minSize nodes at in itself and/or in its children.
function subwith(minSize, Value, Tree, RootInstance)
{
#   print "This instance is: " Value " and it is being compared to: " Tree[RootInstance, Val]
  if(((Value == Tree[RootInstance,Val]) || (Value ~ Tree[RootInstance,Val]))\
         && (Tree[RootInstance,All] >= minSize))
  {
#       print "They are equal"
    return RootInstance
  }
#   print "not equal"
  if(Value < Tree[RootInstance,Val])
  {
    if (Tree[RootInstance,Left]!= "?")
    {
      if(Tree[Tree[RootInstance,Left],All] >= minSize)
      {
        return subwith(minSize,Value, Tree, Tree[RootInstance,Left])
      }
      else return RootInstance
    }
    else return RootInstance
  }
  if (Value > Tree[RootInstance,Val])
  {
    if (Tree[RootInstance,Right] != "?")
    {
      if(Tree[Tree[RootInstance,Right],All] >= minSize)
      {
        return subwith(minSize, Value, Tree, Tree[RootInstance,Right])
      }
      else return RootInstance
    }
    else return RootInstance

  }
  else
  {
    if(int(2*rand()) > 1)
    {
      if (Tree[RootInstance,Right] != "?")
      {
        if(Tree[Tree[RootInstance,Right],All] >= minSize)
        {
          return subwith(minSize, Value, Tree, Tree[RootInstance,Right])
        }
        else return RootInstance
      }
      else return RootInstance
    }
    else
    {
      if (Tree[RootInstance,Left]!= "?")
      {
        if(Tree[Tree[RootInstance,Left],All] >= minSize)
        {
          return subwith(minSize,Value, Tree, Tree[RootInstance,Left])
        }
        else return RootInstance
      }
```

```
            else return RootInstance
        }
    }
}

#Function to obtain the lists of discretized classes
#To be a part of this list, the given node must
#be of at least minSize
function discClasses(Tree, RootInstance, minSize, myClasses)
{
    if(Tree[RootInstance,Left] != "?")
    {
        myClasses=discClasses(Tree, Tree[RootInstance,Left], minSize, myClasses)
    }
    if(Tree[RootInstance,All] >= minSize)
    {
        if(myClasses != "")
        {
            myClasses = myClasses " , " RootInstance
        }
        else
        {
            myClasses = RootInstance
        }
    }
    if(Tree[RootInstance,Right] != "?")
    {
        myClasses=discClasses(Tree, Tree[RootInstance,Right], minSize, myClasses)
    }
    return myClasses

}

#Function to prune tree to an appropriate size/keep it from exceeding memory availability
function garbageCollect(RootInstance, Tree, MaxSize,          low, high, size)
{
    #Set up a set to track the tree nodes that have been "touched by the breadth first search
    size=0;
    high=1;
    low=0;
    NodeSet[0]=RootInstance

    #Function uses a breadth-first approach to span the tree, adding nodes to the set left to right
    #across each level and then moving down to the following one. When the max size of the tree is
    #reached, the nodes remaining in the set are removed, along with any of their children that were
    #not discovered.
    while (high != low)
    {
        if (Tree[NodeSet[low],Left] != "?")
        {
            NodeSet[high]= Tree[NodeSet[low],Left];
            high++;
        }
        if (Tree[NodeSet[low],Right] != "?")
        {
            NodeSet[high]= Tree[NodeSet[low],Right];
            high++;
        }
        size++;
        low++;
        if (size == MaxSize)
        {
            high = high-1;
            while(low <= high)
            {
                removeVal(RootInstance, Tree, pruneNode(NodeSet[high], Tree))
                high = high - 1;
            }
        }
    }
}

#Function to remove a given node and any of its children from the tree
function pruneNode(RootInstance, Tree)
{
    #If this node has a left child, remove that left child before this node
    if(Tree[RootInstance,Left] != "?")
    {
        pruneNode(Tree[RootInstance,Left], Tree)
    }
    #If this node has a right child, remove that right child before this node
    if(Tree[RootInstance,Right] != "?")
    {
        pruneNode(Tree[RootInstance,Right],Tree)
    }

    #Set all relevant values of this tree to zero and disconnect the childrens nodes
```

```
  Tree[RootInstance,Left]="?"
  Tree[RootInstance,Right]="?"
  Tree[RootInstance,All]=0
  Tree[RootInstance,Here]=0
  return Tree[RootInstance,Val]
}

#Function to remove a node from the tree based on its value
#Starting root node will not be removed using this method
function removeVal(RootInstance, Tree, myVal)
{
  if(Tree[RootInstance,Val] > myVal)
  {
    if(Tree[Tree[RootInstance, Left], Val] == myVal)
    {
       Tree[RootInstance, Left]="?"
    }
    else
    {
      removeVal(Tree[RootInstance,Left], Tree, myVal)
    }
  }
  else
  {
    if(Tree[Tree[RootInstance, Right], Val] == myVal)
    {
       Tree[RootInstance, Right]="?"
    }
    else
    {
      removeVal(Tree[RootInstance, Right], Tree, myVal)
    }
  }
}
```

Appendix B

crossval Source Code

```
#!/bin/bash
# /* vim: set filetype=sh : */ -*- sh -*-

DESTDIR=${DESTDIR="/srv/bronze/dj"}
MyHome=${MyHome=${HOME}} #${HOME}}

# Q0: where will the output be stored
Safe=${Safe=${DESTDIR}/var/weka}
BinDir=${BinDir=${MyHome}/bin}

# Q1: where is your data?
Data=${Data=${MyHome}/var/data/discrete}

# Q2: what data sets will we run?
# A2: only those with discrete classes:
Datums=${Datums="a2b/audiology a2b/auto-mpg
                 a2b/breast-cancer a2b/breast-cancer-wisconsin
                 c2d/credit-a c2d/diabetes e2i/ecoli e2i/flag e2i/hayes-roth
                 e2i/heart-c e2i/heart-h e2i/hepatitis e2i/imports-85
                 e2i/iris j2p/kr-vs-kp j2p/letter j2p/mushroom
                 q2s/segment  q2s/splice q2s/soybean t2z/vowel
                 t2z/wine t2z/wdbc t2z/waveform-5000"}

# Q3: what learners will you try?
Learners=${Learners="bayes"}
bayes() { wttp $1 $2 $wBayes | gotwant ; }

[ -f  "$Functions" ] && . $Functions

# Q4 : what pre-processors will you use
Preps=${Preps="cat disctree3 fayyadIrani pkid tbin"}
disctree3() { dtree3 DynTree=1 Pass=1 $1 Pass=2 $1; }
tbin()  { tenbins Pass=1 $1 Pass=2 $1 Pass=3 $1; }
pkid() { wpkid $1;}

# Q5: how many repeats?
Repeats=${Repeats=10}

# Q6: how many bins?
Bins=${Bins=10}

# All right then. Lets go!
mkdir -p $Safe                  # ensure safe place exists
Tmp=`mktemp -d`                 # make a sandbox where only you will play
trap "rm -rf $Tmp" 0 1 2 3 15   # leave nothing behind when you quit
cd $Tmp                         # go to the sandbox

main() {
  set -x
  for datum in $Datums
  do
        echo "#data,learner,prep,train,test,repeats,bin,goal,a,b,c,d,accuracy,pd,pf,precision,bal"
        for prep in $Preps
        do
          $prep ${Data}/${datum}.arff > data.arff
          for((repeats=1;repeats<=$Repeats;repeats++))
          do
             seed=$RANDOM
             for((bin=1;bin<=${Bins};bin++))
             do
                cat data.arff | someArff --seed $seed --bins $Bins --bin $bin
                goals=`cat data.arff | classes --brief`
                n1=`instances train.arff`
                n2=`instances test.arff`
                for learn in $Learners
                do
                    $learn train.arff test.arff > results.csv
                    cp results.csv $Safe/results.csv
                    for goal in $goals
                    do
                      b4="$datum,$learn,$prep,$n1,$n2,$repeats,$bin,$goal";
                      cat results.csv | ${BinDir}/abcd -g "$goal" -p "${b4}" -d 1 | tail -n 1
                    #abcd Prefix="${b4}" False="${one}" True="${two}"
                    done
done; done; done; done; done
}

Log=$$
Start=`date +%H:%M:%S%t%m/%d/%Y`
(main | tee $Safe/xval.$Log ) 2> $Safe/xval.err.$Log
doExtras $Safe/xval.$Log > $Safe/myresults.$Log.csv

echo "Computing acc comparison"
winlosstie --input $Safe/myresults.$Log.csv --fields 18 --perform 13 --key 3 --95 --high > $Safe/crossvalaccmw.$Log.csv
echo "Computing pd comparison"
winlosstie --input $Safe/myresults.$Log.csv --fields 18 --perform 14 --key 3 --95 --high > $Safe/crossvalpdmw.$Log.csv
echo "Computing prec comparison"
winlosstie --input $Safe/myresults.$Log.csv --fields 18 --perform 16 --key 3 --95 --high > $Safe/crossvalprecmw.$Log.csv
echo "Computing bal comparison"
winlosstie --input $Safe/myresults.$Log.csv --fields 18 --perform 17 --key 3 --95 --high > $Safe/crossvalbalmw.$Log.csv
echo "Computing npf comparison"
winlosstie --input $Safe/myresults.$Log.csv --fields 18 --perform 18 --key 3 --95 --high > $Safe/crossvalnpfmw.$Log.csv

End=`date +%H:%M:%S%t%m/%d/%Y`
echo $Start "to" $End
echo $Start "to" $End >> myresults.$Log.csv
echo $Safe/xval.$Log
```

Notes on Support Scripts:

- *instances* counts the number of data instances that occur in the data file provided to it.
- *someArff* randomly generates the testing and training data sets from the file provided.
- *classes* returns the classes found in the provided data set.
- *abcd* computes the A, B, C, and D values for the provided data file and also calculates each instances Accuracy, PD, PF, Balance, and Precision score
- *doExtras* computes the !PF score for each instance of the results
- *winlosstie* computers the U-test tables for each performance measure provided.

WTTP Script

```
#!/bin/bash
t=$1;
T=$2;
shift 2
$wEka $* -p 0 -t $t -T $T
```

Appendix C

tenbins Source Code

```
#!/usr/bin/gawk -f
# /* vim: set filetype=awk : */ -*- awk -*-
# Original Script by Dr. Tim Menzies as nbins
# Modified for use as tenbins by DJ Boland, January 2007
### patterns0: initialization stuff
 BEGIN{
    BinLog=0; # if non-zero, use bin logging after rounding numbers
    Bins=10;
    FS=OFS=",";
    IGNORECASE=1;
    Inf=10**32;
 }
               {sub(/\%.*/,"")}
/^[ \t]*$/     {next}

/@relation/    { Header=1; Data=0; Attr=0; }
/@data/        { Header=0; Data=1; }
/@attribute/   { Attr++; }

Pass==1 && /@attribute/ && (/numeric/ || /real/ || /integer/) {
    Numeric[Attr]= 1;
    Max[Attr]     = -1*Inf;
    Min[Attr]     = Inf;
}
Pass==1 && Data && NF > 1{
    for(I in Numeric) {
       if ( $I ~  /?/ ) continue;
       if ($I > Max[I] ) Max[I]=$I;
       if ($I < Min[I] ) Min[I]=$I;
          Seen[I,$I]++;
         }
 }
Pass==2 && /@relation/ {
    for(I in Numeric) {
       Div[I]=Bins
       Bin[I]=(Max[I]-Min[I])/ Div[I];
       print "% attribute:" I "=-=min:"Min[I]  "=-=max:"Max[I] \
                               "=-=bins:"Div[I] "=-=steps:"Bin[I]}
}

Pass==2 && /@attribute/ {
          if  ( (Attr in Numeric) && Attr != MaxAttr ) {
                    Names="";
                    for(I=2;I<=Div[Attr];I++) Names=Names",_"I
                            sub(/integer|numeric|real/,"{_1"Names"}");
          }
}
Pass==2 && Data && NF> 1{
    for(I in Numeric) {
       if ( I == MaxAttr ) continue;
       if ( $I ~ /?/ )       continue;
       $I="_"label(I,$I)};
 }
Pass==2 {print $0}

function round(x) { return int(x + (x<0 ? -0.5 : 0.5)) }
function label(i,val,    x) {
    if ( Bin[i]==0 ) return 1;
    x=round((val-Min[i])/Bin[i]);
    if (x==0) {return 1} else {return x};
 }
```

Appendix D

Script for PKID

PKID Call BASH Script

Usage: pkid *datafile*

```
#!/bin/bash
myJava=${myJava="java -Xmx1024M -cp /home/donaldb/bin/weka.jar "}
myFilter=${myFilter="weka.filters.unsupervised.attribute.PKIDiscretize "}
inFile=${inFile="-i" $1}
myOpts=${myOpts="-R first-last -c last"}

command=${command= $myJava $myFilter $inFile}
$command > out.data

prunejava out.data > out2.data
cat out2.data
rm out.data
rm out2.data
```

Prunejava GAWK Script

```
#!/usr/bin/gawk -f
BEGIN{
  FS = " *, *";
}
  {sub(/\%.*/,"")}          #Substitute comments with blanks
  {gsub(/[\'\"\`]/,"",$0)} #Remove any single or double quotes from this line
  {gsub("'", "", $0)}       #Remove any single quotes

/^[ \t]*$/ {next}           #When blank line observed, go to the next line

/^java/ {next}

{print $0}
```

Appendix E

Entropy-Minimization Method Script

usage: fayyadIrani *datafile*

```
#!/bin/bash
java -cp $wJar weka.filters.supervised.attribute.Discretize -i $1 -c last
```

Where $wJar is an environmental variable set to the location of the weka.jar file and $1 represents the file presented to the script for discretization.

Appendix F

Performance Measure U-test Tables

F.1 Accuracy U-test By Data Set

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.1: audiology for acc

key	*win*	*loss*	*ties*	*win-loss*
disctree3	4	0	0	4
pkid	1	1	2	0
fayyadIrani	1	1	2	0
tbin	0	1	3	-1
cat	0	3	1	-3

Figure F.2: auto-mpg for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.3: breast-cancer for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.4: breast-cancer-wisconsin for acc

key	*win*	*loss*	*ties*	*win-loss*
disctree3	3	0	1	3
fayyadIrani	2	0	2	2
pkid	2	1	1	1
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.5: credit-a for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	2	1	1	1
cat	2	1	1	1
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.6: diabetes for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.7: ecoli for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	1	0	3	1
pkid	1	0	3	1
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
cat	0	4	0	-4

Figure F.8: flag for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
cat	2	0	2	2
disctree3	1	0	3	1
tbin	1	2	1	-1
pkid	0	4	0	-4

Figure F.9: hayes-roth for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	1	0	3	1
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0
pkid	0	1	3	-1

Figure F.10: heart-c for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.11: heart-h for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
tbin	1	0	3	1
disctree3	0	1	3	-1
cat	0	1	3	-1
pkid	0	2	2	-2

Figure F.12: hepatitis for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	1	1	2	0
disctree3	0	1	3	-1
cat	0	1	3	-1
pkid	0	2	2	-2

Figure F.13: imports-85 for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.14: iris for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.15: kr-vs-kp for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
pkid	2	0	2	2
disctree3	2	1	1	1
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.16: letter for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.17: mushroom for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
pkid	1	2	1	-1
disctree3	1	2	1	-1
cat	0	4	0	-4

Figure F.18: segment for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.19: soybean for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.20: splice for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
cat	2	2	0	0
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.21: vowel for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	2	1	1	1
cat	2	1	1	1
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.22: waveform-5000 for acc

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	1	1	2	0
pkid	1	1	2	0
disctree3	1	1	2	0
cat	0	4	0	-4

Figure F.23: wdbc for acc

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.24: wine for acc

F.2 Balance *U*-test by Data Set

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.25: audiology for bal

key	*win*	*loss*	*ties*	*win-loss*
disctree3	4	0	0	4
pkid	1	1	2	0
tbin	0	1	3	-1
fayyadIrani	0	1	3	-1
cat	0	2	2	-2

Figure F.26: auto-mpg for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.27: breast-cancer for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.28: breast-cancer-wisconsin for bal

key	*win*	*loss*	*ties*	*win-loss*
disctree3	3	0	1	3
fayyadIrani	2	0	2	2
pkid	2	1	1	1
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.29: credit-a for bal

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
disctree3	1	1	2	0
tbin	0	1	3	-1
pkid	0	1	3	-1
cat	0	2	2	-2

Figure F.30: diabetes for bal

key	*win*	*loss*	*ties*	*win-loss*
pkid	2	0	2	2
fayyadIrani	2	0	2	2
cat	2	0	2	2
tbin	0	3	1	-3
disctree3	0	3	1	-3

Figure F.31: ecoli for bal

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
pkid	1	0	3	1
disctree3	1	0	3	1
cat	1	1	2	0
tbin	0	4	0	-4

Figure F.32: flag for bal

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
cat	2	0	2	2
tbin	1	3	0	-2
pkid	0	4	0	-4

Figure F.33: hayes-roth for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	1	0	3	1
disctree3	1	0	3	1
fayyadIrani	0	0	4	0
cat	0	0	4	0
pkid	0	2	2	-2

Figure F.34: heart-c for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.35: heart-h for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	2	0	2	2
fayyadIrani	2	0	2	2
disctree3	1	0	3	1
pkid	0	2	2	-2
cat	0	3	1	-3

Figure F.36: hepatitis for bal

key	*win*	*loss*	*ties*	*win-loss*
pkid	1	0	3	1
fayyadIrani	1	0	3	1
tbin	0	0	4	0
disctree3	0	0	4	0
cat	0	2	2	-2

Figure F.37: imports-85 for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.38: iris for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.39: kr-vs-kp for bal

key	*win*	*loss*	*ties*	*win-loss*
pkid	2	0	2	2
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.40: letter for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.41: mushroom for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
pkid	1	2	1	-1
disctree3	1	2	1	-1
cat	0	4	0	-4

Figure F.42: segment for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.43: soybean for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.44: splice for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
cat	2	2	0	0
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.45: vowel for bal

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
tbin	1	0	3	1
cat	1	1	2	0
pkid	0	1	3	-1
disctree3	0	3	1	-3

Figure F.46: waveform-5000 for bal

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	1	1	2	0
pkid	1	1	2	0
disctree3	1	1	2	0
cat	0	4	0	-4

Figure F.47: wdbc for bal

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.48: wine for bal

F.3 Precision U-test by Data Set

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.49: audiology for prec

key	*win*	*loss*	*ties*	*win-loss*
disctree3	3	0	1	3
pkid	0	0	4	0
tbin	0	1	3	-1
fayyadIrani	0	1	3	-1
cat	0	1	3	-1

Figure F.50: auto-mpg for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.51: breast-cancer for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.52: breast-cancer-wisconsin for prec

key	*win*	*loss*	*ties*	*win-loss*
pkid	2	0	2	2
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.53: credit-a for prec

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	0	1	3	-1
pkid	0	1	3	-1
disctree3	0	1	3	-1
cat	0	1	3	-1

Figure F.54: diabetes for prec

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
pkid	1	0	3	1
cat	1	0	3	1
tbin	0	1	3	-1
disctree3	0	3	1	-3

Figure F.55: ecoli for prec

key	*win*	*loss*	*ties*	*win-loss*
pkid	1	0	3	1
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
cat	1	0	3	1
tbin	0	4	0	-4

Figure F.56: flag for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	1	0	3	1
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
cat	1	0	3	1
pkid	0	4	0	-4

Figure F.57: hayes-roth for prec

key	*win*	*loss*	*ties*	*win-loss*
disctree3	1	0	3	1
tbin	0	0	4	0
fayyadIrani	0	0	4	0
cat	0	0	4	0
pkid	0	1	3	-1

Figure F.58: heart-c for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.59: heart-h for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.60: hepatitis for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.61: imports-85 for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.62: iris for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.63: kr-vs-kp for prec

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
pkid	2	0	2	2
disctree3	2	1	1	1
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.64: letter for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.65: mushroom for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
pkid	1	2	1	-1
disctree3	0	2	2	-2
cat	0	3	1	-3

Figure F.66: segment for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.67: soybean for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.68: splice for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	2	0	2	2
fayyadIrani	2	0	2	2
cat	2	0	2	2
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.69: vowel for prec

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
tbin	1	0	3	1
cat	1	0	3	1
pkid	0	1	3	-1
disctree3	0	3	1	-3

Figure F.70: waveform-5000 for prec

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
pkid	1	1	2	0
disctree3	1	1	2	0
tbin	0	1	3	-1
cat	0	3	1	-3

Figure F.71: wdbc for prec

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.72: wine for prec

F.4 Probability of Detection U-test by Data Set

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.73: audiology for pd

key	*win*	*loss*	*ties*	*win-loss*
disctree3	4	0	0	4
pkid	1	1	2	0
tbin	0	1	3	-1
fayyadIrani	0	1	3	-1
cat	0	2	2	-2

Figure F.74: auto-mpg for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.75: breast-cancer for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.76: breast-cancer-wisconsin for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
pkid	1	0	3	1
tbin	1	2	1	-1
cat	0	4	0	-4

Figure F.77: credit-a for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	0	1	3	-1
pkid	0	1	3	-1
disctree3	0	1	3	-1
cat	0	1	3	-1

Figure F.78: diabetes for pd

key	*win*	*loss*	*ties*	*win-loss*
cat	4	0	0	4
pkid	2	1	1	1
fayyadIrani	2	1	1	1
tbin	0	3	1	-3
disctree3	0	3	1	-3

Figure F.79: ecoli for pd

key	*win*	*loss*	*ties*	*win-loss*
pkid	1	0	3	1
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
cat	1	0	3	1
tbin	0	4	0	-4

Figure F.80: flag for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
cat	2	0	2	2
disctree3	1	0	3	1
tbin	1	2	1	-1
pkid	0	4	0	-4

Figure F.81: hayes-roth for pd

key	*win*	*loss*	*ties*	*win-loss*
disctree3	1	0	3	1
tbin	0	0	4	0
fayyadIrani	0	0	4	0
cat	0	0	4	0
pkid	0	1	3	-1

Figure F.82: heart-c for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.83: heart-h for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	1	0	3	1
tbin	0	0	4	0
pkid	0	0	4	0
disctree3	0	0	4	0
cat	0	1	3	-1

Figure F.84: hepatitis for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.85: imports-85 for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.86: iris for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.87: kr-vs-kp for pd

key	*win*	*loss*	*ties*	*win-loss*
pkid	2	0	2	2
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.88: letter for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.89: mushroom for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
tbin	2	0	2	2
pkid	1	1	2	0
disctree3	1	2	1	-1
cat	0	4	0	-4

Figure F.90: segment for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.91: soybean for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.92: splice for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	3	0	1	3
fayyadIrani	3	0	1	3
cat	2	2	0	0
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.93: vowel for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
cat	2	1	1	1
tbin	1	1	2	0
pkid	1	2	1	-1
disctree3	0	4	0	-4

Figure F.94: waveform-5000 for pd

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
disctree3	1	0	3	1
tbin	1	1	2	0
pkid	1	1	2	0
cat	0	4	0	-4

Figure F.95: wdbc for pd

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.96: wine for pd

F.5 Probability of Not False Alarm U-test by Data Set

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.97: audiology for npf

key	*win*	*loss*	*ties*	*win-loss*
disctree3	3	0	1	3
pkid	0	0	4	0
tbin	0	1	3	-1
fayyadIrani	0	1	3	-1
cat	0	1	3	-1

Figure F.98: auto-mpg for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.99: breast-cancer for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.100: breast-cancer-wisconsin for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
pkid	1	0	3	1
tbin	1	2	1	-1
cat	0	4	0	-4

Figure F.101: credit-a for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	4	0	0	4
tbin	0	1	3	-1
pkid	0	1	3	-1
disctree3	0	1	3	-1
cat	0	1	3	-1

Figure F.102: diabetes for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.103: ecoli for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	1	0	3	1
pkid	1	0	3	1
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
cat	0	4	0	-4

Figure F.104: flag for npf

key	*win*	*loss*	*ties*	*win-loss*
cat	2	0	2	2
fayyadIrani	1	0	3	1
disctree3	1	0	3	1
tbin	1	1	2	0
pkid	0	4	0	-4

Figure F.105: hayes-roth for npf

key	*win*	*loss*	*ties*	*win-loss*
disctree3	1	0	3	1
tbin	0	0	4	0
fayyadIrani	0	0	4	0
cat	0	0	4	0
pkid	0	1	3	-1

Figure F.106: heart-c for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.107: heart-h for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	1	0	3	1
tbin	0	0	4	0
pkid	0	0	4	0
disctree3	0	0	4	0
cat	0	1	3	-1

Figure F.108: hepatitis for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
tbin	1	0	3	1
disctree3	0	1	3	-1
cat	0	1	3	-1
pkid	0	2	2	-2

Figure F.109: imports-85 for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.110: iris for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.111: kr-vs-kp for npf

key	*win*	*loss*	*ties*	*win-loss*
pkid	2	0	2	2
fayyadIrani	2	0	2	2
disctree3	2	0	2	2
tbin	1	3	0	-2
cat	0	4	0	-4

Figure F.112: letter for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.113: mushroom for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	2	0	2	2
fayyadIrani	2	0	2	2
cat	0	0	4	0
pkid	0	2	2	-2
disctree3	0	2	2	-2

Figure F.114: segment for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.115: soybean for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.116: splice for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	2	0	2	2
fayyadIrani	2	0	2	2
cat	2	0	2	2
pkid	0	3	1	-3
disctree3	0	3	1	-3

Figure F.117: vowel for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	2	0	2	2
tbin	1	0	3	1
cat	1	0	3	1
pkid	0	1	3	-1
disctree3	0	3	1	-3

Figure F.118: waveform-5000 for npf

key	*win*	*loss*	*ties*	*win-loss*
fayyadIrani	3	0	1	3
disctree3	1	0	3	1
tbin	1	1	2	0
pkid	1	1	2	0
cat	0	4	0	-4

Figure F.119: wdbc for npf

key	*win*	*loss*	*ties*	*win-loss*
tbin	0	0	4	0
pkid	0	0	4	0
fayyadIrani	0	0	4	0
disctree3	0	0	4	0
cat	0	0	4	0

Figure F.120: wine for npf

Bibliography

[1] D.J. Newman A. Asuncion. UCI machine learning repository, 2007.

[2] B. Kitchenham, L. Pickard, S. MacDonell, and M. Shepperd. What accuracy statistics really measure. *Software,IEE Proceedings*, 148(3):81–85, 2001.

[3] J. Cendrowska. Prism: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27:349–370, 1987.

[4] CN. Hsu, HJ. Huang, and TT. Wong. Implications of the Dirichlet Assumption for Discretization of Continuous Variables in Naive Bayesian Classifiers. *Machine Learning*, 53(3):235–263, 2003. Available online at: http://www.iis.sinica.edu.tw/ chunnan/DOWNLOADS/MACH-1735-00.pdf.

[5] William W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[6] D. S. Moore and G. P. McCabe. *Introduction to the Practice of Statistics*. W. H. Freeman and Company, New York, 2nd edition, 1993.

[7] J. Demsar. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, 7:1–30, 2006. available from http://jmlr.csail.mit.edu/papers/v7/demsar06a.html.

[8] T. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.

[9] P. Domingos and M. Pazzani. On the Optimality of Simple Bayesian Classifier under Zero-One Loss. *Machine Learning*, 29:103–130, 1997.

[10] U. M. Fayyad and K. B. Irani. On the Handling of Continuous-Valued Attributes in Decision Tree Generation. *Machine Learning*, 8:87–102, 1992.

[11] G.H. John and P. Langley. Estimating Continuous Distributions in Bayesian Classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*. Morgan Kaufmann, 1995.

[12] H. Liu, F. Hussain, C. L. Tan, and M. Dash. Discretization: An Enabling Technique. *Data Mining and Knowledge Discovery*, 6(4):393–423, October 2002.

[13] R. C. Holte. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning*, 11(1):63–90, April 1993.

[14] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and Unsupervised Discretization of Continuous Features. In *Machine Learning, Proceedings of the Twelfth International Conference*, pages 194–202, 1995.

[15] H.B. Mann and D. R. Whitney. On A Test Of Whether One Of Two Random Variables Is Stochastically Larger Than The Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[16] C. Martinez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, March 1998.

[17] A. S. Orrego. Sawtooth: Learning from huge amounts of data. Master's thesis, West Virginia University, 2004.

[18] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[19] Robert Sedgewick. *Algorithms in Java: Parts 1-4*. Addison-Wesley Professional, 3rd edition, 2002.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, New York, 2nd edition, 2001.

[21] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:80–83, 1945.

[22] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufman, 2 edition, 2005.

[23] Y. Yang and G. I. Webb. Weighted Proportional k-Interval Discretization for Naive-Bayes Classifiers. In *Proceedings of PAKDD 2003: The 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2003. Available from http://www.csse.monash.edu/ webb/Files/YangWebb03.pdf.

[24] Y. Yang and G.I.Webb. A comparative study of discretization methods for naive-bayes classifiers. In *Proceedings of PKAW 2002, The 2002 Pacific Rim Knowledge Acquisition Workshop*, pages 159–173, Tokyo, 2002.

[25] Y. Yang and G. I. Webb. Proportional k-interval discretization for naive-bayes classifiers. In *Proceedings of the 12th European Conference on Machine Learning*, pages 564–575. Springer Berlin, 2001.

[26] Y. Yang and G. I. Webb. Non-disjoint discretization for naive-bayes classifiers. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 666–673, 2002.

[27] Y. Yang and G. I. Webb. Discretization for naive-bayes learning: managing discretization bias and variance. Technical report, School of Computer Science and Software Engineering, Monash University, 2003.

[28] Y.Yang and G.I. Webb. On Why Discretization Works for Naive-Bayes Classifiers. In *AI 2003: Advances in Artificial Intelligence*, pages 440–452. Springer Berlin, 2003.