Evaluating Biological Data Using Rank Correlation Methods

Douglas J. Slotta

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Lenwood S. Heath, Chairman
Richard F. Helm
T. M. Murali
Malcolm Potts
Narendran Ramakrishnan
John Paul C. Vergara

May 5, 2005
Blacksburg, Virginia

Keywords: feature selection, rank-order, spoiler count, microarrays, MPSS, bioinformatics

Evaluating Biological Data Using Rank Correlation Methods

Douglas J. Slotta

(ABSTRACT)

Analyses based upon rank correlation methods, such as Spearman's ρ and Kendall's τ, can provide quick insights into large biological data sets. Comparing expression levels between different technologies and models is problematic due to the different units of measure. Here again, rank correlation provides an effective means of comparison between the two techniques. Massively Parallel Signature Sequencing (MPSS) transcript abundance levels to microarray signal intensities for *Arabidopsis thaliana* are compared. Rank correlations can be applied to subsets as well as the entire set. Results of subset comparisons can be used to improve the capabilities of predictive models, such as Predicted Highly Expressed (PHX). This is done for *Escherichia coli*. Methods are given to combine predictive models based upon feedback from experimental data. The problem of feature selection in supervised learning situations is also considered, where all features are drawn from a common domain and are best interpreted via ordinal comparisons with other features, rather than as numerical values. This is done for synthetic data as well as for microarray experiments examining the life cycle of *Drosophila melanogaster* and human leukemia cells. Two novel methods are presented based upon ρ and τ, and their efficacy is tested with synthetic and real world data. The method based upon Spearman's ρ is shown to be more effective.

Contents

List of Figures

Acknowledgments

First, I am indebted to the advice and assistance of my advisor, Lenwood Heath, who has taught me about more than computer science. I would also like to thank my committee for their aid, especially Narendran Ramakrishnan and John Paul C. Vergara for their insights and critique into the machine learning aspects of this work. It is unfortunate that they could not make it to the final defense, but it is a good thing that Liqing Zhang was heroic enough to take the place of both of them for that task. Richard Helm and Malcolm Potts provided the biological foundation that I so desperately needed, and T. M. Murali who was the jack-of-all-trades that never let me get away with any hand waving.

Finally, all of the blame (and the glory) for my pursuit of a PhD. belongs solely to my wife, Tracey, who had the grace to show me how it was done first.

Chapter 1

Introduction

Many objects in this universe are amenable to being ranked according to some measure. The planets can be ranked according to size or distance from the sun; colleges can be ranked by their football teams. For planets, size is an objective measure that everyone will agree upon. Ranking college football teams is a subjective measure hotly debated in bars across the country. There are three canonical sources of ranks for college football teams: the AP coaches poll, the USA Today coaches poll, and the Bowl Championship Series. All three of these rank the same objects (college football teams), though typically in different orders. The rankings are the results of different methods of comparing teams, but the rankings themselves can also be compared.

A number, such as 5 or 3.2, indicates quantity but not order, unless it is compared with another number. An ordinal number indicates order or succession, such as first, second, third, or 513th. When objects are arranged according to some quality possessed to a varying degree they are said to be ranked with respect to that number. The arrangement as a whole is called a ranking. The rank (position) of each object indicates its respective position in the ranking.

Many features of biological objects lead to a natural ranking of those objects. For example, people can be ranked according to their height, weight, or IQ, but not by their hair color or gender (at least not naturally). Genes can be ranked by the expression level of their mRNA under some experimental condition. For example, an *E. coli* colony cultured at 13.5°C , relative to *E. coli* at 37°, the gene *ackA* has expression level 4.5, the gene *gltA* has expression level 2.6, and gene *atpD* has expression level 3.2. However, when *E. coli* is at 42°C, relative to *E. coli* at 37°, the

expression of *ackA* is 5.2, that of *gltA* is 1.7, and that of *atpD* is 3.4 [50]. The objects are *E. coli* at 13.5°C and 42°C. The features are the expression levels of the genes *ackA*, *gltA*, and *atpD*. In this instance, the values of the features are different, but the ranks of the features are exactly the same; in both experiments, *ackA* has more expression than *atpD*, which has more expression than *gltA*. Two popular methods of measuring a difference between two rankings of the same objects are Kendall's tau and Spearman's rho. Both assign a coefficient of concordance (a number) that quantifies how alike two rankings are. Section 2 describes these two coefficients in detail.

Kim et al. [27] used Spearman's rank correlation methods solely to examine microarray reproducibility. Spearman's Footrule is used as a basis for their Index.R, which is a coarse measure of how close two replications are in terms of their reproducibile features. Spearman's Footrule, however, has less discriminatory power [26, page 32] than other rank correlation methods, such as Kendall's τ or Spearman's ρ. Therefore, it is not discerning enough to note subtle distinctions between rank orders that could have important biological significance.

A number of important economic, financial, and scientific domains present situations where features are best interpreted via ordinal comparisons with other features, rather than as absolute values. This is especially the case when the features represent counts, ratings, rankings, or otherwise dimensionless quantities. For instance, a financial analyst may characterize a trading day in terms of several features, each an ordinal relationship among market indices (e.g., HANG SENG > NIKKEI > S&P > DOW) according to criteria such as percentage change or trading volume.

For instance, consider a financial analyst interested in assessing the outlook of the stock market ("bear" or "bull") by analyzing the changes during one trading day. Instances are hence trading days and the (binary) outlook forecasts constitute the classes. Each instance is described by, say, the average percentage change experienced by stocks in various market indices, e.g., HANG SENG, NIKKEI, S&P, and DOW. For instance, a given trading day would be qualified using the features HANG SENG $\uparrow$ 2.5%, NIKKEI $\uparrow$ 4.8%, S&P $\downarrow$ 2.8%, DOW $\uparrow$ 1.3%. However, it is not meaningful to work directly with the percentage changes in an attribute-value sense; instead we rank the features and, henceforth, think of a trading day in terms of an order (total or partial) over the features. The above example can be represented as: NIKKEI > HANG SENG > DOW > S&P. The goal of supervised learning in this setting is to infer a mapping from such orders to given classifications. For instance, a possible implication is 'if DOW is ranked higher than S&P but ranked lower than NIKKEI, then the outlook is bullish'. Feature selection in such rank-order datasets is important for the same reasons it is in regular feature spaces,

namely, reducing the complexity of induction, removing irrelevant information from a dataset, and improving prediction performance. Perhaps the relative ordering of the HANG SENG index among the other indices is not informative toward the goal of outlook prediction, and hence the feature can be safely eliminated.

There are many applications that highlight the importance of feature selection in rank-order datasets. In biomedical instrumentation, the desire is to select a subset of electrodes from an EEG dataset and use profiles of relative signal strength as indicators of patient health [56]. Here the instances are the patients, the classes are the diagnoses, and the features denote signal strength as measured using different electrodes. In large-scale gene expression assays [1, 49], one possible aim is to classify an experimental condition using expression changes only across a 'salient' subset of genes. For instance, by observing a handful of genes (features) and ranking them by their expression levels, it is possible to qualitatively characterize the cellular transcriptional state (class) for a given condition (instance). In decision-making referendums, one goal is to identify key voting indicators to infer political biases of constituencies. Here, the instances are the constituencies, the classes denote political party strongholds, and the features could be socio-economic indicators. In a lighter vein, it appears possible to classify a movie as an art film or a mass market flick by ranking critics! Given such a widespread prevalence of applications where rank order is pertinent [36], it is surprising that this feature selection problem has received little attention.

The formulation is made precise in Section 5.1, but, informally, we posit a setting where the features can be ordered within an instance (and thus, must come from the same domain). This differs starkly from the more traditional settings where the instances or classes are ordered or where the feature values are ranked *across instances*. These other formulations are pertinent in applications such as recommender systems or information retrieval [11, 22], where the goal is to learn and mine a ranking or to infer total orders from given preference information. Here, one assumes the existence of a supervised learning algorithm that learns predictive mappings from orders to classes and the focus is on feature selection as a preprocessing step to such an algorithm.

The specific focus is on identifying a subset of ordinal features (i. e., projections of the given set of features) that exhibit sufficient relationships to model feature-conditional class distributions for use in supervised learning. The result is a 'best subset' of ordinal features, not a 'best ordering' of the features; aggregating orderings by consolidating given ranking information is the purview of subjects such as social choice theory [46] (an area rich in impossibility results [2]).

The feature selection problem considered here is unique in its formulation although it has motivations similar to others found in existing literature on feature selection. The importance of removing redundant as well as irrelevant features was recognized early in machine learning research and several theoretical frameworks have been put forth [31, 34, 4]. The work of Koller and Sahami [29], as does our research, follows the preprocessing paradigm where feature selection is considered as preliminary to induction and hence learning is agnostic to a specific algorithm. In contrast, the wrapper-based approach studies feature selection in the context of a specific learning algorithm (e.g., kernel machines [20]).

Order theoretic considerations have been introduced in different guises in machine learning research. Cohen et al. [11] describe how to induce an (approximate) global order from given (partial) rankings or preference information. Mannila and Meek [35] view partial orders as generative models of sequential trace data. Kamishima and Akaho [23] extend the work of Cohen et al. by accommodating problems where inputs are themselves orders. A compelling application context for these ideas is described in Kamishima's Nantonac collaborative filtering [22]. More recently, Lebanon and Lafferty [32] present a boosting-like algorithm to combine multiple rankings by reasoning about probability distributions over perturbations. This work is extended by Lebanon and Lafferty [33] into a unifying framework for classification and ranking. These papers do not directly either address the problem of feature selection or even a supervised learning scenario from orders to discrete classes. Gionis et al. [18] study 'fragments of order', although their setting posits unlabeled data and the learned orders are actually sets of association rules summarized as dependencies. Sai et al. [45] describe a scenario closer to our work where dataset instances are viewed only through ordinal comparisons across features, but their goal is to learn association rules between such ordinal comparisons, not to predict a class or feature selection. The work presented here is different from all the above works in the viewpoint of the dataset (labeled), the input instance (a total order over features), or the desired output (a reduced set of features suitable for defining relationships from orders to classes). Slotta et al. [47] originally developed this concept for an application in proteomics.

This document is organized as follows. First, in Chapter 2 there is a discussion of the rank correlation methods that form the basis of the rest of the work. Then these correlation methods are applied to several biological data and models in Chapter 3. Afterwards, rank correlation methods are applied to the feature selection problem in Chapter 5. Finally, Chapter 6 discusses some conclusions and possible future work, followed by the appendices containing the implementation of the algorithms using both C and Python.

Chapter 2

Rank Correlation Methods

Let L_n denote the set $\{1, 2, \ldots, n\}$, of integers between 1 and n. A permutation of L_n is a bijective function $\pi : L_n \to L_n$. The *symmetric group* S_n is the set of all permutations of L_n. A permutation $\pi \in S_n$ is an ordered arrangement of the elements in L_n where $\pi(i)$ is the element in position i and $\pi(i, j)$ is the subsequence of elements at positions i through j. There are two representations of permutations: an *order vector*, which is an ordered list of the elements, and a *rank vector*, which is a list of the ranks of the elements. The rank of an element is its position in the order vector. In this document, the integers are used to indicate both position and elements. Therefore, an *italic* font shall be used to indicate position or rank, and a **boldface** font shall be used to indicate elements. The order vector is a sequence of objects, written in boldface and delimited by spaces. Note that the rank vector presumes a canonical ordering of the items in the set L_n. In this paper, the canonical order will always be standard integer order.

For example, if $n = 5$, so $L_n = \{1, 2, 3, 4, 5\}$, then one permutation of L_n in order vector representation is $\pi =$ **3 2 4 1 5**; the corresponding rank vector is $\pi =$ *4 2 1 3 5* because 1 is ranked 4th, 2 is ranked second, 3 is ranked first, 4 is ranked third, and 5 is ranked fifth. Care should be taken so as not to confuse the two representations.

Order statistics are used to determine the distance between two permutations. By convention, if a distance is given for only a single permutation, then it is assumed to be compared to the canonical ordering, i.e. $\pi =$ **1 2 3 . . . n-1 n**, or $\pi =$ *1 2 3 . . . n-1 n*.

Informally, Marden [36] states that there are two different types of distances between

$$
\begin{array}{rll}
\pi_j = & \mathbf{3\ 2\ 4\ 5\ 1\ 6} & \\
 & \mathbf{3\ 2\ 5\ 4\ 1\ 6} & \text{swap(4,5)} \\
 & \mathbf{3\ 5\ 2\ 4\ 1\ 6} & \text{swap(2,5)} \\
 & \mathbf{3\ 5\ 2\ 1\ 4\ 6} & \text{swap(4,1)} \\
\pi_i = & \mathbf{3\ 5\ 2\ 1\ 4\ 6} &
\end{array}
$$

Figure 2.1: Three interchanges are required to transform π_j to π_i

linear orders: spatial and disorder. Spatial distance is a measure of travel time between permutation to another in n-dimensional space. Disorder distance measures the effort required to reorder one permutation into another. Distances are commonly normalized into coefficients of correlation to facilitate comparisons between distances derived from symmetric groups of different n. There are many methods of determining correlation between two permutations but the two coefficients most commonly used are Kendall's tau and Spearman's rho.

2.1 Kendall's Tau

The Kendall distance (d_K), which is computed from the order vector, is both a disorder and spatial distance metric. As a disorder metric, it is the minimum number of swaps of adjacent pairs required to transform $\pi_i \in S_n$ into $\pi_j \in S_n$. Figure 2.1 provides an example transformation of $\pi_j = \mathbf{3\ 2\ 4\ 5\ 1\ 6}$ to $\pi_i = \mathbf{3\ 5\ 2\ 1\ 4\ 6}$ in 3 swaps, showing the three intermediate permutations. This is exactly the same as the inversion count of Knuth [28]. If $\pi_i(x) < \pi_i(y)$ and $\pi_j(x) > \pi_j(y)$ then the pair (x, y) is an inversion. The total number of inversions between two lists provides a measure of how sorted one list is with respect to another. A permutation π_j with no inversions with respect to π_i has the same order as π_i. A permutation with only one inversion has an adjacent pair of items out of order. For any permutation π, $d_K(\pi, \pi^{-1}) = \binom{n}{2}$, the greatest possible Kendall distance between two elements of S_n.

The probability of a random permutation having inversion count x can be determined by knowing the distribution of inversion counts for all possible permutations. The total number of permutations of a list is $n!$, and the minimum inversion count is 0, while the maximum inversion count is $n(n-1)/2$. Knuth [28] (page 16) shows that the distribution of counts is approximately normal with mean $\mu = n(n-1)/4$ and standard deviation $\sigma = \sqrt{n(2n+5)(n-1)/72}$. These values are summarized

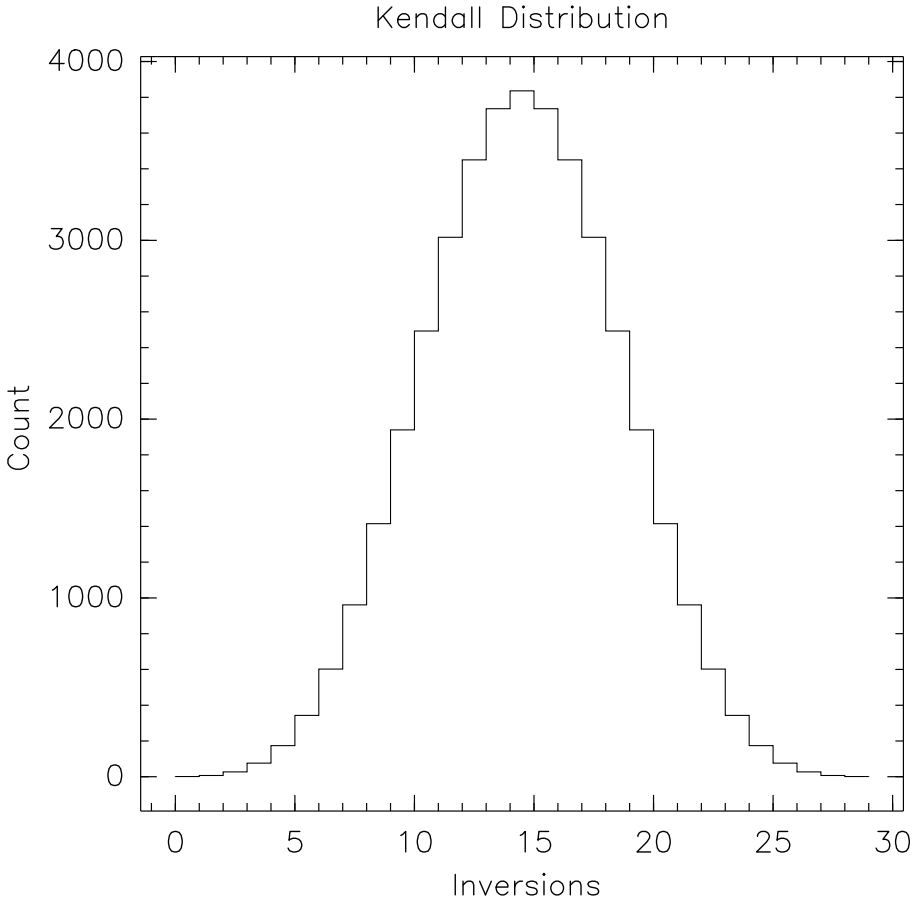


Figure 2.2: Distribution of inversion counts for a list of length 8.

in Figure 2.3.

Since the distribution of scores is approximately normal, there are standard statistical evaluations of that can be performed. The number of standard deviations that a permutation with inversion count x is away from the average is its z-score: $z = |\frac{\mu - x}{\sigma}|$. The greater the magnitude of the z score of an order, the less likely it is that an order with that inversion count occurs at random.

Figure 2.2 shows the distribution of inversion counts for lists of length 8. This was created by generating all permutations in S_8 and computing $d_K(\pi_i, \pi)$ for each π where $\pi_i = \mathbf{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}$. The choice of π_i is arbitrary since the results are symmetric with respect to the distribution. Out of 40,320 permutations of length 8, there are 3,836 with 14 inversions, while only 7 have a single inversion. If a

permutation has an inversion count of 3, then 111 permutations have an equal or lesser score, which is only 0.275%. The z-score for a list of length 8 with 3 inversions is

$$\frac{\sqrt{\frac{8(2(8)+5)(8-1)}{72}} - 3}{\frac{8(8-1)}{4}} \approx 0.0743.$$

To facilitate the comparison of distances when the metrics are derived from orders of differing length, the spread of the distance metric is normalized to be a coefficient between -1 and 1. A result of 1 occurs when two permutations are identical and -1 indicates that the two permutations are in reverse order with respect to each other. To do this, take $d(\pi_i, \pi_j)/Max(d)$ where $\pi_a, \pi_b \in S_n$ and $Max(d) = Max\{d(\pi_a, \pi_b) | \pi_a, \pi_b \in S_n\}$ is the maximum possible distance between any two permutations of the same length as π_i and π_j, or $n(n-1)/2$. This normalizes d_k to be a number between 0 and 1. Kendall's tau maps this normalization of the inversion count d_K to the range -1 through 1. Therefore

$$\tau = 1 - \frac{4d_K}{n(n-1)}.$$

With a bit of algebraic manipulation it can be shown that the formula to convert from the z-score to τ is,

$$\tau = \frac{z \cdot \sqrt{2}(2n+5)}{3\sqrt{n(n-1)(2n+5)}},$$

and to convert from τ to the z-score is

$$z = \frac{3n\tau(n-1)}{\sqrt{2n(n-1)(2n+5)}}.$$

The inverse of a permutation is found by interchanging the rank representation and order representation of a permutation. For example, given $\pi =$ **4 2 1 3 5**, the rank representation is $\pi =$ *3 2 5 1 4*. Therefore the inverse of π is $\pi^{-1} =$ **3 2 5 1 4**. Another way to state this is that for all i, $\pi^{-1}(\pi(i)) = \pi(i)$. This relationship was defined by Rothe [44] who also proved that the inverse of a permutation has the same inversion count as that permutation.

In the same manner that the inverse of a permutation, π, has the same inversion count as the original permutation, so does the reverse of a permutation, π^R. The

reverse of a permutation is a complete transposition of the order, such that $\pi^R(1) = \pi(n)$, $\pi^R(2) = \pi(n-1)$, and so forth. The reverse of $\pi =$ **4 2 1 3 5** is $\pi^R =$ **5 3 1 2 4**. The reverse function only operates on the order vector and not the rank. The reverse of $\pi =$ *3 2 5 1 4* is $\pi =$ *3 4 2 5 1* . Note that both the inverse function and reverse function form bijections over S_n.

It is important to handle counting inversions with tied rankings correctly, otherwise their inverse and reverse will not have the same inversion counts. This means that the distance function will be asymmetric and therefore no longer a metric. Consider the following two orders, $\pi_i =$ **3 6** $\begin{smallmatrix}\mathbf{1}\\\mathbf{7}\end{smallmatrix}$ $\begin{smallmatrix}\mathbf{2}\\\mathbf{4}\end{smallmatrix}$ **5 8** and $\pi_j =$ **3 2 6** $\begin{smallmatrix}\mathbf{5}\\\mathbf{7}\end{smallmatrix}$ **8 1 4**, where elements in the same column denote tied rankings. Ignoring ties, $d_K(\pi_i, \pi_j) = 7$ and $\tau(\pi_i, \pi_j) = 0.5$. If one of the orders is reversed such that $\pi_i^R =$ **8 5** $\begin{smallmatrix}\mathbf{2}\\\mathbf{4}\end{smallmatrix}$ $\begin{smallmatrix}\mathbf{1}\\\mathbf{7}\end{smallmatrix}$ **6 3**, then $d_K(\pi_i, \pi_j) = 18$ and $\tau(\pi_i, \pi_j) \approx -0.286$. The difference between the absolute magnitudes of the preceding τ scores is large considering that only 3 of the 28 paired comparisons are ties.

Kendall [26] (pages 40-43) explains how τ can be a measure of agreement between two orders with ties. The agreement score, S, is the difference between the number of pairs in the same order, P, and the number of pairs with an inverse order, Q, therefore $S = P - Q$. This is normalized by dividing by the total number of paired comparisons, $\frac{1}{2}n(n-1)$, so $\tau = \frac{2S}{n(n-1)}$. In the previous example, for π_i and π_j, $S = 18 - 7 = 11$, therefore $\tau(\pi_i, \pi_j) \approx 0.39$. For π_i^R and π_j, $S = 7 - 18 = -11$, therefore $\tau(\pi_i^R, \pi_j) \approx -0.39$. The function is again symmetric.

Using $\frac{1}{2}n(n-1)$ to normalize the score implies that there is a universal objective order where no ties exist. This is not the case for gene expression levels where ties frequently occur. Kendall [26] (page 41) shows that ties should be taken into account in the denominator as well. If U is defined as the number of tied pairs in π_i and V is the number of tied pairs in π_j then

$$\tau(\pi_i, \pi_j) = \frac{S}{\sqrt{\frac{1}{2}n(n-1) - V}\sqrt{\frac{1}{2}n(n-1) - U}}.$$

Applying this new formula to the previous example, we obtain $\tau(\pi_i, \pi_j) \approx 0.42$ and $\tau(\pi_i^{-1}, \pi_j) \approx -0.42$.

Metric	Function	Min	Max	Mean	Std. Deviation	Normalization
Kendall	d_K	0	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{4}$	$\sqrt{\frac{n(n-1)(2n+5)}{72}}$	$\tau = 1 - \frac{4d_K}{n(n-1)}$
Spearman	${d_S}^2$	0	$\frac{(n^3-n)}{3}$	$\frac{n(n^2-1)}{6}$	$\sqrt{\frac{n^2(n-1)(n+1)^2}{36}}$	$\rho = 1 - \frac{6{d_S}^2}{(n^3-n)}$

Figure 2.3: Summary of equations for ρ and τ

2.2 Spearman's Rho

The Spearman distance (d_S) is computed from the rank vector. Ranks are assigned from 1 through n. If two or more features have the same rank then they are assigned the average of the ranks they would possess if they were not tied. If the 3rd through the 6th features are tied, then they are all assigned the rank $4\frac{1}{2}$. This midpoint method allows the sum of ranks for a given n to be the same regardless of ties.

The Spearman distance is the Euclidean distance between two n-dimensional vectors:

$$d_S(\pi_i, \pi_j) = \sqrt{\sum_{x=1}^{n} \Big(\pi_i(x) - \pi_j(x)\Big)^2}.$$

The distance d_S is a metric; however its normalization ρ is based upon the square of d_S, which is not a metric. The minimum distance is 0, while the square of the maximum distance is $(n^3 - n)/3$. The distribution of ${d_S}^2$ is also approximately normal, as illustrated in Figure 2.4, with mean $\mu = n(n^2 - 1)/6$ and standard deviation $\sigma = \sqrt{n^2(n-1)(n+1)^2/36}$. These values are summarized in Figure 2.3.

The normalization ρ is ${d_S}^2$ divided by the maximum possible distance and then normalized to be between 1 and -1. Therefore

$$\rho = 1 - \frac{6{d_S}^2}{(n^3 - n)}.$$

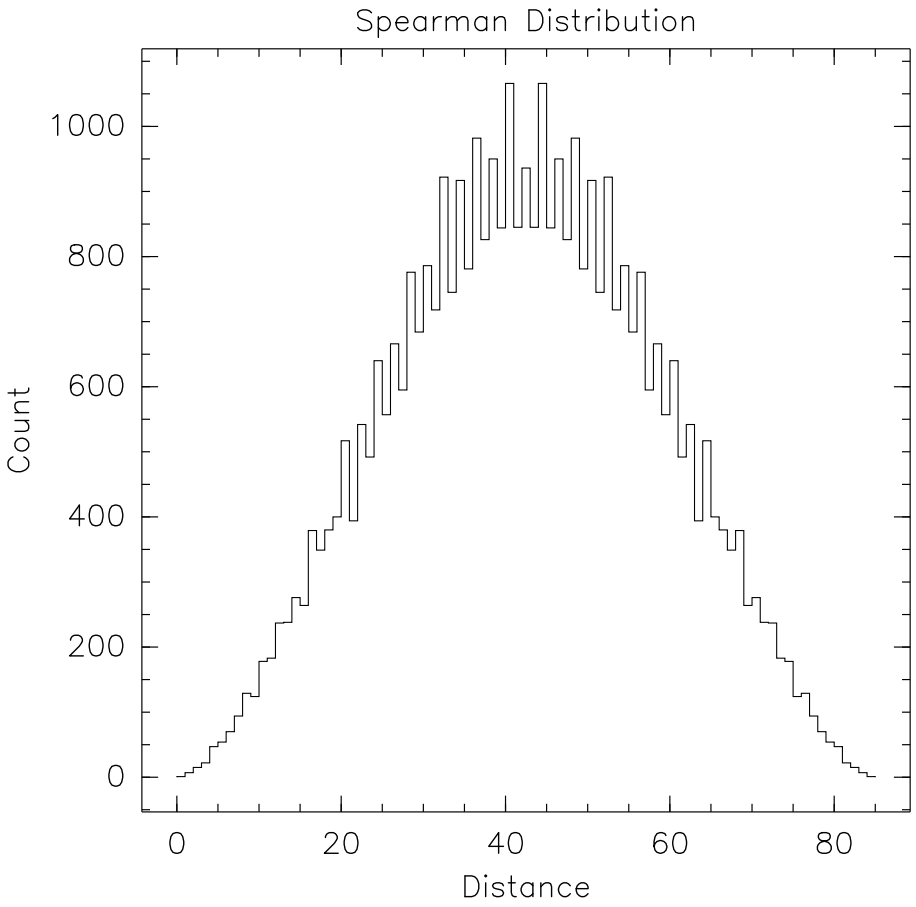


Figure 2.4: Distribution of Spearman distance for a list of length 8.

2.3 Spearman's Rho vs. Kendall's Tau

As shown in Figure 2.3, the normalizations of ρ and τ are based upon a score for the differences between ranks that ranges from zero, when the ranks are the same, to a maximum score that occurs when one rank is the reverse of the other. A natural question is whether the methods are redundant, that is, do the two correlations always agree upon how close two permutations are. An examination of Figure 2.5 will show that this is not true. In Figure 2.5, a point is placed at the intersection of each permutation's ρ and τ coefficient measured from the same arbitrary permutation. If the correlations were the same, then all of the dots would fall along the diagonal. Monjardet [40] shows that the maximum difference between ρ and τ approaches 0.5 as $n \to \infty$. Monjardet also gives some examples for the structures of the permutations that give the same score for ρ as for τ, but he is unable to provide an exhaustive list.

Spearman's Rho has more unique scores with which to assign a permutation to, thereby creating more bins into which to sort all the permutations of S_n. This provides a finer scale to use when evaluating the difference between permutations.

2.4 Spoilers

When comparing two permutations using Kendall's metric, it is possible that there are one or more elements of the list causing most of the inversions. For example, if $\pi_i =$ **1 2 3 4 5 6 7 8** and $\pi_j =$ **8 1 2 3 4 5 6 7**, then $d_K(\pi_i, \pi_j) = 7$. Element **8** is involved in all 7 inversion pairs. If this element were to be removed, then the coefficient τ would improve from 0.5 to 1. Such elements having high (relative to the rest) inversion counts are called *spoilers*.

The Spearman equivalent of spoilers are those elements with the greatest absolute difference between the ranks. For element x in permutation π_i and π_j, the distance is $|\pi_i(x) - \pi_j(x)|$. In the previous example, the rank vectors are $\pi_i =$ *1 2 3 4 5 6 7 8* and $\pi_j =$ *2 3 4 5 6 7 8 1*. Then $d_S(\pi_i, \pi_j) = 7.483$ and the coefficient $\rho = 0.91$. Removing the 8th rank improves ρ to 1.

Knowing which, if any, elements of a list are spoilers illustrates the structure of the list. A spoiler in a biological context is a definite candidate for further investigation. Spoilers show promising applications in combining models and feature selection.

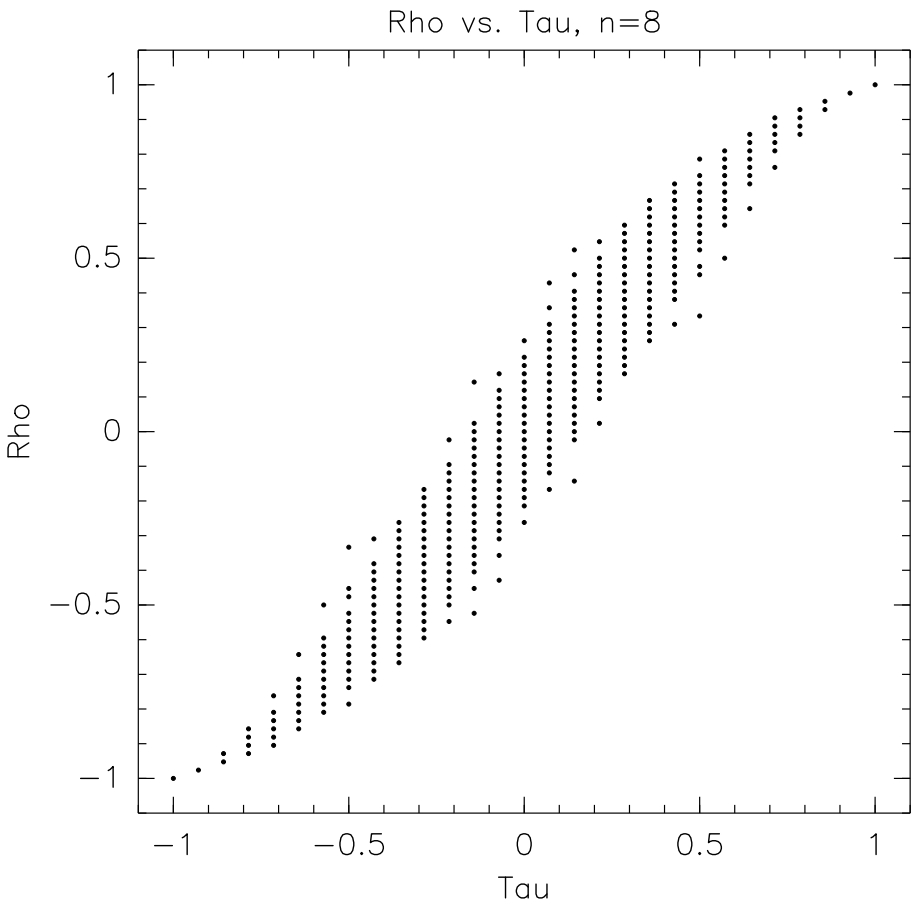


Figure 2.5: Distribution of ρ vs τ for lists of length 8.

2.5 Concordance of Permutation Sets

The *concordance* (W) of a set of permutations is the amount of agreement in that set [26, page 94]. If there are m permutations with n ranks then their rank vectors would form an $m \times n$ matrix. If all of the ranks vectors were identical, then the sums of the columns would consist of

$$m, 2m, 3m, \ldots, nm,$$

but not necessarily in that order. The deviations from this ideal order would be

$$-\frac{1}{2}m(n-1), \quad -\frac{1}{2}m(n-3), \quad \ldots, \quad \frac{1}{2}m(n-1).$$

The sum of squares of these deviations is $\frac{1}{12}m^2(n^3-n)$. Let s be the sum of the squares of the actual deviations, then

$$W = \frac{12s}{m^2(n^3-n)}$$

The concordance of a set of permutations provides a measure of how close the order of each permutation is to every other permutation within the set using d_S as a distance coefficient. The range of the concordance is from 0 to 1. If the concordance is 1, then every permutation has the exact same order; if 0, they are as different from each other as they can possibly be.

2.6 Algorithms

The algorithm shown in Figure 2.6 computes the ρ concordance of a set of permutations. The complexity of the algorithm is $\Theta(nm)$ where n is the length of the rank vectors and m is the number of rank vectors. The corresponding algorithm for computing the τ concordance is shown in Figure 2.7.

One way to classify an unknown instance is to add it to the set of instances for each class and compute the new concordance. Whichever set decreases the least (or improves the most) is the class that the new instance belongs to. This is a time consuming process, since it needs to be done for each new instance. A better way would be to find the central rank order for each class. This rank order is the ideal rank order that exemplifies that class.

RhoConcordance(Π).
INPUT: A set of permutations $\Pi = \{\pi_1 \dots \pi_m\}$, each of length n
OUTPUT: The ρ Concordance.

```
1   sum, rankTotals[] ← 0
2   for i ← (1 : n)
3       for j ← (1 : m)
4           π ← Π(j)
5           rankTotals(i) ← rankTotals(i) + π(i)
6           sum ← sum + π(i)
7   mean ← sum / n
8   S ← 0
9   for i ← (1 : n)
10      S ← S + (rankTotals(i) - mean)^2
11  return (12 × S)/(m^2 × (n^3 − n))
```

Figure 2.6: Computing the ρ concordance for a set of permutations.

TauConcordance(Π).
INPUT: A set of permutations $\Pi = \{\pi_1 \dots \pi_m\}$, each of length n
OUTPUT: The τ concordance.

```
1   S ← 0
2   for i ← (1 : n − 1)
3       for j ← (i + 1 : n)
4           sub ← 0
5           for k ← (1 : m)
6               π ← Π(k)
7               if π(i) < π(j) then
8                   sub ← sub + 1
9               else
10                  sub ← sub - 1
11          S ← S+ sub^2
12  return (2 × S/(n^2 − n)(m^2 − m)) − 1/(m − 1)
```

Figure 2.7: Computing the τ concordance for a set of permutations.

RhoCenter(Π).
INPUT: A set of permutations $\Pi = \pi_1 \dots \pi_m$, each of length n
OUTPUT: A permutation that is the ρ Center.

```
1  rankTotals[] ← 0
2  for i ← (1 : n)
3       for j ← (1 : m)
4            π ← Π(j)
5            rankTotals(i) ← rankTotals(i) + π(i)
6  π ← (1 : n)
7  sort π by the values of rankTotals
8  return π
```

Figure 2.8: Computing the ρ Center for a set of permutations.

The algorithm RhoCenter in Figure 2.8 finds the permutation that is the closest to all other permutations in the least squares sense using d_S. The run time complexity of this algorithm is $\Theta(nm)$. The corresponding algorithm for τ is the well known linear ordering problem [15] and is NP-complete.

Using this, a new instance only needs to be compared to each central order, which is a $\Theta(n)$ operation rather than an $\Theta(nm)$ operation for each class.

The mean concordance of a set of sets of permutations is computed by the algorithm in Figure 2.9. Figure 2.10 is an algorithm that computes a center permutation for each set, and then computes the concordance of that set of centers. Each algorithm is $O(cnm)$ where c is the number of sets.

MeanConcordance(Ω).
INPUT: A set of sets of permutations $\Omega = \Pi_1 \ldots \Pi_c$
OUTPUT: The mean ρ Concordance.

```
1  total ← 0
2  for i ← (1 : c)
3       Π ← Ω(i)
4       total ← total + RhoConcordance(Π)
5  mean ← total / c
6  return mean
```

Figure 2.9: Computing the mean ρ Concordance for a set of sets of permutations.

ConcordanceOfCenters(Ω).
INPUT: A set of sets of permutations $\Omega = \Pi_1 \ldots \Pi_c$
OUTPUT: The ρ Concordance of the set of center permutations.

```
1  Π_C ← {}
2  for i ← (1 : c)
3       Π ← Ω(i)
4       π ← RhoCenter(Π)
5       Π_C(i) ← π
6  result ← RhoConcordance(Π_C)
7  return result
```

Figure 2.10: Computing the concordance of a set of ρ Centers for a set of sets of permutations.

Chapter 3

Model Evaluation

Biology provides a rich source of data and of complex models to explain that data. For many biological phenomena, none of the currently available models fully describes or explains the data under all conditions. Phenomena at the cellular level are of intense current interest. When designing a computer system to model cellular response to external conditions, it is necessary to employ biological models at different levels of abstraction. In the most abstract models, groups of genes that exhibit coordinated expression under experimental conditions can be mined, represented, and visualized as a network. At lower levels of abstraction, the expression of individual genes (genomics) and the determination of *in vivo* quantities of thousands of proteins (proteomics) can be evaluated and analyzed using clustering algorithms or inductive logic programming. A complete set of accurate models is far beyond current computational or modeling technology.

However, the models in this project have limited and tractable goals: to estimate gene and protein expression under combinations of conditions that can be inferred from actual experimental data. The experimental data come from numerous sources of diverse types, including known DNA sequences from genome sequencing projects, protein abundance measured under different experimental conditions, gene expression data, and *a priori* biological knowledge. From these data, models are created using a variety of methods, e.g., inductive logic programming, sequence and genome analysis, Bayesian networks, and statistical pattern recognition. For example, Ramakrishnan and Grama [43] have argued that inductive logic programming is well suited to analyzing gene expression data in the presence of *a priori* information about gene function, while whole genome analysis enhances our ability to detect patterns in biological processes at the level of an entire cell.

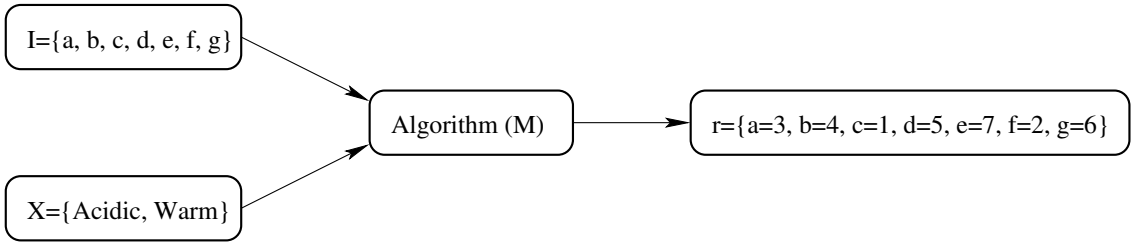


Figure 3.1: Graphical depiction of a model

All of these models give a useful, yet incomplete view of the processes taking place within the cell. A more complete understanding is gained by combining the results of these models into a consistent system-wide model by the use of information integration techniques. A combined model should have predictive and even explanatory power in the context of external environmental conditions. In particular, a model can be used as a substitute for costly biological experiments in isolating the essential biological processes.

Ideally, when combining models, one wants to take the best elements of each model to form the combined model. Therefore, there needs to be some objective assessment of when a model performs well. Typically, two kinds of models are used for predictive gene expression. The first kind will, given a gene, say yes or no to its expression being above a given threshold under certain conditions. The second kind assigns a score to the expression level of the gene. A simple analysis matches these scores against some arbitrary cutoff value with those scores exceeding the value considered to be significant. A more complex analysis, such as that presented by Wolfinger et al. [54] uses a normalization model based upon a global ANOVA of the expression levels. This practice misses important latent information in the data, namely the context in which the score was assigned.

For the purpose of this document, a model is an algorithm M that, given I, a set of items, and X, a set of external conditions, returns an ordered list, or ranking, r, of the items in I. A graphical depiction of a model is shown in Figure 3.1. Note that this is an abstract view, therefore the algorithm could be a predictive function or a biological experiment. The important point is that the result is a set of features that can be ranked.

Evaluating models using order comparisons eliminates a possible source of error common to other evaluation schemes. Measuring biological data is inherently uncertain, and different experiments might have different baselines of measurement. If this

baseline is unknown, then comparing the results of those experiments in an absolute sense, without being able to normalize the results, is impossible. In addition, a model prediction of expression level is not necessarily using the same units as an experimental procedure. Given the item set I containing items a and b, and two methods that give two different rankings, r_1 and r_2, of the items in I, a comparison is made between the results. If the baseline for the results is unknown, then the statement $r_1(a) < r_2(a)$ is not verifiable. The statement $(r_1(a) < r_1(b)$ and $r_2(a) < r_2(b))$, however, is verifiable since it depends only upon comparisons within the same experiment, even though the baseline for that experiment is unknown.

The early work in this area uses the z-score rather than the τ-correlation. This was done until it was realized how unwieldly the z-score becomes as n increases. Furthermore, d_K is the basis for the comparisons since Kendall and Gibbons [26, page 13] states that τ is preferable to ρ from many theoretical points of view. In spite of this, ρ can be substituted for τ for much of the following, as long as the substitution is consistent and the *tau*-correlation of one set of rank orders is never directly compared to the ρ-correlation of another set of orders.

3.1 Sources of Experimental Data

Experimental biological data comes from many sources. Two important sources are microarrays and mass spectrometry. The results of both of these allow a ranking of relevant features.

3.1.1 Microarrays

The central dogma of molecular biology states that DNA is transcribed into mRNA, which is translated into proteins. There is also a reverse transcription process where mRNA codes back to complementary DNA (cDNA). In microarrays, cDNAs labeled with fluorescent dyes are hybridized to an array of DNA probes affixed to a solid support. The array elements can be either synthetic oligonucleotides or larger DNA strands. Targets for thousands of genes can be present on a single microarray [8].

In a typical cDNA microarray experiment, a robotic device deposits DNA probes in a rectangular array on a series of glass slides. The slides are then hybridized to a mixture of two cDNA pools containing experimental and control samples that have been labeled with spectrally distinct fluorochromes. After stringency washes,

the microarray is scanned in a laser-scanning device and the image is processed to derive numerical intensity data [3]. The result is a list of transcription levels for every gene in the organism represented by a probe on the chip. Therefore in this experiment, M is the microarray, I is the set of gene probes, and r is an ordered list of genes.

3.1.2 Mass Spectrometry

There are several different mass spectrometry methods [42], including matrix assisted laser desorption ionization mass spectrometry (MALDI) and electrospray ionization mass spectrometry (ESI-MS). All are used to determine the masses of components in a mixture, starting from a very small sample. Specific proteins can be identified by their patterns of fragmented peptide masses and matched to a specific amino acid sequence. So far, this will only determine the presence or absence of a gene product.

Quantifying the amount of protein requires some other method, such as staining, radio labeling, fluorescence, or time-of-flight, in conjunction with the mass spectrometry. The result is a list of the amount of each protein (gene product) present in the organism at the time of measurement. Here, M is the mass spectrometry method, I is a set of proteins, and r is an ordered list of proteins.

3.1.3 MPSS

Massively Parallel Signature Sequencing (MPSS) is a new technology to obtain an absolute count of the expression level for each gene in a single experimental sample [6, 7]. For MPSS, a cDNA library of short (17-20 nucleotide) sequence tags (signatures) is generated from mRNA extracted from a tissue culture. Each tag is counted, and the tags are matched to the genomic sequence to determine the transcript for each signature. Meyers et al. [37, 38] applied this process to *Arabidopsis thaliana*. Gene expression levels derived from this process fall naturally into ranks and are readily evaluated using rank correlation methods.

Coughlan, Agrawal, and Meyers [12] performed a series of microarray analyses using the same tissue cultures from the MPSS *Arabidopsis* experiments for comparison of technologies. As they state in their paper:

> "To directly compare data between platforms, we will need to convert these measurements to a single common scale, based on the original fluorescent signals in the raw microarray data, and this will need to be compatible with the MPSS units normalized to 'transcripts per million'."

3.2 Predictive Models

A predictive model has one of several types of outputs. The output may be a list of numerical values representing quantity, one for each gene, from which a ranking may be obtained. The output may also be a list of rules, such as those created by inductive logic programming: "Under condition X, the expression levels of genes A and C are greater than gene B." This can be construed as a partial ordering on the list of genes. The ordering of genes A and B are known, as well as the ordering of genes B and C, but nothing is known about the relationship between genes A and C. Finally, the output may be a list of rules that do not constitute a partial ordering such as "If gene A is positively expressed under condition X, gene A is negatively expressed under condition Y." This rule does not position gene A with respect to any other gene.

3.2.1 Predicted Highly Expressed Genes

The predicted highly expressed (PHX) method of Karlin and Mrázak [41, 24, 25] employs the codon bias of a prokaryotic gene to predict its expression level. The codon bias of a gene is the tendency for a gene to use certain codon triplets for an amino acid rather than alternate codon triplets. An organism has a codon bias as well in the tRNA with which it assembles proteins from amino acids in the cytoplasm. If the codon bias of the gene matches the codon bias of the organism, then those proteins will be more readily constructed. These genes are presumably highly expressed. The codon bias of a cell is inferred from the codon bias of genes known to be highly expressed. By comparing this bias to the bias of an unknown gene, an expression level of the gene can be predicted. For PHX, I is a set of genes and r is an ordered list of genes.

To compute the PHX score of a gene, first the codon frequencies are normalized with respect to the amino acids. Let G be a set of genes, a be an amino acid, x, y, z be a codon triplet coding for a, and $g(x, y, z)$ be the average codon frequency for

the codon triplet (x, y, z) in the set of G genes, normalized such that

$$\sum_{(x,y,z)=a} g(x, y, z) = 1,$$

where the sum extends over all codons (x, y, z) whose translation is amino acid a. Similarly, let $f(x, y, z)$ indicate the normalized average codon frequencies for gene family F. The frequency of an amino acid a in a gene family is computed by the function

$$p_a(F) = \frac{\sum_{a \in F} n_a}{n_g},$$

where n_a is the count of amino acid a and n_g is the number of genes in family F. Then the codon bias of gene family F with respect to gene family G is

$$B(F|G) = \sum_{\text{All } a} p_a(F) \left[\sum_{\text{tr}(x,y,z)=a} |f(x, y, z) - g(x, y, z)| \right],$$

as shown by Karlin and Mrázek [24].

For example, consider the following synthetic amino acid sequences for genes in families F and G:

Gene	Sequence						
F_1	Met	Ile	Asp	Ile	Lys	Asn	Ile
F_2	Met	Ile	Asp	Ile	Lys	Asn	Ile
G_1	Met	Ile	Asp	Ile	Lys	Asn	Ile
G_2	Met	Ile	Asp	Ile	Lys	Asn	Ile
G_3	Met	Ile	Asp	Ile	Lys	Asn	Ile
G_4	Met	Ile	Asp	Ile	Lys	Asn	Ile

which have the following DNA sequences:

Gene	Sequence						
F_1	AUG	AUA	GAC	AUC	AAA	AAC	AUA
F_2	AUG	AUC	GAC	AUU	AAG	AAC	AUC
G_1	AUG	AUC	GAU	AUC	AAG	AAU	AUC
G_2	AUG	AUU	GAU	AUC	AAA	AAU	AUU
G_3	AUG	AUA	GAC	AUC	AAG	AAU	AUU
G_4	AUG	AUU	GAU	AUC	AAA	AAU	AUC

The counts for the above codons are:

AA	Codon (F-count|G-count)		
Met	`AUG` (2|4)		
Ile	`AUA` (2|1)	`AUC` (3|7)	`AUU` (1|4)
Asp	`GAC` (2|1)	`GAU` (0|3)	
Lys	`AAA` (1|2)	`AAG` (1|2)	
Asn	`AAC` (2|0)	`AAU` (0|4)	

For the amino acid isoleucine, $f(\texttt{AUA}) = 1/3$ and $g(\texttt{AUA}) = 1/12$, therefore $f(\texttt{AUA}) - g(\texttt{AUA}) = 1/4$. Similarly, $f(\texttt{AUC}) - g(\texttt{AUC}) = -1/12$ and $f(\texttt{AUU}) - g(\texttt{AUU}) = -1/6$. The frequency $p_{\text{Ile}}(F) = 3$, therefore the bias for isoleucine is $B(F_{\text{Ile}}|G_{\text{Ile}}) = 3(\frac{1}{4} + \frac{1}{12} + \frac{1}{6}) = \frac{3}{2}$. Similarly, $B(F_{\text{Met}}|G_{\text{Met}}) = 2$, $B(F_{\text{Asp}}|G_{\text{Asp}}) = \frac{3}{2}$, $B(F_{\text{Lys}}|G_{\text{Lys}}) = 0$, and $B(F_{\text{Asn}}|G_{\text{Asn}}) = \frac{3}{2}$. Therefore $B(F|G) = 6.5$.

The idea behind the PHX method is that certain genes families are known to be highly expressed. If the codon bias of a gene is different from the standard bias of all genes (C), and matches the codon bias of the known highly expressed gene families, then that gene will also be highly expressed. The standard gene families that Karlin and Mrázak take to be highly expressed are ribosomal protein genes (RP), chaperone/degradation genes (CH), and the translation/transcription processing genes (TF).

Karlin and Mrázak assign these families different weights and estimate the expression level of a gene with the following formula:

$$E(g) = \frac{B(g|C)}{\frac{1}{2}B(g|RP) + \frac{1}{4}B(g|CH) + \frac{1}{4}B(g|TF)}.$$

For example, the gene slr0335 from the *Synechocystis* PCC6803 genome has the codon biases $B(g|C) = 2.27$, $B(g|RP) = 1.73$, $B(g|CH) = 1.41$, and $B(g|TF) = 1.23$. Therefore the gene's expression score is:

$$E(g) = \frac{2.27}{\frac{1}{2} \times 1.73 + \frac{1}{4} \times 1.41 + \frac{1}{4} \times 1.23} = 1.49,$$

which is the highest PHX score of all genes in *Synechocystis*.

3.2.2 Hydropathy

The function of a protein depends on from its 3-dimensional structure. A simple indicator of protein structure is its hydropathy. Molecules can be hydrophobic

Amino Acid Sequence	R	K	D	B	N	S	E
Hydropathy Score	0	0.6	1	1	1	3.6	1

Sliding Window	RKD	KDB	DBN	BNS	NSE
Average Score	0.533	0.867	3	1.867	1.867

Figure 3.2: Computing the average hydropathy plot for the amino acid sequence RKDBNSE using a sliding window of size 3.

(water-fearing), hydrophilic (water-loving), or neither. The hydropathy of a protein is a measure of how hydrophobic or hydrophilic it is and can be estimated from the hydrophobic and hydrophilic tendencies of the amino acids in the protein. Kyte and Doolittle [30] assign a hydropathy score to each amino acid, based on its relative hydrophobicity or hydrophilicity. They create a plot by computing the average value for a sliding window over the entire sequence. This shows which regions are hydrophobic and which are hydrophilic. An example is shown in Figure 3.2.

Garay-Arroyo *et al.* [16] extend the work of Kyte and Doolittle and state that the overall average hydropathy of a protein can be important to its expression under certain conditions. Specifically, highly hydrophilic proteins in prokaryotes and eukaryotes are common during water deficit conditions. In an amino acid sequence of length n, where a_i is the amino acid at position i, H_i is the hydropathy score for the amino acid a position i, and G is the amino acid Glycine, the following formula is used to score the hydropathy of a protein:

$$100 \times \left(\frac{100}{n} \times \sum_{i=1}^{n} g(i) - 8 \right) \times \left(\frac{\sum_{i=1}^{n} H_i}{n} - 1 \right), \text{ where } g(i) = \begin{cases} 1 & \text{if } a_i = \text{G} \\ 0 & \text{otherwise} \end{cases}.$$

It is evident that, to be considered highly hydrophilic, the average hydrophilicity of a protein must be above 1, and the protein must contain at least 8% glycine. This formula provides a score, but also may be used to rank the genes. Therefore, for the hydropathy model, I is a set of genes and r is an ordered list of genes.

3.3 Assessment and Improvement

There are two goals of this research, namely, model assessment and model improvement. If the output of a model predicting gene expression levels is interpreted as an ordered list, ranked by expression level, then the model can be evaluated by comparing its predicted order to the order determined by experimental data. Experimental

data is expensive to obtain. If it is not feasible to obtain for all samples, a subset of the predicted data can be compared to the experimental data. The underlying idea is that, if the model predicts the expression levels accurately for the subset, then the remaining predictions gain some plausibility.

Ordered lists can be compared by using order statistics. The intuition is that, if the score of the predicted list is no better than the score of a randomly generated list order, then the model is hardly a good predictor.

There are two ways to interpret a model that gives a list with a score worse than that of a randomly generated list. The first is that the model is simply a poor predictor and its results are therefore discarded. The second is to take the model as predicting the opposite of what actually occurs, so to find a better model, invert the output of that model. For example, if model M_1 has a negative score under condition X, then create a new model M_2, where M_2 orders the gene in the reverse order to M_1.

The output of a model can be evaluated through comparison with experimental results. It is then possible to combine models in a primitive fashion if the goal is well defined. For example, if there are two models, M_1 and M_2, and M_2 is known to perform better under all temperatures X except when X is 35°C, then a combined model M_c can be defined as follows:

$$M_c(X, I) = \begin{cases} M_1(35°\text{C}, I) & \text{if } X = 35°\text{C}; \\ M_2(X, I) & \text{otherwise.} \end{cases}$$

Or the question could be posed, "What are the expression levels of genes for functional subgroup Y, as a function of temperature?" The results from the model that best predicts results for that functional subgroup are then chosen. Another approach combines models in such a way that the overall order is improved by combining the output of two models using the knowledge of which subgroups are better predicted under which model. For example, given the item sets $I = \{a, b, c, d, e, f, g\}$ and $J = \{d, e, f\}$ under physiological condition X, suppose $M_1(X, I)$ is a better predictor than $M_2(X, I)$, but $M_2(X, J)$ is a better predictor than $M_1(X, J)$. The new combination model M_c is

$$M_c(X, I) = M_1(X, I - J) + M_2(X, J),$$

where the '+' represents some method of combining models.

For example, if $M_1(X, I) = \mathbf{g\ b\ c\ a\ f\ e\ d}$ and $M_2(X, J) = \mathbf{d\ e\ f}$ then a plausible method of combining the models is to maintain the relative order of elements J

Types	Subsequence		Method of Combining
	Order	**Support**	
A	$d_K(M^a, M_1^a) < d_K(M^a, M_2^a)$	$\text{Sup}(M, M_1, a) < \text{Sup}(M, M_2, a)$	Use order and positions of a from M_2 for a in M_1
B	$d_K(M^a, M_1^a) < d_K(M^a, M_2^a)$	$\text{Sup}(M, M_1, a) > \text{Sup}(M, M_2, a)$	Use order of a from M_2 for a in M_1
C	$d_K(M^a, M_1^a) > d_K(M^a, M_2^a)$	$\text{Sup}(M, M_1, a) < \text{Sup}(M, M_2, a)$	Use positions of a from M_2 for a in M_1
D	$d_K(M^a, M_1^a) \geq d_K(M^a, M_2^a)$	$\text{Sup}(M, M_1, a) \geq \text{Sup}(M, M_2, a)$	Do nothing

Figure 3.3: Possibilities for comparing subsequence a in the output of models M_1 and M_2 to the authoritative data in M.

and combine that with the order of $I - J$ from I. Therefore $M_c(X_I)$ would be **g b c a d e f**.

3.4 Combining Models

Let a be a subset of L_n and π be a permutation in S_n, then π^a is a subsequence of π with elements of a in the same relative order as they occur in π. Let b also be a subset of L_n and let π_i and π_j be permutations of S_n. If $d_K(\pi_i^a, \pi_j^a) < d_K(\pi_i^b, \pi_j^b)$, then the order of subset a is more consistent than the order of subset b.

Let $\neg a$ be those elements in L_n not in a. If the coefficient $\tau(\pi_i^{\neg a}, \pi_j^{\neg a})$ is greater than the coefficient $\tau(\pi_i, \pi_j)$ then subset a is important to the coefficient $\tau(\pi_i, \pi_j)$, if the coefficient is less, then that subset is detrimental to the coefficient $\tau(\pi_i, \pi_j)$. The support function is:

$$\text{Sup}(\pi_i, \pi_j, a) = \tau(\pi_i, \pi_j) - \tau(\pi_i^{\neg a}, \pi_j^{\neg a})$$

There are four possible types of outcomes when the output of two models is evaluated against authoritative data as shown in Figure 3.3. The order column compares the consistency of the subset orders and the support column evaluates the importance of that subset.

The solutions to combining models in Figure 3.3 are heuristics and are not always guaranteed to give results that are better than they were before being combined. Figure 3.4 shows the probability of the method of combination actually improving the overall score of a model given two random permutations meeting the criteria for the conditions. The list used was $\{1, \ldots, n\}$ and the subsequence was comprised of all of the even numbers in the list.

	Condition		
N	**A**	**B**	**C**
4	94.7%	100.0%	90.9%
5	90.4%	100.0%	68.9%
6	92.4%	100.0%	86.5%
7	89.7%	100.0%	72.4%
8	92.6%	94.6%	81.6%
9	90.7%	94.9%	70.3%
10	92.9%	90.3%	77.7%
20	94.3%	85.2%	68.7%
30	95.1%	82.4%	64.9%

Figure 3.4: Percentage of permutations of length $(1 \ldots n)$ whose order was improved using the subsequence of even numbers from a different permutation

Figure 3.5 shows the distribution of the comparison between order of the subsequence of even numbers in π_1 vs. the order of the subsequence of even numbers in π_2 on the x-axis. The y-axis shows the difference in probabilities using the z-score of π_1 and π_2 with the even numbers removed. The plot shows 10,000 random comparisons of permutations of a list of length 30 to another random permutation of the list. The total number of possible comparisons is $(n!)^2$, which is far too many to show here. The motivation of the plot is to show the range of possible effects the internal order of a group can have on the overall order.

The following experiments take the results of two predictive biological models and compare the results with the actual expression levels of known genes from *E. coli* under various conditions [50, 53]. At the time of this writing, Genbank lists 3907 genes for *E. coli*, of which approximately 46 to 80 genes had known expression levels under various conditions which are shown in Figure 3.6.

The first model used for these experiments is the PHX model described in Section 3.2.1. The data were not taken from the paper, rather they were computed based upon the current known genomic data from Genbank. The second model (F25) is a modified version of PHX created by the author based upon the suggestion of Chen and Inouye [10] that the bias of the first 25 codons of an open reading frame (ORF) is most important to gene expression. In this model, the PHX algorithm is applied to only the first 25 codons in each gene.

The genes were assigned to functional groups shown in Figure 3.7 using Clusters of Orthologous Groups (COGs) as defined in Tatusov *et al.* [51, 52]. Figure 3.8 shows

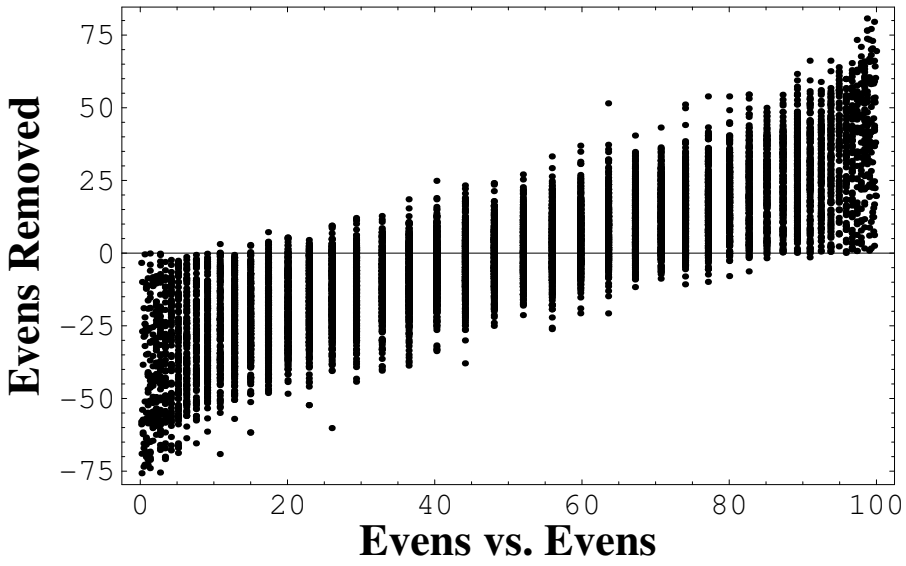


Figure 3.5: Distribution of the subsequence of even numbers for permutations of a list $\{1, \ldots, 30\}$.

the relative orders of expression levels from the AB column of the *E. coli* data, and the orders predicted by the PHX and F25 models. The numbers following the gene name are either the expression levels for column AB, or the scores predicted by the models.

Results

Figure 3.9 shows the evaluation of the predictions made by the two models under various conditions. The correspondence is measured using the z-score, therefore, the lower the percentage, the higher the correspondence. Note that, with the exception of condition T46, the PHX model is superior to the F25 model.

Figures 3.10a and 3.10b show the relationship between how well the model predicts the ordering within the group vs. how much the group supports the overall ordering. For the group to group ordering (y-axis), the value is the probability that two orders are this close by chance. So the lower the score is, the more significant the result. Note that for Figure 3.10a, some of the values exceed 50%, so the other interpretation is possible, that this model is predicting the reverse order under those conditions. For the x-axis, the value is the difference in probability if this entire group is removed

Name	Description
AB	Abundance of proteins from cultures grown in glucose minimal MOPS at 7°C in alpha prime (a') units X 103. These units multiplied by 0.1 give the percent of total protein.
ACE	Relative level of proteins from cultures grown in acetate minimal MOPS compared to glucose minimal MOPS at 37°C
GLY	Relative level of proteins from cultures grown in glycerol minimal MOPS compared to glucose minimal MOPS at 37°C
RIC	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS compared to glucose minimal MOPS at 37°C
T13.5	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 13.5°C compared to 37°C
T15	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 15°C compared to 37°C
T23	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 23°C compared to 37°C
T30	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 30°C compared to 37°C
T42	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 42°C compared to 37°C
T46	Relative level of proteins from cultures grown in glucose rich (amino acid, bases and vitamins) MOPS at 46°C compared to 37°C

Figure 3.6: Description of conditions

Information storage and processing	
J	Translation, ribosomal structure and biogenesis
K	Transcription
L	DNA replication, recombination and repair
Cellular processes	
D	Cell division and chromosome partitioning
O	Post-translational modification, protein turnover, chaperones
M	Cell envelope biogenesis, outer membrane
N	Cell motility and secretion
P	Inorganic ion transport and metabolism
T	Signal transduction mechanisms
Metabolism	
C	Energy production and conversion
G	Carbohydrate transport and metabolism
E	Amino acid transport and metabolism
F	Nucleotide transport and metabolism
H	Coenzyme metabolism
I	Lipid metabolism
Poorly characterized	
R	General function prediction only
S	Function unknown

Figure 3.7: COG Functional Groups

AB	**PHX**	**F25**
aceE(8.5)	*atpA*(1.56)	*aceE*(1.06)
atpA(6.9)	*aceE*(1.53)	*sucA*(1.05)
atpD(5.6)	*pta*(1.53)	*aceF*(1.05)
lpdA(4.6)	*lpdA*(1.45)	*lpdA*(1.03)
ppc(4.3)	*ackA*(1.44)	*gltA*(1.02)
mdh(2.6)	*aceF*(1.34)	*atpA*(1.01)
sucA(1.9)	*atpD*(1.32)	*atpD*(1.01)
sucB(1.7)	*sucC*(1.29)	*sucB*(0.99)
sucC(1.6)	*mdh*(1.15)	*sdhA*(0.97)
pta(1.5)	*sucB*(1.11)	*sucC*(0.94)
sdhA(1.1)	*gltA*(1.05)	*ackA*(0.92)
gltA(1.1)	*sucA*(1.04)	*pta*(0.90)
ackA(1.0)	*sdhA*(0.90)	*gor*(0.89)
aceF(0.9)	*ppc*(0.74)	*mdh*(0.87)
gor(0.5)	*gor*(0.63)	*ppc*(0.86)

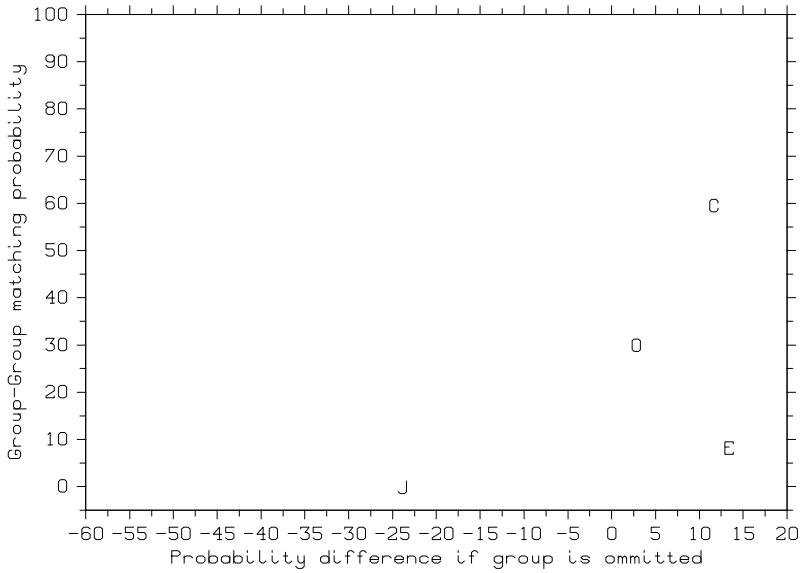
Figure 3.8: Selected lists of gene expression levels for functional group C, energy production and conversion.

	Model	
Conditions	PHX	F25
AB	0.0017%	13.098%
ACE	0.0073%	30.904%
GLY	0.0002%	17.602%
RIC	<0.0000%	7.1560%
T13.5	2.4240%	5.4816%
T15	0.8738%	2.0893%
T23	0.6733%	1.1250%
T30	0.4725%	1.2552%
T42	0.3266%	1.4391%
T46	21.894%	15.454%

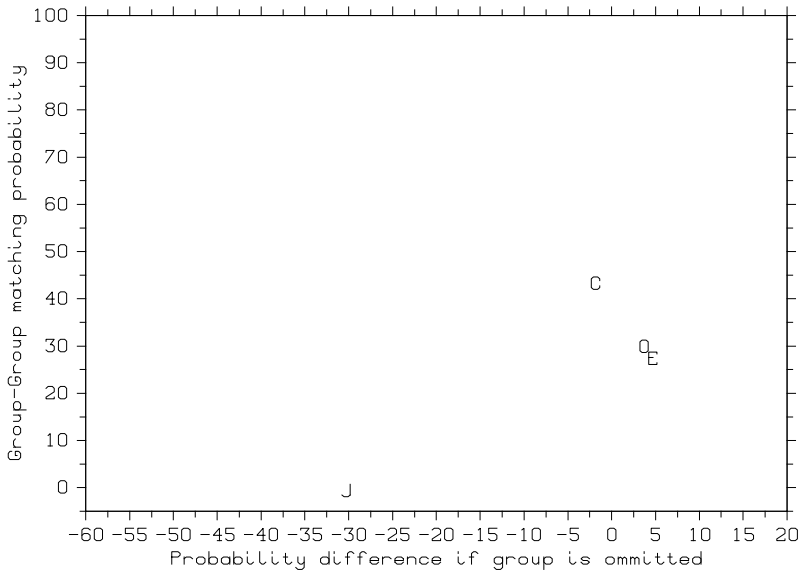
Figure 3.9: Likelihood of correspondence between model prediction and actual abundance levels.

from the computation. For a negative value, the probability is less, so that group is important to the results. For a positive value, the probability is greater, so the inclusion of that group has a detrimental effect on the outcome.

Overall, model F25 is a better predictor under physiological condition T46 than PHX, however, group C under PHX is better predicted under physiological condition T46 than group C under F25. As can been seen from Figures 3.10 a and b, group C meets the criteria for condition C as shown in Figure 3.3, therefore the positions of group C in PHX will be used in the combined model, but the order of group C from F25 will be preserved. The results are shown in Figure 3.11. The overall score of F25 was 15.454%, the score of F25 with the addition of group C from PHX is 7.873%, a significant improvement.



a) Model PHX



b) Model F25

Figure 3.10: Group expression level prediction scores for two models under physiological condition T46

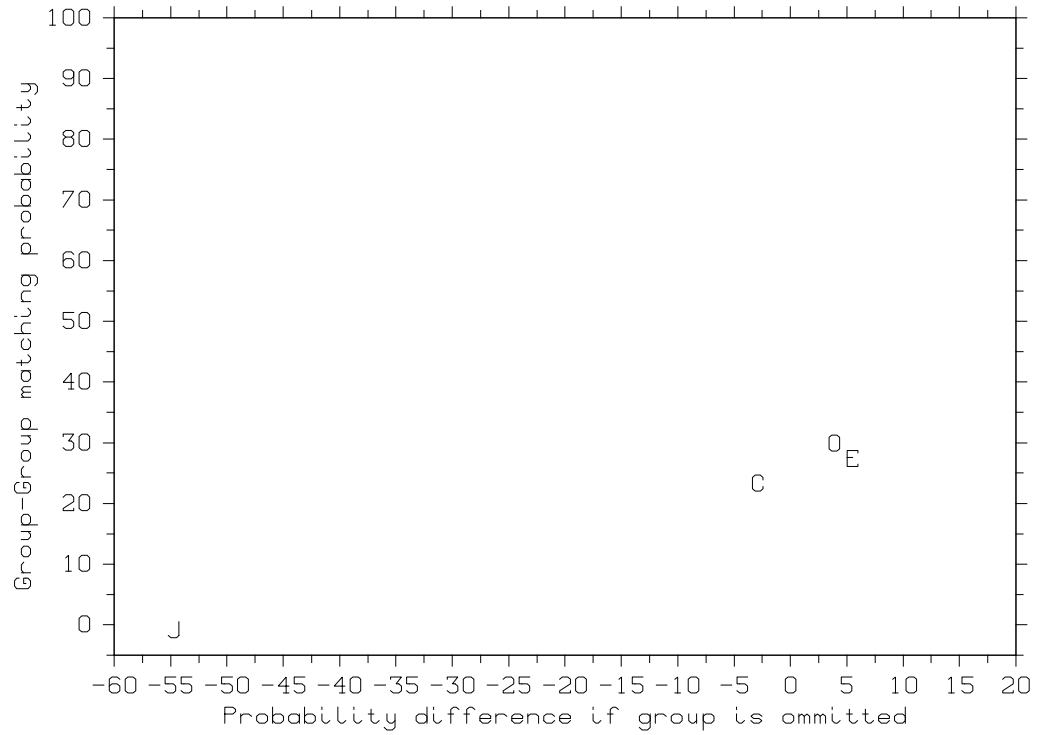


Figure 3.11: Group expression level prediction scores for an improved model.

Chapter 4

MPSS vs. Microarrays

The MPSS data is described in Meyers et al. [38]. The data consists of 14 *Arabidopsis* tissue libraries, each with 20,000 to 50,000 distinct signatures. The 14 libraries are listed in Figure 4.1. Meyers et al. [38] classify each signature according to its significance, reliability, and location on the strand. For this paper, all signatures used are those that are significant, reliable, and within one exon on the same strand (group 1 in their paper).

The MPSS data was collected from the same tissue samples using both 17-base and 20-base sequence signatures. The correlations between the resulting orders is shown in Figure 4.2. The results are very consistent, with an average $\tau \approx 0.901$. This consistency can be clearly seen in Figure 4.3 which is a scatter plot of the abundance levels for the S52 library.

Coughlan, Agrawal, and Meyers [12] conducted microarray analysis examining the differential expression between the same leaf (LEF) and root (ROF) tissues used in the MPSS experiments. Four replicates were created with replicates 3 and 4 being a dye-swap of replicates 1 and 2. The correlations of all replicates are compared in Figure 4.4 with the orders of replicates 3 and 4 reversed for the comparison because of the dye-swap.

The average τ of 0.706 is not quite as high as with the MPSS experiments, however there are far fewer pairs of ties for microarrays than there are for MPSS. These results are based on the 8,372 genes that the microarrays have in common with the 17-base MPSS root and leaf experiments.

To compare the microarray results to MPSS, 7,837 genes were ordered by their

Library	Code	Description
Callus	CAF	Actively growing
Inflorescence	INF	Mixed stage, immature buds
Leaves	LEF	21 day, untreated
Root	ROF	21 day, untreated
Silique	SIF	24 to 48 hr post-fertilization
Ap1-10 inflorescence	AP1	Mixed stage, immature buds
Ap3-6 inflorescence	AP3	Mixed stage, immature buds
Agamous inflorescence	AGM	Mixed stage, immature buds
Inflorescence	INS	Mixed stage, immature buds
Root	ROS	21 day, untreated
Sup/Ap1 inflorescence	SAP	Mixed stage, immature buds
Leaves, 4 hr	S04	After salicylic acid treatment
Leaves, 52 hr	S52	After salicylic acid treatment
Leaves, 21 day	LES	Untreated

Figure 4.1: Listing of all MPSS signature libraries extracted from the *Arabidopsis* tissue samples used in this paper. Includes a three letter abbreviation and a short description.

microarray signal intensities and then compared that order to the MPSS transcript abundance order. The ratio of expression levels for each gene in the LEF library and the ROF library was also computed for MPSS and compared to the log ratio of the differentially expressed genes in the microarrays. Whether the values compared are ratios or the log ratios does not make a difference since the resulting ranks will be the same. Figure 4.5 shows the τ correlations between the MPSS experiments and the microarray replicates. Unlike the standard statistical correlation method used by Coughlan et al. [12, page 251], Kendall's τ does not show that the log ratio produces a better correlation than that observed when the absolute microarray signal intensities for either tissue alone were compared to MPSS transcript abundance.

It is evident that MPSS is a consistent technique since the τ-correlation for all runs were at least 0.87 ($p << 0.001$), despite the fact that different lengths of signature sequences were used in each run. Microarrays are also fairly consistent between the replicates, though less so than MPSS. However, when comparing the results of microarray expression levels to MPSS expression levels, the correlation is much lower than either technique correlated with itself. If the correlation between MPSS and microarrays were high, then their orders must be similar, and MPSS and microarrays are measuring the same expression levels. The MPSS and microarray

	Number of Genes				
Library	17-base	20-base	$17 \cup 20$	τ-correlation	Paired ties
AP3	12505	12236	11882	0.876	4.0%
AGM	10152	9636	9261	0.881	4.2%
SAP	10619	10184	9794	0.886	3.0%
AP1	13202	12957	12598	0.891	5.4%
INS	12448	12286	11893	0.897	4.5%
ROS	11386	11046	10681	0.902	2.6%
ROF	10644	10447	9902	0.902	2.7%
INF	10965	10475	10006	0.907	2.4%
LES	11507	11311	10969	0.908	3.5%
LEF	11420	11144	10665	0.910	3.1%
SIF	10488	10120	9664	0.912	2.4%
CAF	10835	10550	10084	0.913	2.3%
S04	7017	6715	6452	0.917	4.1%
S52	7396	7230	6925	0.922	4.1%

Figure 4.2: Tau correlation for each MPSS signature library between sequence signatures of 17 nucleotide tags and 20 nucleotide tags. The total number of genes for each signature sequence length is given and the size of their intersection upon which the τ-correlation is performed. The percentage of pair ties is based on the total number of paired comparisons, where there is a pair of genes in either library that has the same expression level.

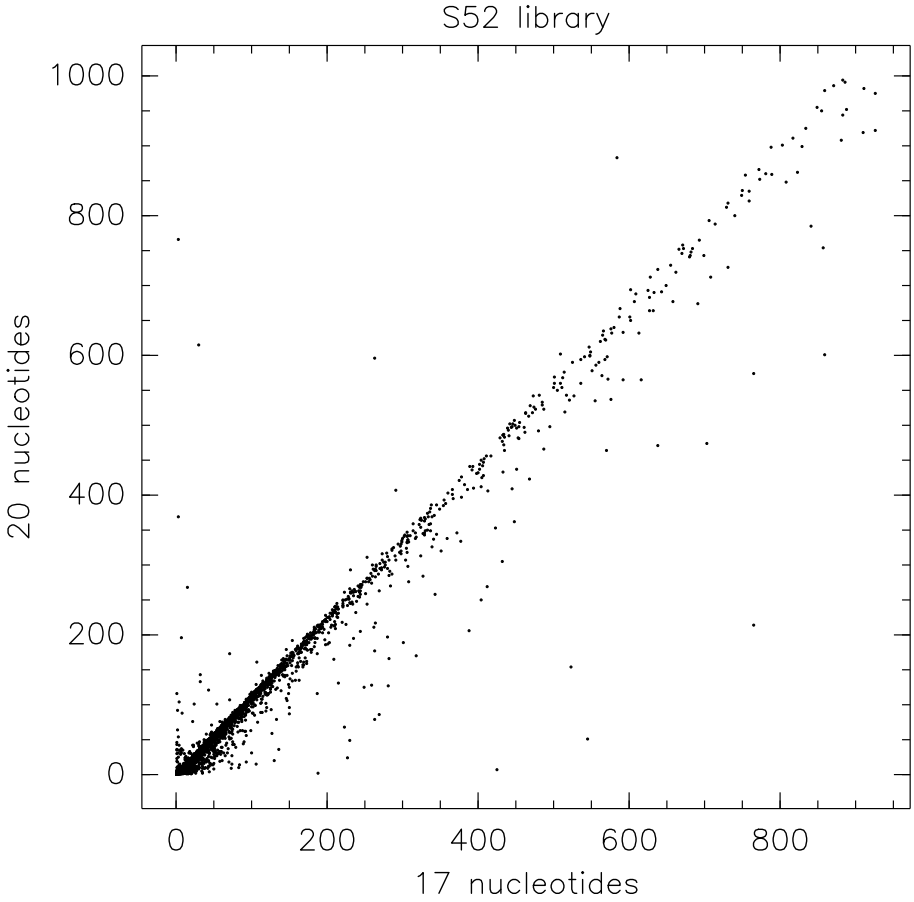


Figure 4.3: Comparison of MPSS signature abundance for 17 nucleotide tags with 20 nucleotide tags for the leaf library, 52 hours after salicylic acid treatment.

	τ-correlation	Paired Ties
1 vs. 2	0.836	0.002%
1 vs. 3	0.730	0.002%
1 vs. 4	0.622	0.003%
2 vs. 3	0.751	0.002%
2 vs. 4	0.619	0.002%
3 vs. 4	0.681	0.002%

Figure 4.4: Tau correlations between the four replicates of the leaf/root microarrays for each component of the expression level signal.

	Leaf		Root		Leaf/Root	
MPSS vs.	17-base	20-base	17-base	20-base	17-base	20-base
Replicate 1	0.472	0.472	0.446	0.447	0.422	0.425
Replicate 2	0.472	0.473	0.446	0.440	0.422	0.424
Replicate 3	0.458	0.459	0.439	0.440	0.428	0.430
Replicate 4	0.494	0.495	0.442	0.444	0.473	0.474

Figure 4.5: Tau correlation of both the 17 and 20 nucleotide tag MPSS data and the four microarray replicates for root and leaf expression levels.

	INF	LEF	ROF	SIF	AP1	AP3	AGM	INS	ROS	SAP	S04	S52	LES
CAF	0.39	0.40	0.47	0.38	0.28	0.32	0.31	0.30	0.35	0.28	0.22	0.25	0.26
INF		0.50	0.41	0.53	0.37	0.42	0.41	0.44	0.28	0.38	0.28	0.25	0.32
LEF			0.41	0.52	0.34	0.41	0.39	0.38	0.29	0.33	0.36	0.33	0.42
ROF				0.42	0.24	0.29	0.28	0.26	0.40	0.24	0.19	0.21	0.23
SIF					0.31	0.38	0.38	0.36	0.25	0.34	0.31	0.27	0.32
AP1						0.62	0.56	0.59	0.42	0.49	0.39	0.35	0.43
AP3							0.62	0.60	0.41	0.51	0.41	0.36	0.46
AGM								0.60	0.38	0.50	0.41	0.37	0.44
INS									0.38	0.49	0.38	0.34	0.43
ROS										0.35	0.29	0.28	0.36
SAP											0.37	0.34	0.38
S04												0.45	0.44
S52													0.39

Figure 4.6: Tau correlations for all pairwise comparisons among 17-base MPSS signature libraries expression level rank-orders.

measurements take place at different times in the DNA → mRNA → Protein process. Their measurements at those times are consistent with themselves, but not with each other. Therefore, either the expression levels at those times are different, or there is a bias inherent in the methods. Two possible sources of bias for microarrays are noise masking the low level signals or over-saturation for the high signal levels.

The τ correlation of the pairwise comparison of all 17-base MPSS signature libraries is shown in Figure 4.6. These comparisons were performed for all the genes with a non-zero expression level common to each paired comparison. The logarithmic scale scatter plots of the pairwise comparisons with the highest and lowest τ correlations are shown in Figure 4.7. The results for the 20-base MPSS as shown in Figure 4.8 are very similar.

It is useful to look at logical subsets of lists to compare and contrast their correlation with the correlation between the whole lists. Let $A \subseteq L_n$ and $\pi \in S_n$. The *restriction* π^A is the subsequence of the order vector of π consisting only of elements of A. Let

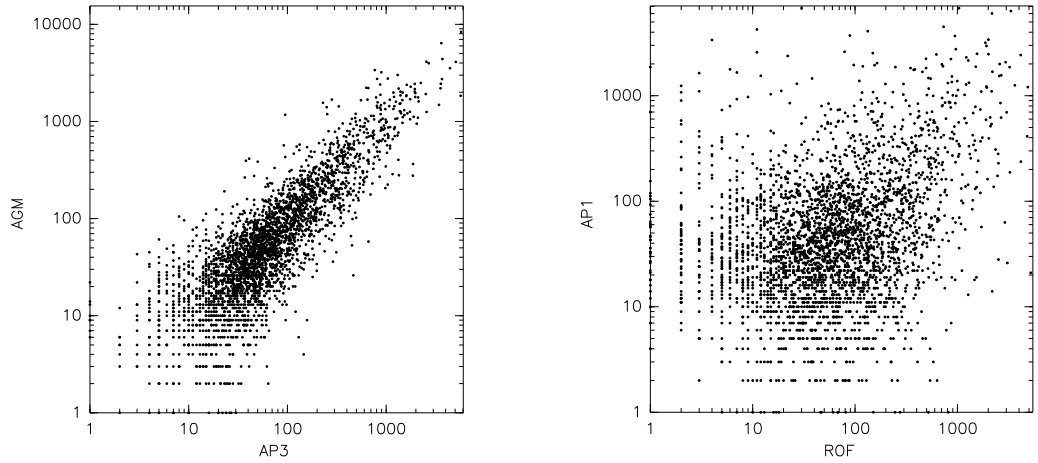


Figure 4.7: Comparison of 17-base MPSS expression levels for AGM vs. AP3 and AP1 vs. ROF as scatter plots on a logarithmic scale. AGM vs. AP3 has one of the highest τ correlations of 0.62 and AP1 vs. ROF has one of the lowest τ correlations of 0.24.

	INF	LEF	ROF	SIF	AP1	AP3	AGM	INS	ROS	SAP	S04	S52	LES
CAF	0.39	0.40	0.47	0.38	0.28	0.31	0.31	0.29	0.36	0.27	0.22	0.26	0.26
INF		0.49	0.41	0.53	0.37	0.41	0.41	0.44	0.28	0.37	0.28	0.26	0.31
LEF			0.41	0.52	0.34	0.29	0.39	0.37	0.30	0.34	0.36	0.33	0.41
ROF				0.42	0.24	0.29	0.28	0.26	0.40	0.24	0.19	0.21	0.23
SIF					0.31	0.37	0.38	0.35	0.26	0.32	0.30	0.27	0.31
AP1						0.62	0.55	0.59	0.41	0.48	0.39	0.35	0.43
AP3							0.61	0.60	0.41	0.50	0.41	0.37	0.46
AGM								0.60	0.39	0.50	0.41	0.38	0.44
INS									0.38	0.48	0.38	0.35	0.42
ROS										0.34	0.29	0.29	0.38
SAP											0.37	0.34	0.36
S04												0.46	0.44
S52													0.39

Figure 4.8: Tau correlations for all pairwise comparisons among 20-base MPSS signature libraries expression level rank-orders.

Comparison	Overall τ	GO-Slim category	Members	Category τ	τ Diff
INF vs. SIF	0.5880	transporter activity	278	0.5880	0.0
ROF vs. LES	0.2251	ER	46	0.2251	0.0
ROF vs. SIF	0.4737	other enzyme activity	493	0.4737	0.0
AGM vs. S52	0.4096	ribosome	168	0.6928	-0.2832
AGM vs. S04	0.4221	ribosome	168	0.7236	-0.2649
S04 vs. S52	0.5082	ribosome	168	0.7720	-0.2638
ROF vs. S04	0.2118	plastid	61	-0.2131	0.4249
ROS vs. S52	0.3109	plastid	61	-0.1060	0.4286
CAF vs. S04	0.2295	plastid	61	-0.2219	0.4514

Figure 4.9: Selected results from comparing the τ-correlations of the 17-base MPSS transcript abundance of 3,879 genes to the τ-correlations between the subset of genes (with the number of genes in each subset given) in the different GO-Slim categories. Both the overall τ-correlation and the τ-correlation of the genes within the category are shown, along with the difference between them.

$B \subseteq L_n$ and let $\pi_i, \pi_j \in S_n$. If $d_K(\pi_i^A, \pi_j^A) < d_K(\pi_i^A, \pi_j^A)$ then the order of subset A is more consistent than the order subset B.

Let $\neg A$ be those elements in π not in A. If the coefficient $\tau(\pi_i^{\neg A}, \pi_j^{\neg A})$ is greater than the coefficient $\tau(\pi_i, \pi_j)$ then subset A is supportive of the ordering of π_j, while, if the coefficient is less, then that subset is detrimental to the ordering of π_j.

The genes were divided into subsets according to their GO-Slim [17] categories, as determined by the October 10, 2004 updated GO-Slim. The τ correlation of the pairwise comparison for each GO-Slim category for all tissue libraries was computed. Some of the result are shown in Figure 4.9. These comparisons were performed for the 3,879 genes with a non-zero expression level common to all of the 17-base MPSS results.

Excluding a subset from a correlation computation and comparing the result to the original correlation shows how important that subset is to the overall agreement or disagreement between the original orders. This difference is greatly dependent upon the size of the subset. To measure the effect of removing a subset, relative to its size, the actual difference is divided by the maximum possible difference. The maximum possible number of inversions removed by a subset of size x is

$$\binom{n}{2} - \binom{n-x}{2} = \frac{x(2n-x-1)}{2}.$$

Comparison	Overall τ	GO-Slim category	Members	New τ	Significance
INF vs. SIF	0.5880	transporter activity	278	0.5797	20.8%
ROF vs. LES	0.2251	ER	46	0.2115	36.2%
ROF vs. SIF	0.4737	other enzyme activity	493	0.4624	26.8%
AGM vs. S52	0.4096	ribosome	168	0.3665	18.8%
AGM vs. S04	0.4221	ribosome	168	0.4155	16.9%
S04 vs. S52	0.5082	ribosome	168	0.4651	15.2%
ROF vs. S04	0.2118	plastid	61	0.3015	46.1%
ROS vs. S52	0.3109	plastid	61	0.3150	46.0%
CAF vs. S04	0.2295	plastid	61	0.2192	49.6%

Figure 4.10: Selected results from comparing the τ-correlations of the 17-base MPSS transcript abundance of 3,879 genes to the τ-correlations when the subset of genes (with the number of genes in each subset given) in the different GO-Slim categories are removed. The significance is percentage of the total change in the τ-correlation that the removal of a subset with the given number of members could have had.

Figure 4.10 shows the results of excluding the same subsets from the comparisons in Figure 4.9. Note that the ribosomes all have higher correlations between the subsets than the sets as a whole, and their removal lowered the correlation between the sets. The plastids have lower correlations between the subsets than the sets as a whole, and their removal raised the correlations between the sets. While the first three comparisons have the same correlation between the subsets as between the whole sets, their removal lowers the correlation. Therefore their position relative to the rest of the genes is more important than their relative order.

Chapter 5

Machine Learning

The problem of feature selection in supervised learning situations is considered, where all features are drawn from a common domain and are best interpreted via ordinal comparisons with other features, rather than as numerical values. In particular, each instance is a member of a space of ranked features. This problem is pertinent in electoral, financial, and bioinformatics contexts, where features denote assessments in terms of counts, ratings, or rankings. Four algorithms for feature selection in such rank-order spaces are presented. Two are information-theoretic, and two are order-theoretic. These algorithms are empirically evaluated against both synthetic and real world datasets. The main results of this work are (i) characterization of relationships and equivalences between different feature selection strategies with respect to the spaces in which they operate, and the distributions they seek to approximate; (ii) identification of computationally simple and efficient strategies that perform surprisingly well; and (iii) a feasibility study of order-theoretic feature selection for large scale datasets.

In the remaining sections, some definitions are provided, four algorithms for feature selection are introduced, and theoretical intuitions as well as experimental results on synthetic and real datasets are presented.

5.1 Definitions

Definition 1. *Let* $\mathbf{F} = \{F_1, F_2, \ldots, F_n\}$ *be a set of* features*, and let* D_i *be the domain of feature* F_i*. Let* $\mathbf{D} = D_1 \times D_2 \times \cdots \times D_n$ *be the cartesian product of*

the feature domains. A feature instance *is a tuple* $\mathbf{f} = (f_1, f_2, \ldots, f_n) \in \mathbf{D}$. *Let* $C = \{C_1, C_2, \ldots, C_n\}$ *be a set of* classes. *A* dataset T *is a nonempty multiset of pairs* $(\mathbf{f}, c)$, *where* $\mathbf{f} \in \mathbf{D}$ *and* $c \in C$. *Let* $|T|$ *be the multiset cardinality of* T. *The probability distribution* P_T *on* $\mathbf{D} \times C$ *is:*

$$P_T(\mathbf{f}, c) \;=\; \frac{|\{(\mathbf{f}, c) \in T\}|}{|T|}.$$

If feature instance $\mathbf{f}$ *occurs in at least one pair of* T, *then the conditional probability distribution*

$$P_T(c \mid \mathbf{f}) \;=\; \frac{P_T(\mathbf{f}, c)}{P_T(\mathbf{f})}$$

is defined, for all $c \in C$, *where*

$$P_T(\mathbf{f}) \;=\; \frac{|\{\mathbf{f} \in T\}|}{|T|}.$$

Numerous criteria can be applied in a reduction of $\mathbf{F}$ to a subset $\mathbf{F}' \subset \mathbf{F}$, including accuracy of predictive modeling, description length for learned mappings, or preservation of as much of the relationship between class distributions and features as possible. This latter criterion is the motivation for the classic work of Koller and Sahami [29]. For $\mathbf{f} \in \mathbf{F}$, let $\mathbf{f}_{\mathbf{F}'}$ be the projection of $\mathbf{f}$ onto the features in $\mathbf{F}'$. For a dataset T, let $T_{\mathbf{F}'} = \{\mathbf{f}_{\mathbf{F}'} \mid \mathbf{f} \in \mathbf{F}\}$ be the projection of T using the $\mathbf{F}'$ feature set. The goal advanced by Koller and Sahami is to approximate $P(c \mid \mathbf{f})$ with $P(c \mid \mathbf{f}_{\mathbf{F}'})$, for all feature instances $\mathbf{f}$ that occur in some pair of T. A popular approach to characterizing the difference between two distributions is the (non-symmetric) KL-divergence [13].

Definition 2. *Let* P *and* Q *be probability distributions on the sample space* X. *The KL-divergence between* P *and* Q *is* $KL(P, Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}$, *where* $0 \log \frac{0}{Q(x)} = 0$ *and* $P(x) \log \frac{P(x)}{0} = \infty$ *whenever* $P(x) > 0$.

Typically, P is the true distribution, and Q the approximation; $KL(P, Q)$ captures the number of extra bits required if we encoded data arising from the distribution P using a code designed using distribution Q. Koller and Sahami use KL-divergence to define two feature subset divergence quantities, $\delta_{\mathbf{F}'}$ and $\Delta_{\mathbf{F}'}$.

Definition 3. *(Koller and Sahami) Let* $\mathbf{F}' \subset \mathbf{F}$*. Let* $T \subset \mathbf{D} \times C$ *be a dataset. For each feature instance* $\mathbf{f}$ *in* T*, define its divergence to be*

$$\delta_{\mathbf{F}'}(\mathbf{f}) \quad = \quad KL(P(\cdot \mid \mathbf{f}), P(\cdot \mid \mathbf{f}_{\mathbf{F}'})).$$

Define the feature subset divergence *of* $\mathbf{F}'$ *to be*

$$\Delta_{\mathbf{F}'} \quad = \quad \sum_{\mathbf{f}} P(\mathbf{f}) \delta_{\mathbf{F}'}(\mathbf{f}),$$

where $P(\mathbf{f})$ *is taken from the distribution of feature instances in* T*.*

Two ways to utilize the feature subset divergence are (1) to define a divergence threshold and seek a smallest subset $\mathbf{F}'$ such that $\Delta_{\mathbf{F}'}$ is at most that threshold; and (2) to seek among all $\mathbf{F}' \subset \mathbf{F}$ of a fixed size one that minimizes $\Delta_{\mathbf{F}'}$.

For simplicity in incorporating orders into feature space, we assume that all feature domains are identical and that a total order is defined on that single domain. Any feature instance $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ defines an order on the features in F by the rule $F_i < F_j$ if $f_i < f_j$. In this case, we can recast the given dataset into one whose features are boolean values that capture the relative order between pairs of feature values of an instance.

Definition 4. *Let* $\mathbf{F} = \{F_1, F_2, \ldots, F_n\}$ *be a feature set, where* D *is the common domain of* F_i*. Define the* boolean order feature set *for* $\mathbf{F}$ *to be*

$$\mathbf{B} \quad = \quad \{B_{i,j} \mid 1 \leq i < j \leq n\},$$

where the domain of each $B_{i,j}$ *is* $\{true, false\}$*. Let* $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ *be a feature instance. Then the boolean order features for* $\mathbf{f}$ *have values given by*

$$b_{i,j} \quad = \quad \begin{cases} true & \text{if } f_i < f_j; \\ false & \text{otherwise.} \end{cases}$$

There are $\binom{n}{2}$ *boolean order features. The vector comprised of the boolean order feature values* $b_{i,j}$ *of* $\mathbf{f}$ *is referred to as* $\mathbf{b}$*. If* $T \subset \mathbf{D} \times C$ *is a dataset, then the corresponding* boolean order dataset *is the multiset* $T^B = \{(\mathbf{b}, c) \mid (\mathbf{f}, c) \in T\}$*.*

Boolean order datasets suffer greater space complexity than other datasets due to the $\binom{n}{2}$ boolean order features. This drawback can be ameliorated with the use of ranks.

Definition 5. *Let* $\mathbf{F} = \{F_1, F_2, \ldots, F_n\}$ *be a feature set, where* D *is the common domain of* F_i*. Define the* rank-order feature set *for* $\mathbf{F}$ *to be*

$$\mathbf{R} \;=\; \{R_1, R_2, \ldots, R_n\},$$

where the domain of each R_i *is* $\{1, 2, \ldots, n\}$*. Let* $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ *be a feature instance, and let* $\pi_{\mathbf{f}} : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ *be a permutation that sorts* $\mathbf{f}$ *into non-decreasing order. (If two features have the same value, then choose* $\pi_{\mathbf{f}}$ *arbitrarily from the permutations that satisfy the above condition.) Then the rank-order features for* $\mathbf{f}$ *have values given by*

$$r_i \;=\; \pi_{\mathbf{f}}(i).$$

The vector comprised of the rank-order feature values r_i *of* $\mathbf{f}$ *is referred to as* $\mathbf{r}$ (note that $\mathbf{r}$ is also a permutation of $\{1, 2, \ldots, n\}$)*. If* $T \subset \mathbf{D} \times C$ *is a dataset, then the corresponding* rank dataset *is the multiset* $T^R = \{(\mathbf{r}, c) \mid (\mathbf{f}, c) \in T\}$*.*

Notice that the transformation to boolean order features provided in Definition 4 applies even if we begin with a rank-order dataset. Figure 5.1 illustrates Definitions 1, 4, and 5. Difficulties arise in Definitions 4 and 5 if any feature instance contains two identical feature values, resulting in rank ambiguity. However, rank ambiguity can be addressed in boolean order datasets by setting the domain to $\{\text{true}, \text{false}, =\}$, instead of just $\{\text{true}, \text{false}\}$. In fact, setting the domain to $\{\text{true}, \text{false}, =, \text{unknown}\}$ allows us to address partial orders in general. For rank-ordered datasets, there is a corresponding method to address rank ambiguity; addressing partial orders, however, is not straightforward in this context.

For simplicity, we assume total orders and that rank ambiguity does not occur. However, with additional labor, it is possible to relax the rank-ambiguity restriction and apply most of the statements and conclusions in this paper in the relaxed case; exceptions will be noted when appropriate. Since there is no rank ambiguity in $\mathbf{f}$, the permutation $\pi_{\mathbf{f}}$ is uniquely defined; call it the *feature order* of $\mathbf{f}$.

The update required if a feature is removed in the context of boolean order or rank-order datasets is more complicated than in the unranked context, as the projection onto a smaller feature space requires additional effort. If feature F_i is removed in a boolean order context having n original features and $\binom{n}{2}$ boolean order features, then there are $n - 1$ boolean order features of the form $B_{i,j}$ or $B_{j,i}$ to eliminate, and there are other boolean order features to re-index. If feature F_i is removed in a rank-order context where there are n original features, then rank-order feature R_i is eliminated, and the remaining $n - 1$ rank-order features is re-indexed and

F_1	F_2	F_3	F_4	class
20	40	65	33	a
20	40	65	33	a
50	25	55	99	b
88	76	10	60	a

$B_{1,2}$	$B_{1,3}$	$B_{1,4}$	$B_{2,3}$	$B_{2,4}$	$B_{3,4}$	class
true	true	true	true	false	false	a
true	true	true	true	false	false	a
false	true	true	true	true	true	b
false	false	false	false	false	true	a

R_1	R_2	R_3	R_4	class
1	3	4	2	a
1	3	4	2	a
3	1	2	4	b
4	3	1	2	a

Figure 5.1: Dataset T (top table) with its corresponding boolean order dataset T^B and rank-order dataset T^R (bottom tables). Note that T is a *multiset*, due to the first and second feature instances.

their rank values updated, For example, projecting the rank-order feature instance $\mathbf{r} = (3, 1, 4, 2)$ onto the first three (rank) features results in the rank-order feature instance $(2, 1, 3)$, not $(3, 1, 4)$.

Let $T \subset \mathbf{D} \times C$ be a dataset, and let T^R be the corresponding rank-order dataset. Let $(\mathbf{f}, c) \in T$ be a feature instance, and let $(\mathbf{r}, c) \in T^R$ be its corresponding rank-order feature instance. It is possible that $P(\cdot \mid \mathbf{f}) \neq P(\cdot \mid \mathbf{r})$. This is because information (specifically, information that is not order-specific) is lost in the transformation from T to T^R. On the other hand, order-specific information is now available in T^R, presumably not considered in the original dataset. We provide some results illustrating this point in Section 5.2.

Since, in the context of a rank-order dataset, an instance (as well as its implied feature order) is a permutation, we require methods to measure differences between permutations. We present two established approaches for defining a distance function between two permutations, one that works with (rank) instances, and another that works with feature orders. The first distance function is Spearman's ρ [48], which maps a permutation of length n to an n-dimensional vector and utilizes Euclidean distance between vectors.

Definition 6. *(Spearman's distance)*

$$\rho(\pi_i, \pi_j) = \sqrt{\sum_{k=1}^{n} (\pi_i(k) - \pi_j(k))^2}$$

Let P be a set of permutations. The center $\mathrm{ctr}_\rho(P)$ *of P is a permutation π_c (not necessarily in P) that minimizes*

$$\sum_{\pi \in P} (\rho(\pi, \pi_c))^2.$$

Algorithmically, we can compute the permutation $\mathrm{ctr}_\rho(P)$ by summing the ranks in each position of all permutations and then derive $\mathrm{ctr}_\rho(P)$ from the order of the resulting sums (if there are duplicate sums, ties are broken arbitrarily).

In the context of feature selection, every class C_j for which $P_j = \{\mathbf{r} \mid (\mathbf{r}, C_j) \in T^R\}$ is nonempty, $\mathrm{ctr}_\rho(P_j)$ is a permutation at the center of a smallest hypersphere containing all the permutations of P_j. If C_i and C_j are distinct classes, then $\mathrm{ctr}_\rho(P_i)$ and $\mathrm{ctr}_\rho(P_j)$ are representatives of the two classes that can be used to define a distance between the two classes. A feature removal algorithm might choose to remove a feature that yields distances that are closest to the original distances.

A second distance function between permutations is based on swaps.

$$\begin{array}{rll} \pi_i = & F_3\ F_2\ F_4\ F_5\ F_1\ F_6 & \\ & F_3\ F_2\ F_5\ F_4\ F_1\ F_6 & \text{swap}(F_4,F_5) \\ & F_3\ F_5\ F_2\ F_4\ F_1\ F_6 & \text{swap}(F_2,F_5) \\ & F_3\ F_5\ F_2\ F_1\ F_4\ F_6 & \text{swap}(F_4,F_1) \\ \pi_j = & F_3\ F_5\ F_2\ F_1\ F_4\ F_6 & \end{array}$$

Figure 5.2: Three interchanges are required to transform π_i to π_j.

Definition 7. *(Kendall's distance) A permutation* $\pi : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ *is an* adjacent swap *if there exists an* i*, where* $1 \leq i \leq n-1$*, such that*

$$\pi(j) \quad = \quad \begin{cases} j+1 & \text{if } j = i; \\ j-1 & \text{if } j = i+1; \\ j & \text{otherwise.} \end{cases}$$

The function $\tau(\pi, \pi')$ *is the minimum number of adjacent swaps required to transform* π *to* π'*.*

One transformation of $\pi_i = F_3\ F_2\ F_4\ F_5\ F_1\ F_6$ to $\pi_j = F_3\ F_5\ F_2\ F_1\ F_4\ F_6$ with three adjacent swaps is illustrated in Figure 5.2, showing the three intermediate permutations. The minimum number of swaps of adjacent pairs is exactly the same as Knuth's inversion count [28]. If $\pi_i(x) < \pi_i(y)$ and $\pi_j(x) > \pi_j(y)$ then the pair (x, y) is an inversion, where $\pi_i(x)$ is the x'th element of permutation π_i.

Instead of using centers based on τ (the decision problem for τ-centers being NP-complete [19]), we propose a different notion for feature removal that uses inversions. Consider the feature orders of two instances in rank-order space. In a transformation from one permutation to the other using inversions, each inversion involves two features. Let $\text{inv}(R_i, \pi_a, \pi_b)$ represent the number of inversions that a feature R_i is involved in, when going from π_a to π_b (for the example in Figure 2.1, $\text{inv}(F_2, \pi_i, \pi_j) = 1$ while $\text{inv}(F_5, \pi_i, \pi_j) = 2$).

Definition 8. *Let* I *be a multiset of feature orders. The* spoiler count $\text{sp}(R_i, I)$ *of feature* R_i *with respect to* I *is*

$$\text{sp}(R_i, I) = \sum_{\pi_a, \pi_b \in I} \text{inv}(R_i, \pi_a, \pi_b).$$

Suppose I is the multiset of feature orders of instances associated with a single class in T^R. Since the instances belong to the same class and order is presumably

indicative of class membership, these permutations ought to be similar in order. The feature that has the highest spoiler count contributes most to the differences in order and is therefore a good candidate for removal.

For example, if π_1, π_2, π_3 belong to class I_a and the spoiler count for F_7 when comparing π_1 vs. π_2 is 5, for π_1 vs. π_3 it is 3, and for π_2 vs. π_3 the spoiler count is 7, then the spoiler count for F_7 with respect to I_a is 15.

Definition 9. *For each class $c \in C$, let $T^R(c)$ represent the set of feature orders of instances associated to c in T^R. The* total spoiler count $\text{tsp}(R_i)$ *of rank-order feature R_i is*

$$\text{tsp}(R_i) = \sum_{c \in C} \text{sp}(R_i, T^R(c)).$$

For example, given a set of feature orders with 4 classes, I_a, I_b, I_c, and I_d. If the spoiler count of R_7 with respect to I_a is 15, with respect to I_b is 12, with respect to I_c is 26, and with respect to I_d is 19, then the total spoiler count for R_7 is 72. A feature selection algorithm might choose to remove the feature that has the highest total spoiler count.

5.2 Comparing Feature Spaces

In this section, we investigate relationships among feature spaces, boolean order spaces, and rank-order spaces. We first identify examples where order matters more than value and vice-versa.

Let F_k be a feature in $\mathbf{F}$, R_k the corresponding feature in $\mathbf{R}$, $\mathbf{F}' = \mathbf{F} - \{F_k\}$, and $\mathbf{R}' = \mathbf{R} - \{R_k\}$. We say that $\mathbf{F}_k$ is a *removable* feature if $\Delta_{\mathbf{F}'} = 0$. We now examine the relationship between $\Delta_{\mathbf{F}'}$ and $\Delta_{\mathbf{R}'}$. The next two conjectures and their counterexamples illustrate that the relationship is not simple and easily exploitable.

Conjecture 1. *Let T be a dataset with feature set $\mathbf{F}$ and suppose F_k is a feature such that the feature set divergence $\Delta_{\mathbf{F}-\{F_k\}}$ is 0. Then, $\Delta_{\mathbf{R}-\{R_k\}} = 0$ in T^R,*

Counterexample: Figure 5.3 provides an example of a dataset T and the corresponding rank-order dataset T^R. Here, F_1 is a removable feature in dataset T (values assigned to F_1 are the same for all instances). This means $\Delta_{\mathbf{F}-\{F_k\}} = 0$. However, $\Delta_{\mathbf{R}-\{R_1\}} \approx 0.824$ in T^R, because there is at least one instance $\mathbf{r}$ (such as $\mathbf{r} = (3, 1, 2)$) such that $\delta_{\mathbf{R}'}(\mathbf{r}) \approx 0.275$.

F_1	F_2	F_3	class
3	4	5	a
3	4	1	b
3	1	2	c
3	2	5	d

R_1	R_2	R_3	class
1	2	3	a
2	3	1	b
3	1	2	c
2	1	3	d

Figure 5.3: Datasets T (top) and T^R (bottom) where a removable feature in T does not apply in T^R.

Conjecture 2. *Let T be a dataset with feature set $\mathbf{F}$, and let T^R with feature set $\mathbf{R}$ be its corresponding rank-order dataset. Suppose R_k is a feature such that $\Delta_{\mathbf{R}'} = 0$. Then, $\Delta_{\mathbf{F}'} = 0$ in T.*

COUNTEREXAMPLE: Figure 5.4 provides an example of a dataset T and corresponding rank-order dataset T^R. Here, R_1 can be removed (as can any single feature) while retaining the same capacity to classify so that $\Delta_{\mathbf{R}'} = 0$ in T^R, where $\mathbf{R}' = \mathbf{R} - \{R_1\}$. However, $\Delta_{\mathbf{F}'} \neq 0$ for $\mathbf{F}' = \mathbf{F} - \{F_1\}$ since F_1 is in fact the feature that distinguishes the two instances in the dataset ($\delta_{\mathbf{F}'}(\mathbf{f}) \neq 0$ for both instances).

On the other hand, the following result demonstrates that rank-order datasets and boolean order datasets contain the same order-theoretic information with respect to feature selection.

Lemma 1. *Let T^R be a rank-order dataset with feature set $\mathbf{R}$, and let T^B with feature set $\mathbf{B}$ be its corresponding boolean order dataset. Furthermore, let $\mathbf{R}' \subset \mathbf{R}$. Define $\mathbf{B}' \subset \mathbf{B}$ to be the set of all features $B_{i,j}$ such that $R_i, R_j \in \mathbf{R}'$. Then, $\Delta_{\mathbf{R}'} = \Delta_{\mathbf{B}'}$.*

Proof. From Definition 4, it suffices to show that $P(\mathbf{r}) = P(\mathbf{b})$ and $\delta_{\mathbf{R}'}(\mathbf{r}) = \delta_{\mathbf{B}'}(\mathbf{b})$, for all instances $\mathbf{r}$. The equality $P(\mathbf{r}) = P(\mathbf{b})$ follows directly from how the boolean

F_1	F_2	F_3	class
1	4	5	a
2	4	5	b

R_1	R_2	R_3	class
1	2	3	a
1	2	3	b

Figure 5.4: Datasets T (top) and T^R (bottom) where a removable feature in T^R does not apply in T.

order set was constructed since there is a one-to-one correspondence between rank-order instances ($\mathbf{r}$) and boolean order instances ($\mathbf{b}$). We obtain that

$$\delta_{\mathbf{R}'}(\mathbf{r}) = KL(P(C \mid \mathbf{r}), P(C \mid \mathbf{r}_{\mathbf{R}'})),$$

and

$$\delta_{\mathbf{B}'}(\mathbf{b}) = KL(P(C \mid \mathbf{b}), P(C \mid \mathbf{b}_{\mathbf{B}'}))$$

are equal by the observation that the projections performed on each of the datasets are equivalent. For a given $\mathbf{r}$, $P(C \mid \mathbf{r})$ and $P(C \mid \mathbf{b})$ obviously yield the same distributions, again because of the one-to-one transformation. For the distributions $P(C \mid \mathbf{r}_{\mathbf{R}'})$ and $P(C \mid \mathbf{b}_{\mathbf{B}'})$, on the other hand, we note that a projection in rank-order space preserves the relative order of the features even with the (possible) update in rank values. This in turn corresponds to the boolean order features that are projected in boolean order space. Thus, $P(C \mid \mathbf{r}_{\mathbf{R}'}) = P(C \mid \mathbf{b}_{\mathbf{B}'})$, and the result follows. □

Lemma 1 suggests that it is sufficient to consider selection strategies on rank-order datasets and that analogous strategies using boolean order datasets will yield the same results.

5.3 Feature Selection Strategies

We present four feature selection strategies (two taking an information-theoretic approach and two motivated by the discrete mathematical concepts introduced in Section 5.1), all of which follow the standard backward stepwise selection framework [21].

$\mathbf{F'}_0 \leftarrow \mathbf{F}$; $i \leftarrow 0$
while $cond(\mathbf{F'}_i)$
    $F_k \leftarrow h(\mathbf{F'}_i)$
    $\mathbf{F'}_{i+1} \leftarrow \mathbf{F'}_i - \{F_k\}$
    $i \leftarrow i + 1$
end while
return $\mathbf{F'}_i$

In this meta-algorithm, the boolean function $cond(\mathbf{F'}_i)$ either monitors subset size or subset divergence. The function $h(\mathbf{F'}_i)$ is the feature selection function for this selection strategy. This function returns a feature from $\mathbf{F'}_i$ in regular feature space but uses rank order space in its selection process. Recall that $\mathbf{F}$, F_k, and $\mathbf{F'}_i$ correspond to $\mathbf{R}$, R_k, and $\mathbf{R'}_i$ in rank-order space, as these terms may be used in the definition of $h(\mathbf{F'}_i)$. The four selection strategies are Greedy KL, KS, CDV, and Spoilers.

Greedy KL Use rank-order space and greedily choose the feature that yields the minimum feature subset divergence when compared against $\mathbf{R}$. That is, choose $h(\mathbf{F}_i) = F_k$ that minimizes $\Delta_{\mathbf{R}_i - \{R_k\}}$ with respect to $\mathbf{R}$.

KS Adapt the Koller-Sahami algorithm [29] for use in rank-order space. First, find (approximate) Markov blankets for all features, in the Bayesian network of rank-order features (and classes) implied by T^R. A Markov blanket for a set of features $\mathbf{F'}$ is another set of features $\mathbf{G}$ whose values, if known, render $\mathbf{F'}$ independent of all others (i. e., $\mathbf{F} - \mathbf{F'} = \mathbf{G}$). This term arises from the graphical models literature where a network encodes conditional independencies, and random variables satisfying the above definition form a 'blanket' around the given set of features. In the Koller-Sahami approach, we remove the feature F_k whose Markov blanket M_k (in rank-order space) yields the minimum feature subset divergence when compared against $M_k \cup R_k$. That is, $h(\mathbf{F}_i) = F_k$ that minimizes Δ_{M_k} with respect to $M_k \cup R_k$.

To approximate the computation of a Markov blanket M_k for a feature R_k, of a given size κ, the approach suggested in [29] is to pick the top κ features that, singly, yield minimum feature subset divergence when pair-wise unioned with R_k.

CDV (Center Distance Vector) Use rank-order space and, for each class, compute centers (ctr_ρ) of all permutations for that class given in the dataset. For each

Strategy	Metric Utilized	Time Complexity Per Iteration
Greedy KL	Low feature subset divergence	$\mathbf{O(n^2m)}$
KS	Availability of Markov blanket	$\mathbf{O(n^4cm}\kappa)$
CDV	Low Spearman's ρ distances	$\mathbf{O(n^2(m+c^2))}$
Spoilers	High spoiler count	$\mathbf{O((n}\log\mathbf{n)m^2)}$

Figure 5.5: Details of the four feature selection strategies considered in this paper.

pair of classes, compute the ρ distances between the centers, obtaining a $\binom{|C|}{2}$-vector. For each feature, remove it, and recompute this vector of ρ distances. $h(\mathbf{F'}_i) = F_k$ where R_k is the feature in $\mathbf{R'}_i$ that yields the minimum Euclidean distance between the recomputed vector and the original vector (computed from $\mathbf{R}$).

Spoilers Use rank-order space (feature orders) and remove the feature with the highest spoiler count. That is, $h(\mathbf{F'}_i) = F_k$ that maximizes $\text{tsp}(R_k)$.

As Figure 5.5 shows, the four algorithms considered here fall under a few broad categories. The table also identifies the time complexities per step for each of the four algorithms, in terms of $\mathbf{n}$, $\mathbf{c}$, and $\mathbf{m}$, which represent the number of features, classes, and instances, respectively.

The KL algorithm is dominated by the computation of feature subset divergence (Δ). Computing this quantity entails projecting each instance in a dataset and grouping identical instances for δ and Pr computation. The projection operation takes $\mathbf{O(n)}$ time and is carried out for all instances in each class. The computation of Δ thus takes $\mathbf{O(nm)}$ time, since the total number of instances is $\mathbf{m}$. Finally, since Δ is computed for each feature, the KL algorithm takes $\mathbf{O(n^2m)}$ time per iteration.

The KS algorithm takes $\mathbf{O(n^4cm}\kappa)$ time per iteration, where κ is the predetermined blanket size. This is different from the standard Koller and Sahami algorithm since the table for the cross-entropy of the class distribution for the given pairs of features (γ_{ij}) needs to be recomputed at each step and not just once at the beginning.

Computing ctr_ρ entails scanning each rank in each instance ($\mathbf{O(nm)}$) and computing the distances between each of the $\binom{\mathbf{c}}{2}$ pairs of centers ($\mathbf{O(c^2n)}$). Each of these computations is done $\mathbf{O(n)}$ times. Therefore the time complexity is $\mathbf{O(n^2(m+c^2))}$ per iteration for the CDV algorithm. Kendall [26] shows how to compute ctr_ρ in

$\mathbf{O(nm)}$ time. Essentially, the center rank is the rank of the features as ordered by the sum of their ranks for each feature.

Finally, using the spoiler count requires processing all pairs of instances in a class. There are $\mathbf{O(m^2)}$ such pairs. Determining spoiler counts for each of these pairs takes $\mathbf{O(n \log n)}$ time if there are no ties in the rankings; otherwise the time complexity is $\mathbf{O(n^2)}$. The Spoilers algorithm therefore runs in $\mathbf{O((n \log n)m^2)}$ time per iteration or $\mathbf{O(n^2)}$ time per iteration, respectively. The spoiler count between two instances can be computed in $\mathbf{O(n \log n)}$ time through a modification of merge sort. Essentially a merge sort is performed to sort one permutation to be in the same order as the other. During the merge phase, as an item is merged from the right hand list, its spoiler count is incremented by the number of items remaining in the left hand list. As an item is merged from the left hand list, its spoiler count is incremented by the number of items from the right hand list that have preceded it.

5.4 Datasets

The four heuristics described in Section 5.3 were tested against both synthetic and real world datasets. There are numerous ways to generate synthetic datasets, two of which were chosen as representative of the range of possibilities. The first method uses a minimal, known exact answer for each class as a seed for generating the rest of the data. The second method generates random orders for each class that are concordant[1] without knowing the exact minimal answer *a priori*.

The first set of synthetic datasets is generated as follows. Select $\mathbf{c}$, the number of classes in the dataset, and build a dataset starting with $\mathbf{n_o}$ features where $\mathbf{n_o}$ is the smallest integer such that $\mathbf{n_o}! \geq \mathbf{c}$. Generate $\mathbf{c}$ different feature orders, one order for each class. Each of these orders determines an $(\mathbf{r}, c)$ instance-class pair. Select i, and generate i identical instances per class so that $\mathbf{m} = i\mathbf{c}$ is the total number of pairs in the dataset. Select e and r, the number of extraneous and redundant features to be added to the dataset, respectively. In this algorithm, $\mathbf{c}$, i, e, and r are tunable parameters chosen by the user.

An extraneous feature is incorporated into the dataset by considering each instance separately and by randomly inserting the new feature into the existing order implied in the instance. A redundant feature is incorporated into the dataset by randomly selecting an existing feature and then for each instance, inserting the new feature

[1] As determined by a user selected minimum concordance score

Class	Orders						Ranks a	b	c	d	e	f
C_1	a	f	e	c	b	d	1	5	4	6	3	2
C_1	a	f	b	c	e	d	1	3	4	6	5	2
C_1	a	f	e	b	d	c	1	4	6	5	3	2
C_2	f	d	c	b	e	a	6	4	3	5	2	1
C_2	b	f	c	e	d	a	6	1	3	5	4	2
C_2	f	d	c	e	a	b	5	6	3	2	4	1
C_3	c	b	e	f	d	a	6	2	1	5	3	4
C_3	e	c	f	d	b	a	6	5	2	4	1	3
C_3	e	c	f	d	b	a	6	5	2	4	1	3

Figure 5.6: Generated dataset, with $\mathbf{n_0}$=(a,f,e), e=(d,b), and r=(c) where c is redundant with e.

either to the left or the right of the selected feature. This way, the new feature and the selected feature have exactly the same relative-order relationships with the other features.

In effect, a dataset with $\mathbf{n} = \mathbf{n_0} + e + r$ features and $\mathbf{m} = i\mathbf{c}$ instances are generated given $\mathbf{c}$, i, e and r. Figure 5.6 provides an example of a generated dataset for $\mathbf{c} = 3$, $i = 3$, $e = 2$, and $r = 1$. The example starts with $\mathbf{n_0} = 3$ features, since $3! \geq 4$ and 3 is the smallest such value that satisfies the expression.

The second method of generating datasets uses the idea of concordance defined in Section 2.5. Figure 2.7 on page 15 shows how to compute the τ-concordance of a set of rank orders. There is a corresponding method for computing the ρ-concordance of a set of rank orders, as shown in Figure 2.6. This measures the proximity of each permutation to every other permutation within the set, using ρ as a distance coefficient. The range of τ-concordance is from 0 to 1. If the concordance is 1, then every permutation has the same order; if 0, they are as different from each other as they can possibly be. The complexity of the algorithm is $\mathbf{O(n^2m)}$.

This measure of concordance is used to construct synthetic datasets to test the feature selection strategies. First, a random set of ranks is generated containing c instances. The concordance of this set must be below some threshold $C_{\max}$. Each instance of this set forms the basis of a class. New randomly generated instances are added to each class as long as the concordance of that class remains above some threshold $C_{\min}$.

The real world datasets we use come from large scale gene expression data derived from DNA microarray experiments. Here, the instances are the microarray experiments, each feature corresponds to a given gene, and the feature values denote expression levels. Classes denote some classification of the experimental conditions. We explore the possibility of considering the relative order between gene expression levels instead of their absolute values. This is because different experiments have different basal levels of expression and hence identifying which genes are more highly expressed than others reveals greater insight into variabilities and similarities across experiments.

The first dataset from Arbeitman, et al. [1] examines the life cycle of *Drosophila melanogaster* using 74 microarrays accessing different time points. The expression levels of 3107 genes were examined during the embryotic, larval, pupal, adult male and adult female stages. For our purposes, the classes are the life cycle stages. The intent is to identify those genes that are most important to the life cycle.

The second dataset is available from the Whitehead Institute Center for Genome Research[2]. This dataset is used to examine several cancerous conditions of the human blood production system (leukemia). This data was also used by Zhang, Yu, and Singer [55] to test their prediction method, which uses a deterministic procedure to form forests of classification trees. There are 72 instances of 7,129 genes using the corrected raw fluorescence for the expression levels, 25 of which have acute myeloid leukemia (AML), 38 instances with B cell acute lymphoblastic leukemia (ALL), and 9 instances with T cell acute lymphoblastic leukemia (ALL). The dataset was tested in two ways, once with the 2 classes (ALL and AML), and again with all three classes (AML, B-ALL, T-ALL).

5.5 Experimental Results

The synthetic datasets in our experiments have $\mathbf{c} = 7$ ($\mathbf{n_0} = 4$) and $i = 20$. We used different combinations of values for e and r, where $e \in \{0, 2, 4, 6, 8\}$ and $r \in \{0, 2, 4, 6, 8\}$. Five samples for each combination were generated. Each of the four algorithms analyzed the different datasets, and feature subset divergence was measured at each iteration. Figure 5.7 displays plots of the extremes of the combinations. The number of features removed versus subset divergence (average across the 5 samples) for each heuristic are shown. The results suggest that the KS algorithm performs poorly with these datasets, whereas there is little differ-

[2] `http://www.broad.mit.edu/cancer`

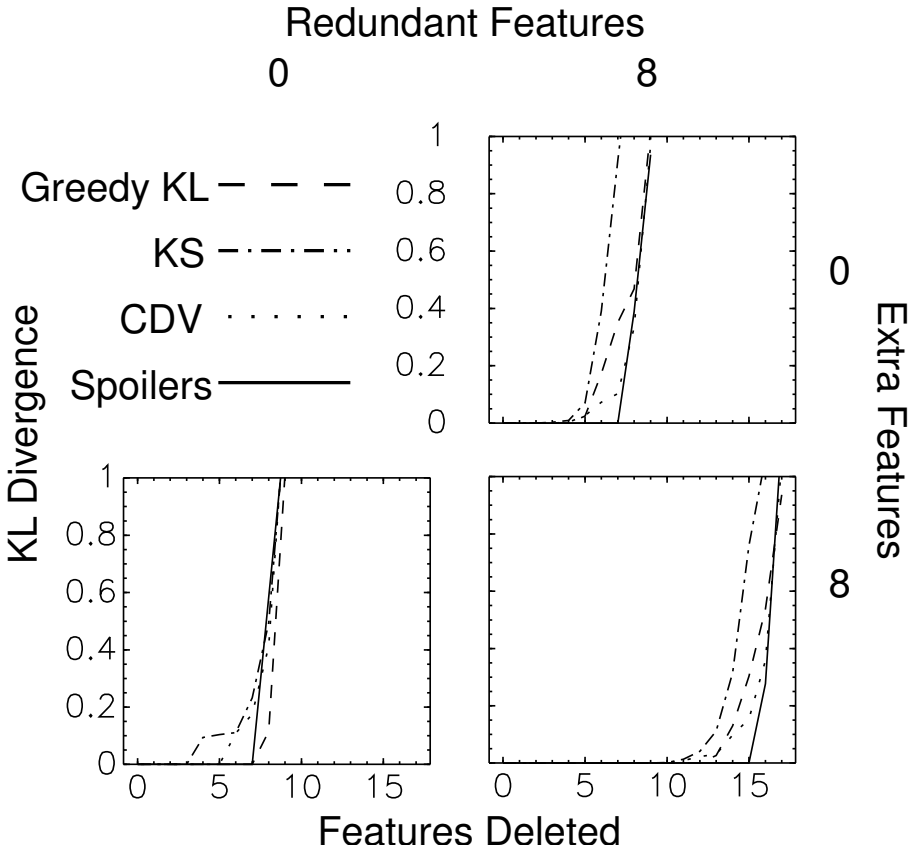


Figure 5.7: KL divergence for synthetic datasets showing the effects of redundant and/or extra features for different feature selection methods.

ence in the performance of the other three algorithms. Anecdotal evidence from our early experiments on the KS algorithm showed that, for rank-order datasets, a small value of k performed marginally better than a larger value. Therefore, for these experiments, k was arbitrarily chosen to be 5.

We also measured Naive Bayes accuracy [39] at each iteration with 5-fold cross validation. A subset of the results are presented in Figure 5.8. The Naive Bayes classification method and 5-fold cross validation were performed with the Orange data mining system [14] using the equivalent boolean order dataset. Those results also show the KS algorithm performing poorly. In addition, the charts suggest that the heuristics based on centers and spoilers perform better than the Greedy KL heuristic, particularly as the number of redundant features in the dataset increases. On the other hand, the Greedy KL heuristic performs slightly better than the other

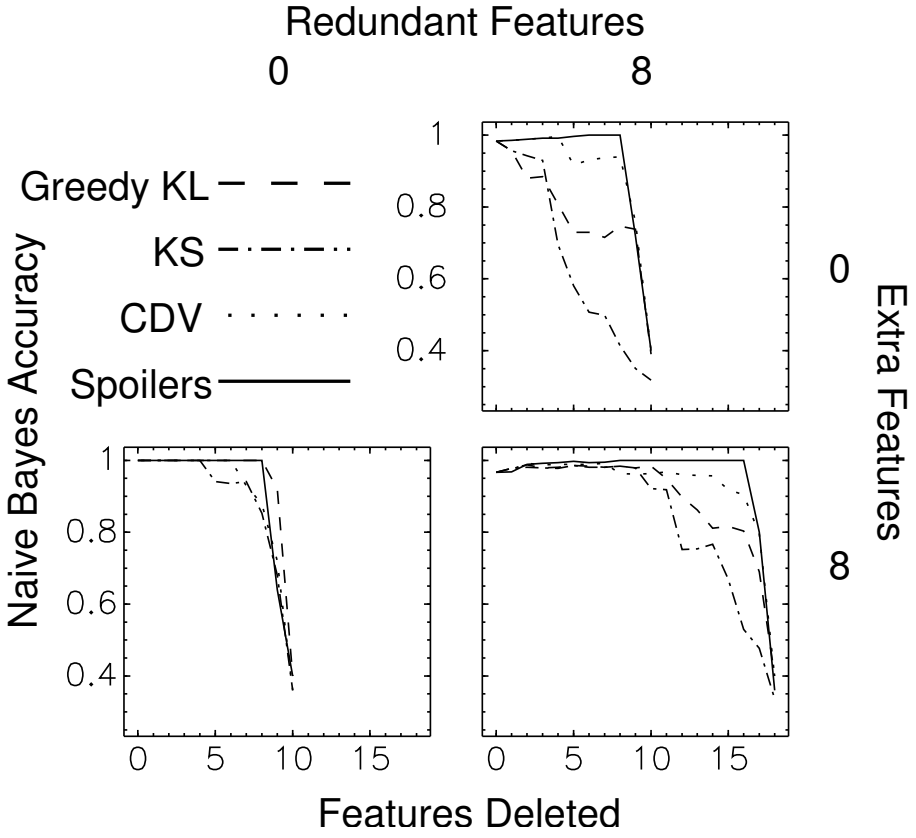


Figure 5.8: Naive Bayes five-fold cross validation accuracies for synthetic datasets showing the effects of redundant and/or extra features for different feature selection methods.

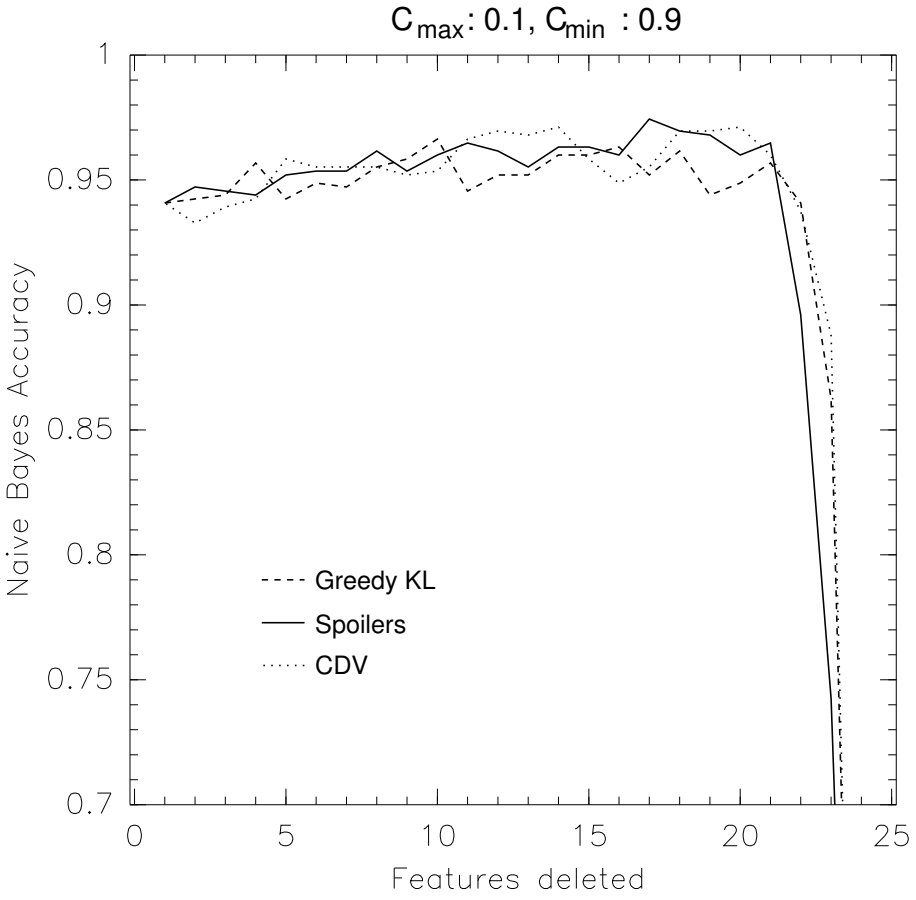


Figure 5.9: Naive Bayes five-fold cross validation accuracies for an easy synthetic dataset generated using concordance methods.

two heuristics when only extraneous features are added to the dataset.

For synthetic datasets generated using concordance methods, different combinations of $C_{\min}$ and $C_{\max}$ were tried. Figures 5.9 and 5.10 show two extremes. Both charts show the average of five samples, where $\mathbf{n} = 25$, $i = 100$, and $\mathbf{c} = 5$. Figure 5.9 has $C_{\min} = 0.1$ and $C_{\max} = 0.9$, so the classes are coherent and distinct. With these conditions, every feature selection strategy does well, since almost any feature can be removed with ill effects. Figure 5.10 has $C_{\min} = 0.9$ and $C_{\max} = 0.1$, so the classes are ill-defined and indistinct. In this case, the features need to be selected in a more intelligent manner. The Spoilers algorithm does better than CDV, which in turn, does better than Greedy KL.

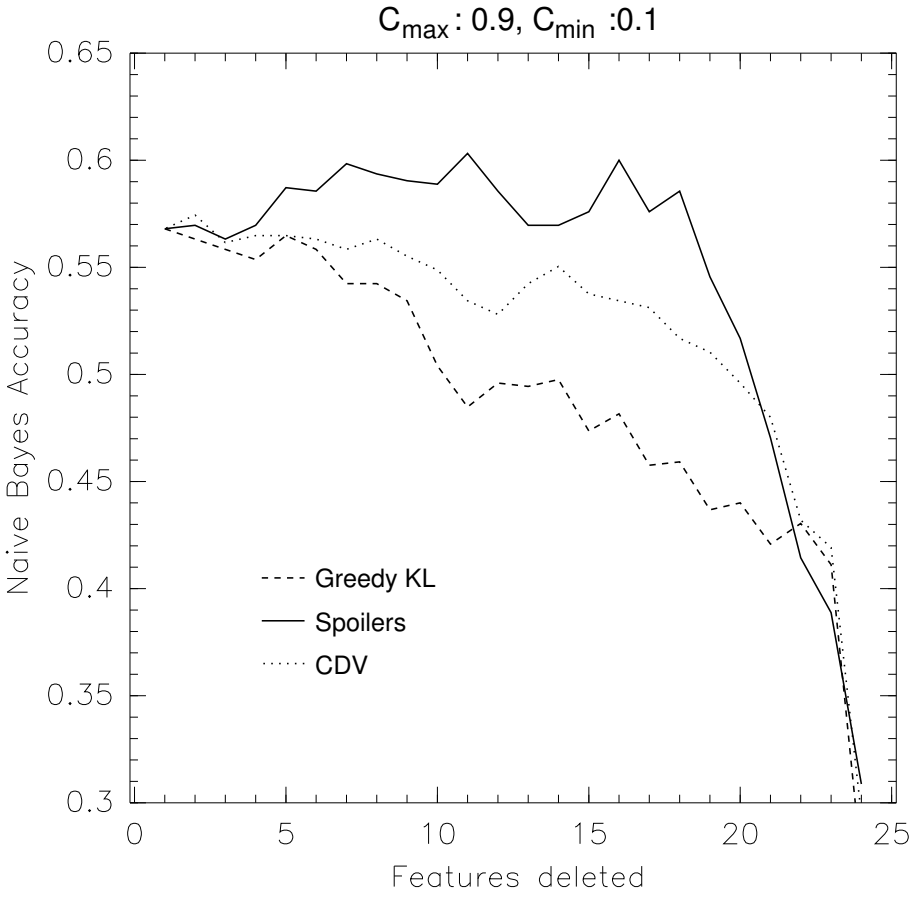


Figure 5.10: Naive Bayes five-fold cross validation accuracies for a hard synthetic dataset generated using concordance methods.

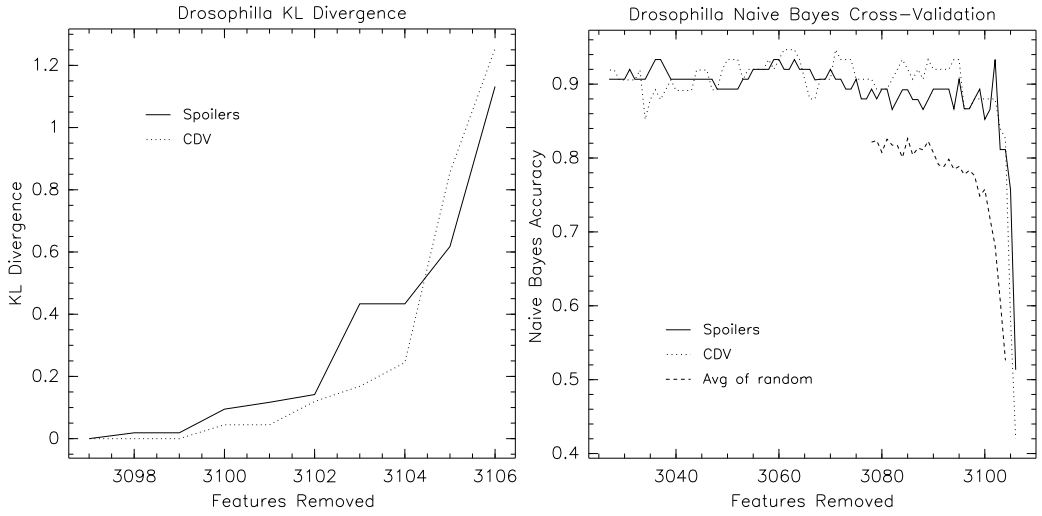


Figure 5.11: Results of two feature selection methods for the *Drosophila melanogaster* life-cycle experiments.

Figure 5.11 shows the results for the CDV algorithm and the Spoilers algorithm on the *Drosophila melanogaster* dataset. The first 3097 features removed by the Spoilers algorithm and the first 3098 features removed by the CDV algorithm maintain a KL divergence of zero. The Naive Bayes accuracies using five-fold cross validation for the last 80 features for both algorithms are approximately the same, and both do better than the average results from uniformly chosen random sets of features. Random sets of size n, where $n = (2 \ldots 30)$, were selected 100 times for each n, and their average Naive Bayes accuracy was calculated to provide the baseline comparison. The CDV algorithm found a set of 13 genes that get a 93.3% accuracy as shown in Figure 5.12, while the Spoilers algorithm found a set of 6 genes that get a 93.3% accuracy as shown in Figure 5.13.

For the cancer datasets using both two and three classes, shown in Figures 5.14 and 5.16 respectively, the CDV algorithm performed markedly better than the Spoilers algorithm. The Spoilers algorithm did no better than the average random sample. The CDV algorithm found a set of 25 genes, shown in Figure 5.15, for two classes that had a Naive Bayes five-fold cross validation accuracy of 93% and a set of 16 genes, shown in Figure 5.17, that had an accuracy of 97% for three classes. This is reflected in the KL divergences which the CDV algorithm maintains at zero longer than the Spoilers algorithm.

The results for the cancer datasets differ from Zhang et al. [55] in the gene subsets

Gene	Annotation
CG10033	foraging, cGMP-dependent protein kinase activity
CG10602	leukotriene-A4 hydrolase
CG12120	unknown function
CG12321	unknown function
CG12699	unknown function
CG14722	apoptotic protease activator activity
CG2019	peroxidase activity
CG2985	structural molecule activity involved in vitellogenesis
CG3991	tripeptidyl-peptidase II activity putatively
CG5940	cyclin-dependent protein kinase regulator activity
CG6483	serine-type endopeptidase activity
CG6933	structural constituent of peritrophic membrane (sensu Insecta)
CG7157	hormone activity involved in sperm storage

Figure 5.12: Subset of 13 genes, chosen by the CDV algorithm, that yield a 93.3% accuracy using Naive Bayes five-fold cross validation when used to predict the *Drosophila melanogaster* life-cycle.

Gene	Annotation
CG10096	unknown function
CG1140	3-oxoacid CoA-transferase activity
CG13841	unknown function
CG3121	putative microtubule binding
CG3333	putative pseudouridylate synthase activity
CG7157	hormone activity involved in sperm storage

Figure 5.13: Subset of 6 genes, chosen by the Spoilers algorithm, that yield a 93.3% accuracy using Naive Bayes five-fold cross validation when used to predict the *Drosophila melanogaster* life-cycle.

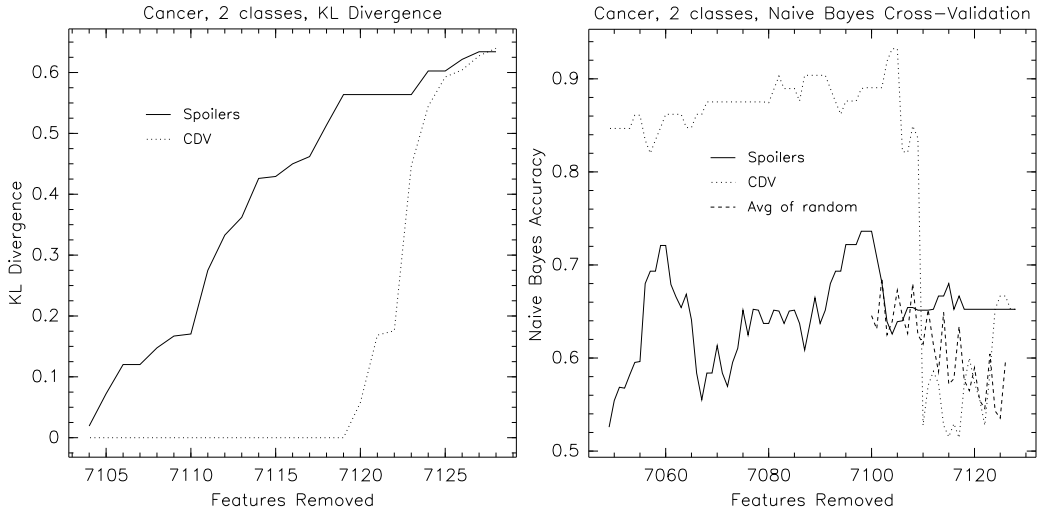


Figure 5.14: Results for two feature selection methods using the cancer dataset with two classes.

that are considered important for classification. There are several differences in the goals of the algorithms. Our method performs feature selection only, while their algorithm does both feature selection and classification. Their method is based upon the absolute values, which are subject to greater variability under different experimental conditions than comparing relative values. As Figures 5.15 and 5.17 reveal, not withstanding that they have only two genes in common, many pertinent features are picked out by the CDV algorithm (e.g., a serine kinase, a cell cycle controller, a proto-oncogene, an oncostatin, all of which are involved in cell growth and regulation and could influence the progression of cancer).

In addition to using a Naive Bayes classifier to test the feature selection strategies, a classifier based on Support Vector Machines (SVMs) first developed by Boser et al. [5] is also used. SVMs work by mapping the training data into higher dimensional space and then finding a linear separating hyperplane with a maximal margin to categorize the data. The software implementation of LibSVM created by Chang and Lin [9] was used to perform the classification. The results for both the cancer and *Drosophila melanogaster* data sets are shown in Figure 5.18. The results are obtained using the default options for LibSVM using five-fold cross validation, without any attempt at tuning the parameters for this feature selection problem. The kernel used is the radial basis function. SVM classification is performed for every 100 features removed using the original data values (not the ranks) scaled to be between 0 and 1. The results the CDV method improving the classification

Gene	Annotation
D49824	HLA-B null allele mRNA
D70830	mRNA for Doc2 beta
HT2696	cholinesterase-related cell division controller
HT2798	Serine/Threonine Kinase Z25424
L13977	prolylcarboxypeptidase mRNA
M17733	thymosin beta-4 mRNA
M24899	triiodothyronine (ear7) mRNA
M27288	oncostatin M gene, exon 3
M27891	cystatin C (CST3) gene, exon 3
S94421	TCR eta=T cell receptor eta-exon
U06155	chromosome 1q subtelomeric sequence D1S553
U09953	ribosomal protein L9 mRNA
U14969	ribosomal protein L28 mRNA
U15422	protamine 1, protamine 2, and transition protein 2 genes
U28727	pregnancy-associated plasma protein-A preproform (PAPPA) mRNA
U33447	putative G-protein-coupled receptor (GPR17) gene
U63717	osteoclast stimulating factor mRNA
U79267	clone 23840 mRNA
U89717	9-cis-retinol specific dehydrogenase mRNA
U89922	lymphotoxin beta isoform variant, alternatively spliced mRNA
X02160	mRNA for insulin receptor precursor
X73460	mRNA for ribosomal protein L3
X79234	mRNA for ribosomal protein L11
X90846	mRNA for mixed lineage kinase 2
Z48579	mRNA for disintegrin-metalloprotease (partial)

Figure 5.15: Subset of 25 genes that yield a 93% accuracy using Naive Bayes five-fold cross validation when used to predict the leukemia type (2 classes ALL and AML) of a cell.

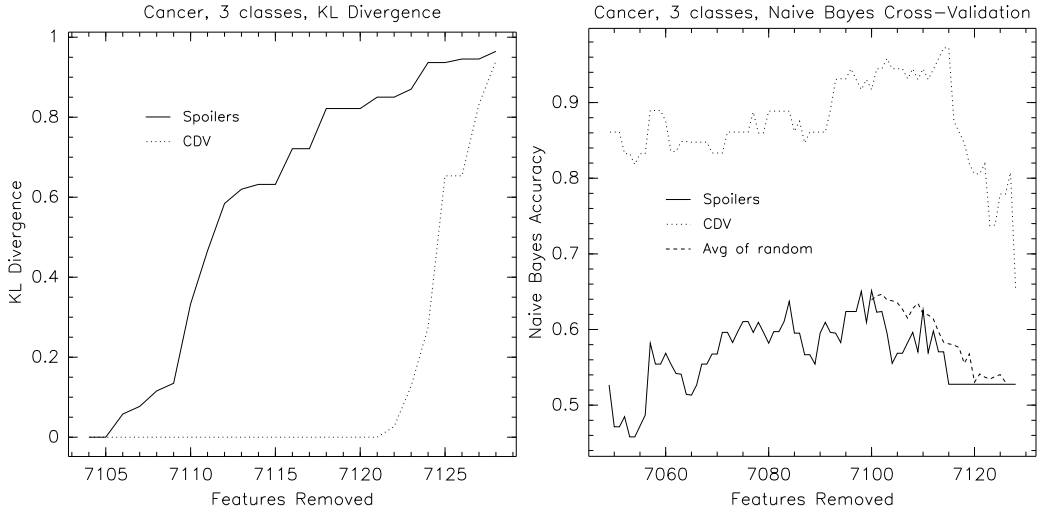


Figure 5.16: Results for two feature selection methods using the cancer dataset with three classes.

Gene	Annotation
D13789	mRNA for N-acetylglucosaminyltransferase III
HT26388	Mucin 1, Epithelial
L38941	ribosomal protein L34 (RPL34) mRNA
M17886	acidic ribosomal phosphoprotein P1 mRNA
M24194	MHC protein homologous to chicken B complex protein mRNA
M31606	phosphorylase kinase (PSK-C3) mRNA
M35093	secreted epithelial tumor mucin antigen (MUC1) gene
M84526	adipsin/complement factor D mRNA
M89957	cell surface glycoprotein (IGB) mRNA
U14969	ribosomal protein L28 mRNA
U86358	chemokine (TECK) mRNA
U89922	lymphotoxin beta isoform variant, alternatively spliced mRNA
X00437	mRNA for T-cell specific protein
X52056	mRNA for spi-1 proto-oncogene
X54741	CYPXIB2 gene for aldosterone synthase
X90846	mRNA for mixed lineage kinase 2

Figure 5.17: Subset of 16 genes that yield a 97% accuracy using Naive Bayes five-fold cross validation when used to predict the leukemia type (3 classes: AML, B-ALL, and T-ALL) of a cell.

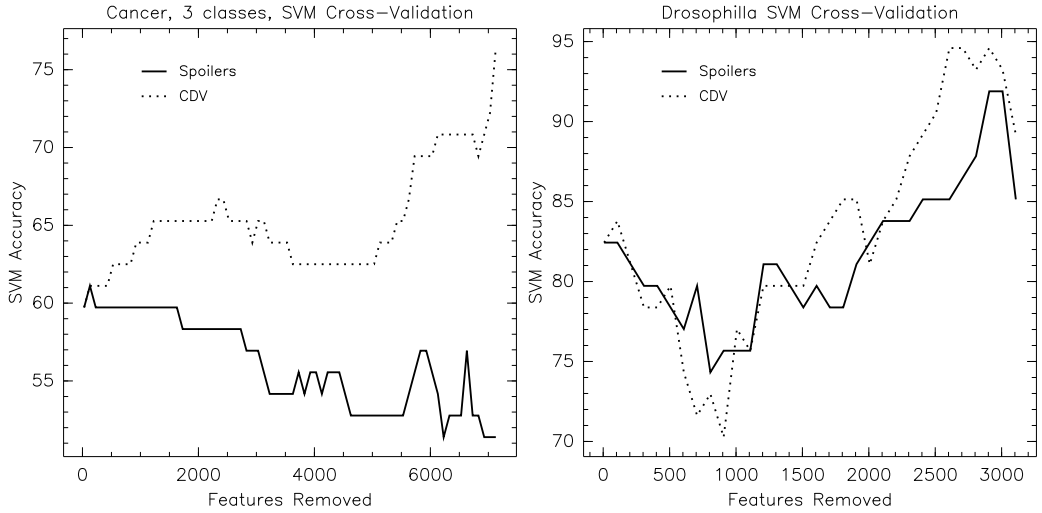


Figure 5.18: SVM results for two feature selection methods using the cancer dataset with three classes and the *Drosophila melanogaster* dataset

.

accuracy for both datasets, up to a maximum of 76.3% for the cancer with 3 classes dataset from an initial value of 59.7% and for the *Drosophila melanogaster* data set, the initial accuracy is 82.4% which improved to a maximum accuracy of 94.6%. For the cancer with 3 classes data, Spoilers performed as poorly as it did for the Naive Bayes classification. For the *Drosophila melanogaster* data set it did better, although not as well as the CDV method, with a highest accuracy of 91.9%.

The experiments that were carried out suggest that the KS algorithm does not work well with rank-order spaces. The result is not surprising since the Koller-Sahami method treats ranks as independent values and does not particularly distinguish between its relationship with ranks of other features.

The experiments also suggest that the CDV algorithm and the Spoilers algorithm generally perform better than the KL-divergence based methods (the Greedy KL algorithm and the KS algorithm). To understand the success of these methods over the KL-divergence based methods, examine the dataset shown in Figure 5.19. For this dataset, features x and y are consistent across the classes. However, any feature other than x may be removed and still maintain a KL divergence of zero. In addition, if y is removed, then any further feature selection process does not yield the optimal result. For this dataset, both the CDV algorithm and the Spoilers algorithm retain y and correctly choose either w or z because they examine the underlying ordinal

Class	Orders				Ranks w	x	y	z
C_1	w	x	y	z	1	2	3	4
C_1	x	w	z	y	2	1	4	3
C_2	y	x	z	w	4	2	1	3
C_2	w	z	y	x	1	4	3	2

Figure 5.19: Small rank-order dataset.

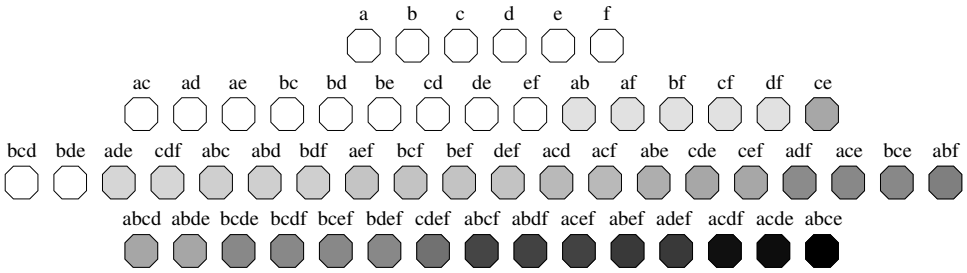


Figure 5.20: Chart showing the spectrum of KL divergences for all feature selection subsets for the data shown in Figure 5.6. Black denotes the highest KL divergence and white has a KL divergence of zero.

structure of the instances. In particular, the CDV algorithm deliberately addresses ordinal differences across instances in different classes while the Spoilers algorithm addresses similarities between instances in a class. On the other hand, the conditions imposed by the KL algorithm to arrive at a subset divergence of zero is easily satisfied resulting in a significant number of removable features, at the onset.

The phenomenon of multiple removable features is illustrated in Figure 5.20. The figure shows the KL divergence in shades of gray for all of the feature selection subsets of length 1 through 4 from the original dataset given in Figure 5.6. The subset with the highest KL divergence is (a,b,c,e) with a KL divergence of 1.343 and is denoted with black. Those choices that have a zero KL divergence are shown with white and the rest have various shades in between, based upon their magnitude. Note that any feature can be selected first and retain a KL divergence of zero. However, of the 17 possible paths leading to 2 features removed (still with a zero KL divergence), only 12 paths lead to the 2 choices of 3 features removed. A feature selection strategy based upon a greedy selection minimizing KL divergence at each step will only find one of these 2 best sets 70% of the time. This ratio decreases as the problem size increases.

Of course, the other feature selection strategies do not guarantee optimal results either. The KS algorithm chooses (a,b,c) resulting in a KL divergence of 0.249, the CDV algorithm chooses (b,c,f) which has a KL divergence of 0.313 and the Spoilers algorithm chooses (b,c,e) with a KL divergence of 0.616.

Almost all experiments show the Spoilers algorithm performing as well as the CDV algorithm, with the exception of the cancer dataset. For the experiments on the cancer datasets, Spoilers performed no better then the average random sample. The cancer data contained a large number features (7000+) and a small set of classes (2 or 3). In this case, the CDV algorithm might be better at global optimization than the Spoiler algorithm, which may have been confused by several different, locally optimal, choices early in the feature selection process. However, except for this extreme case, the Spoilers algorithm performed well and is much faster than the other algorithms. It is also possible that the methods of generating artificial rank-order data contained a bias toward feature selection using the Spoilers algorithm, since the generation method depends upon order.

It appears that KL divergence is not the best measure to use for a feature selection heuristic, since it is likely that a number of random features may be removed from rank-order space and still have a KL divergence of zero. Only in the later stages does the KL divergence become meaningful. Our experiments show that the Naive Bayes classifier showed adverse effects much sooner for the poorer methods than better methods, even though the KL divergence was unchanged. This indicates that the poor strategies were making incorrect choices, leading them down a suboptimal path in the feature selection tree.

In a modification of the previous techniques, the *temperament* of an item, x, is the total number of paired comparisons with x where the order is the same, minus the total number of paired comparisons with x where the order is the reverse. This total is for all pairwise comparisons between rank orders in a set of rank orders. By iteratively removing the most disagreeable elements from a set of rank orders, a subset of the most agreeable elements will be revealed.

The MPSS experiments are divided into logical sets. The standard set, with one of each tissue type under normal conditions, consists of CAF, INF, LEF, ROF, and SIF. Inflorescence, under various conditions, consists of INF, AP1, AP3, and SAP, while all of the leaves are LEF, S04, S52, and LES. The results of performing feature selection on all of these groups and the set of all experiments is shown in Figure 5.21.

Intuitively, when a gene is always one of the most highly expressed, or always has one of the lowest expression levels, then its ranking will be invariant with respect to

Standard Libraries		Leaves		Inflorescence		All Libraries	
Gene	Avg Rank	Gene	Avg Rank	Gene	Avg Rank	Gene	Avg Rank
At5g63150	1244	At5g25900	300	At4g13270	331	At5g15570	498
At5g47710	1361	At5g54590	466	At5g42060	677	At1g27750	966
At5g40200	1609	At5g67320	1183	At5g65010	1328	At5g47710	1734
At5g46750	1906	At5g42620	1593	At5g63810	1799	At3g42790	2426
At5g65720	2161	At5g01350	2632	At4g29040	2246	At3g21865	2599
At5g56170	2590	At5g04830	2787	At4g26110	2844	At4g09800	3099
At5g55190	2990	At5g43940	2982	At5g67360	3018	At3g09630	3149
At5g39740	3129	At5g44340	3166	At5g10980	3086	At5g10360	3200
At5g62300	3181	At5g61410	3222	At4g32260	3194	At1g02780	3228
At5g60390	3247	At5g01530	3245	At5g60390	3249	At5g60390	3244

Figure 5.21: Last 10 remaining genes when feature selection is performed on various 3,879 member sets of 17-base MPSS transcript abundance libraries.

the remaining genes. Therefore, those genes will be selected rather than other genes. Figure 5.21 shows that while there are some genes selected that fit this criterion, the majority of selected genes come from the middle ranks of the sets.

Note that, in Figure 5.21, no gene is common to all four sets. To find a common gene, the list length must contain at least 59 genes, in which case the common gene is At3g53430. To find the first common gene for leaves and inflorescence one need only increase the list length to 12 to find At4g32260. When comparing all $\binom{n}{2}$ sets experiments, the list length needs to be increased to 450 to find At5g60390, which is common to all.

Chapter 6

Conclusions and Future Research

The application of Spearman's ρ and Kendall's τ to different types of biological data has been explored in detail. While both expose different structural views of rank orders it is evident that methods based upon Spearman's ρ are more efficient from an algorithmic standpoint. Computing ρ is always an $\mathbf{O(n)}$ operation while τ requires at best $\mathbf{O(n \log n)}$ time and degrades to $\mathbf{O(n^2)}$ time if there are any tied ranks. For measuring the difference between two rank orders, both methods have utility, though ρ uses a finer graduated scale providing more bins to place all permutations in.

The early work in this area uses the z-score rather than the τ-correlation. This was done until it was realized how unwieldly the z-score becomes as n increases. Furthermore, d_K is the basis for the comparisons since Kendall and Gibbons [26, page 13] states that τ is preferable to ρ from many theoretical points of view. In spite of this, ρ can be substituted for τ for much of the following, as long as the substitution is consistent and the *tau*-correlation of one set of rank orders is never directly compared to the ρ-correlation of another set of orders. Which correlation type is better for biological data for most (or all) cases is still an open question.

There are many types of biological data that are amenable to being analyzed from a rank order perspective. Perhaps the most surprising aspect is that you can predict the life-cycle stage of *Drosophila melanogaster* based upon the ranks of the expression levels. The usual approach, when looking at life-cycle stages of organisms, is to look for gene expressions unique to that stage, whereas our method looks at only those genes that are expressed in every stage and notes the relative differences in order.

These methods have been shown to be an effective means for assessing predictive biological models vs. experimental data since many inherent differences in the types of results obtained are discarded and only the salient details are examined. For example, the predictive model PHX uses a unitless score for each gene, while microarrays use a different measure whose absolute value differs based on the current experimental baseline. This difference can be 2-fold or more between experiments taken from the same samples!

An efficient and effective means for combining the results of predictive models was presented. This method was based upon the τ-correlation, however the method is equally amenable to using the ρ-correlation. The output of the combined model is a new rank order vector, without values of the original models, so the utility of this method is limited to the output being only used further with rank correlation methods. That caveat aside, as successful improved model was obtained using PHX and a modified version of PHX as applied to *E. Coli* microarrays. This method of combining models, however, is still simplistic and based on a heuristic, further work can be done in this area.

The first large scale measurement of gene expression data was performed using microarrays, this was compared with the relatively new method using MPSS. Both microarrays and MPSS results for the same tissue samples of *Arabidopsis* under several different types of experimental conditions were compared against replicates of themselves and each other. MPSS was shown to be more self consistent than microarrays, however all of the microarray replicates were much more consistent with each other than they were with the MPSS results. Since MPSS and microarray are taken at slightly different times in the DNA $\rightarrow$ mRNA $\rightarrow$ Protein cycle, our results suggest that the expression levels at those times differ.

For machine learning, we have presented four feature selection algorithms, two of which are novel, for reasoning in rank-order spaces. Via experiments on both synthetic and real-world datasets, we have identified a number of conclusions relating the way the heuristics operate, the characteristics of the datasets, and the performance results.

It appears that KL divergence is not the best measure to use for a feature selection heuristic, since it is likely that a number of random features may be removed from rank-order space and still have a KL divergence of zero. Only in the later stages does the KL divergence become meaningful. Our experiments show that the Naive Bayes classifier exhibits adverse effects much sooner for the KL-divergence based methods, even when the KL divergence is unchanged. This indicates that these strategies were making incorrect choices, leading them down a suboptimal path in

the feature selection tree. On the other hand, the CDV algorithm and the Spoilers algorithm tend to make the correct choices as these algorithms deliberately examine the underlying ordinal structure of the instances.

The CDV method is based upon Spearman's ρ and is more efficient and effective than the method (Spoilers) based upon Kendall's τ, especially for real (not synthetic) data. For CDV, the criteria for removing a feature is based on a brute force method that examined all features and removes the one that maintains the remaining set rank vectors as similar to the original set as possible. On the other hand, Spoilers removes a feature based upon the locally optimal criteria of making the sets more self similar, while increasing the division between sets. This method, even though it is a heuristic, is only slightly more efficient than the brute force method of CDV, especially if there are tied rankings. The success of Spoilers using the artificial datasets can perhaps be explained by the fact that the datasets are generated using τ-correlation methods. Note that both of these feature selection strategies were based on rank correlation methods, but the two classifiers, Naive Bayes and SVM, used to test their effectiveness are not, yet the methods are still effective.

This research opens several questions for future exploration. Is there a method for generating artificial rank-order data that is more like real world datasets? Is there a faster algorithm for CDV so that its speed is favorably comparable with the Spoilers algorithm? Is there a better measure of divergence for rank-orders than KL divergence, one that considers relationships between features instead of just their values? Finally, are there better classification algorithms for rank-ordered data?

Appendix A

Implementation

The algorithms were first prototyped in Python and implemented in C where necessary. The code was run on a variety of Macintosh G5s running Mac OS X and Pentium IVs running Linux.

A.1 Python Code

```
#!/usr/bin/python
from __future__ import generators
import copy
import random
import string
import math
import sys
import re
import types

# Usage
#ar = [1,2,3,4,5,6]
#for uc in gen_choose(ar,4):
#    print uc
def gen_choose(items, n):
    if n==0: yield []
    elif n==len(items): yield items
    else:
        for i in xrange(len(items)):
            for cc in gen_choose(items[i+1:],n-1):
                yield [items[i]]+cc
```

```
def zeros(size):
    arry = range(size)
    for i in range(size):
        arry[i] = 0
    return arry

def zeroszeros(lst):
    A={}
    for i in lst:
        a = {}
        for j in lst:
            if not i==j:
                a[j]=0
        A[i]=a
    return A

class AlphaCount:
    def __init__(self):
        anum = 97
        self.LofA = 26
        self.i2a={}

        for x, y in zip(range(anum, anum+self.LofA), range(1+self.LofA)):
            self.i2a[y] = chr(x)
    def conv(self, num):
        m = (num % self.LofA)
        n = num / self.LofA
        if (n > 0):
            return self.conv(n-1) + self.i2a[m]
        else:
            return self.i2a[m]
    def range(self, num):
        x = range(num)
        y = map(self.conv, x)
        y.sort()
        return y

class Order:
    def __init__(self):
        self.order = {}
        self.rank = []
        self.valid_rank = True
        self.length = 0
        self.has_ties = False
        self.num_ties = 0
    def init_rank(self, rank):
        self.length = len(rank)
        n = AlphaCount()
```

```
        self.canonical = n.range(self.length)
        for a,b in zip(rank, self.canonical):
            self.order[b]=a
        self.rank = map(float,rank)
        self.valid_rank = False
    def init_order(self, order):
        self.length = len(order)
        self.has_ties = False
        for a,b in zip(order, range(1,self.length+1)):
            self.order[a] = b
        # No ties here, so this is faster than using self.update_rank()
        self.rank = []
        self.canonical = copy.copy(order)
        self.canonical.sort()
        for k in self.canonical:
            self.rank.append(int(self.order[k]))
        self.valid_rank = True
    def init_order_rank(self, dict):
        self.order = copy.copy(dict)
        self.length = len(dict)
        self.canonical = dict.keys()
        self.canonical.sort()
        self.valid_rank = False
        self.update_rank()
    def update_rank(self):
        keys = self.get_order()
        startList=[]; stopList=[]
        startList.append(0)
        for k in range(self.length-1):
            cur = keys[k]
            next = keys[k+1]
            if (self.order[cur] < self.order[next]):
                stopList.append(k)
                startList.append(k+1)
        stopList.append(k+1)
        if len(startList) != len(stopList):
            print "Something is drastically wrong!"
            sys.exit()
        self.rank_dict = {}
        self.has_ties = False
        for k in range(len(startList)):
            start=startList[k]
            stop=stopList[k]
            if start == stop:
                key = keys[start]
                self.rank_dict[key] = start+1
            else:
                self.has_ties = True
```

```
                n = (start + stop + 2) / 2.0
                for i in range(start,stop+1):
                    key = keys[i]                                    120
                    self.rank_dict[key] = n
        self.rank = []
        for k in self.canonical:
            self.rank.append(self.rank_dict[k])
        self.valid_rank = True
    def reverse(self):
        for k,v in self.order.items():
            self.order[k] = v * -1
        self.valid_rank = False
    def get_order(self):                                             130
        lst = self.order.keys()
        lst.sort(lambda a,b: cmp(self.order[a],self.order[b]))
        return lst
    def get_rank(self):
        if not self.valid_rank:
            self.update_rank()
        return self.rank
    def intersection(self, other):
        intr = []
        for key in self.order.keys():                                140
            if other.order.has_key(key):
                intr.append(key)
        intr.sort()
        return intr
    def get_bool_features(self):
        lst = self.order.keys()
        lst.sort()
        feats = {}
        for i in range(self.length):
            a = lst[i]                                               150
            for j in range(i+1, self.length):
                b = lst[j]
                str = a + '<' + b
                feats[str] = self.order[a] < self.order[b]
                str = a + '=' + b
                feats[str] = self.order[a] == self.order[b]
        return feats
    def get_bool_features_groups(self):
        lst = self.order.keys()
        lst.sort()                                                   160
        feats = {}
        groups = {}
        for feat in lst:
            groups[feat] = []
        for i in range(self.length):
```

```
            a = lst[i]
            for j in range(i+1, self.length):
                b = lst[j]
                str = a + '<' + b
                groups[a].append(str)
                groups[b].append(str)
                str = a + '=' + b
                groups[a].append(str)
                groups[b].append(str)
        return groups
    def del_feature(self, feature, update=True):
        loc = self.canonical.index(feature)
        del self.canonical[loc]
        del self.order[feature]
        self.length -= 1
        if update:
            if self.valid_rank:
                val = self.rank[loc]
                del self.rank[loc]
                for i in range(self.length):
                    if self.rank[i] == val:
                        self.rank[i] = self.rank[i] - 0.5
                    elif self.rank[i] > val:
                        self.rank[i] = self.rank[i] - 1
            else:
                self.update_rank()
        else:
            self.valid_rank = False
    def spearman_dist(self, other):
        "Returns the Spearman (Euclidian) distance"
        d2 = 0
        r1 = self.get_rank()
        r2 = other.get_rank()
        self.diff_count = range(self.length)
        for n in range(self.length):
            a = r1[n]
            b = r2[n]
            d2 += (a-b)**2
            self.diff_count[n] = abs(a-b)
        #return math.sqrt(d2)
        return d2
    def kendall_dist(self, other):
        if self.has_ties or other.has_ties:
            return self.kendall_dist_ties(other)
        else:
            #return self.kendall_dist_noties(other)
            return self.kendall_dist_ties(other)
    def kendall_dist_ties(self, other):
```

```
        keys = self.order.keys()
        self.num_ties = 0
        self.comp_ties = 0
        other.num_ties = 0
        self.P = 0
        self.Q = 0
        self.diff_count = {}                                          220
        for k in keys:
            self.diff_count[k] = 0
        count = 0
        for i in range(len(keys)):
            k1 = keys[i]
            for j in range(i+1,len(keys)):
                k2 = keys[j]
                u = cmp(self.order[k1],self.order[k2])
                v = cmp(other.order[k1],other.order[k2])
                if u != 0 and v != 0:                                 230
                    if u != v:
                        self.Q += 1
                        count += 1
                        self.diff_count[k1] += 1
                        self.diff_count[k2] += 1
                    else:
                        self.P += 1
                else:
                    self.comp_ties +=1
                    if u == 0:                                        240
                        self.num_ties += 1
                    if v == 0:
                        other.num_ties += 1

                #print k1, k2, "\t", u, v, "\t", count
        return count
    def kendall_dist_noties(self, other):
        def QuickCount(A,i,k):
            if i < k:
                j = (i+k)/2                                           250
                x = QuickCount(A,i,j)
                y = QuickCount(A,j+1,k)
                z = Count(A,i,j,k)
                return x + y + z
            else:
                return 0
        def Count(A,i,j,k):
            left = A[i:j+1]
            right = A[j+1:k+1]
            #print left, right                                        260
            count = 0
```

```
            right_merged = 0
            for idx in range(i,k+1):
                if len(left) == 0:
                    A[idx]=right.pop(0)
                elif len(right) == 0:
                    A[idx]=left.pop(0)
                    self.diff_count[A[idx]] += right_merged
                else:
                    s = left[0]                                          270
                    t = right[0]
                    u = cmp(pos[s],pos[t])
                    v = cmp(spos[s],spos[t])
                    if u == 0 or v == 0:
                        print s, t
                        A[idx]=right.pop(0)
                    elif u == -1:
                        A[idx]=left.pop(0)
                        self.diff_count[s] += right_merged
                    else:                                                280
                        A[idx]=right.pop(0)
                        ct = len(left)
                        count += ct
                        self.diff_count[t] += ct
                        right_merged += 1
                #print A[idx], self.diff_count[A[idx]]

            return count
        o1 = self.get_order()
        pos = other.order                                                290
        spos = self.order
        self.diff_count = {}
        for k in o1:
            self.diff_count[k] = 0
        return QuickCount(o1,0,len(o1)-1)
    def tau_dist(self, other):
        dk = self.kendall_dist(other)
        n = len(self.order)
        if self.has_ties or other.has_ties:
            S = self.P - self.Q                                          300
            U = math.sqrt((n*(n-1)/2) - self.num_ties)
            V = math.sqrt((n*(n-1)/2) - self.num_ties)
            return S/(U*V)
        else:
            return 1.0 - ( (4.0*dk) / (n*(n-1)) )
    def count_all_inversions(self, other):
        keys = self.order.keys()
        n = len(keys)
        self.P = 0; self.Q = 0
```

```
        self.diff_count = {}
        self.inv_count = {}
        for key in keys:
            self.diff_count[key] = 0
            self.inv_count[key] = {}
        for a in range(0,n):
            k1 = keys[a]
            for b in range(a+1,n):
                k2 = keys[b]
                a1 = cmp(self.order[k1],self.order[k2])
                a2 = cmp(other.order[k1],other.order[k2])
                if ((a1==0) or (a2==0)):
                    pass
                else:
                    if (a1 != a2):
                        Q += 1
                        self.inv_count[k1][k2] = 1
                        self.inv_count[k2][k1] = 1
                        self.diff_count[k1] += 1
                        self.diff_count[k2] += 1
                    else:
                        P += 1
        return Q
    def rho_dist(self, other):
        "Returns the Spearman rho (Euclidian distance)"
        if self.length != other.length:
            rho = self.rho_dist_int(other)
            return rho
        d2 = 0
        self.diff_count={}
        for a,b in zip(self.get_rank(), other.get_rank()):
            d2 += (a-b)**2
            self.diff_count[a] = abs(a-b)
        n = self.length
        rho = 1.0-((6.0*d2)/(n**3 - n))
        return rho
    def rho_dist_int(self, other):
        "Returns the Spearman rho (Euclidian distance)"
        keys = self.intersection(other)
        d2 = 0
        self.diff_count={}
        for key in keys:
            a = self.order[key]
            b = other.order[key]
            d2 += (a-b)**2
            self.diff_count[key] = abs(a-b)
        n = len(keys)
        rho = 1.0-((6.0*d2)/(n**3 - n))
```

```
        return rho

class OrderSet:                                                          360
    def __init__(self):
        self.orders = []
        self.members = 0
        self.length = 0
    def add_order(self, order):
        self.orders.append(order)
        self.members += 1
        if (self.length == 0):
            self.length = order.length
        elif (self.length != order.length):                              370
            print "Order length does not match rest of group"
            sys.exit()
    def get_features_legend(self):
        legend = self.orders[0].get_features_legend()
        return legend
    def get_features(self):
        odr = self.orders[0]
        features = odr.order.keys()
        features.sort()
        return features                                                  380
    def del_feature(self, feature, update=True):
        for odr in self.orders:
            odr.del_feature(feature, update)
        self.length -= 1
    def del_last(self):
        del self.orders[-1]
        self.members -= 1
    def get_ranks(self):
        ranks = []
        for odr in self.orders:                                          390
            ranks.append(odr.get_rank())
        return ranks
    def print_ranks(self):
        for odr in self.orders:
            print odr.get_rank()
    def rho_concordance(self):
        # Also known as Kendall's W
        rank_total = zeros(self.length)
        for i in range(self.members):
            cur_rank = self.orders[i].get_rank()                         400
            for j in range(self.length):
                rank_total[j] += cur_rank[j]
        mean = 0.0
        for i in range(self.length):
            mean += rank_total[i]
```

```
        mean /= self.length
        S = 0.0
        for i in range(self.length):
            S += (rank_total[i] - mean)**2
        return (12.0*S)/(self.members**2 * (self.length**3 - self.length))
    def rho_center(self):
        rank_total = zeros(self.length)
        for i in range(self.members):
            cur_rank = self.orders[i].get_rank()
            for j in range(self.length):
                rank_total[j] += cur_rank[j]
        feats = self.get_features()
        new_ranks = {}
        for k, v in zip(feats, rank_total):
            new_ranks[k] = v
        odr = Order()
        odr.init_order_rank(new_ranks)
        return odr
    def rho_diameter(self):
        max_dist = 1
        for i in range(self.members):
            iOrder = self.orders[i]
            for j in range(i+1, self.members):
                cDist = iOrder.rho_dist(self.orders[j])
                # The farther from 1, the greater the distance
                if (cDist < max_dist):
                    max_dist = cDist
        return max_dist
    def print_compare_all(self):
        for x in self.compare:
            for y in x:
                print '%5.2f' % y,
            print
    def rho_compare_all(self):
        self.compare = []
        self.diff_count = {}
        for i in range(self.members):
            self.compare.append([])
            iOrder = self.orders[i]
            for j in range(self.members):
                if i == j:
                    self.compare[i].append(1.0)
                elif j < i:
                    self.compare[i].append(self.compare[j][i])
                else:
                    jOrder = self.orders[j]
                    dist = iOrder.rho_dist_int(jOrder)
                    self.compare[i].append(dist)
```

```
                for key in iOrder.diff_count:
                    if self.diff_count.has_key(key):
                        self.diff_count[key] += iOrder.diff_count[key]
                    else:
                        self.diff_count[key] = iOrder.diff_count[key]
    def tau_compare_all(self):
        self.compare = []
        self.diff_count = {}
        for i in range(self.members):
            self.compare.append([])
            iOrder = self.orders[i]
            for j in range(self.members):
                if i == j:
                    self.compare[i].append(1.0)
                elif j < i:
                    self.compare[i].append(self.compare[j][i])
                else:
                    jOrder = self.orders[j]
                    dist = iOrder.tau_dist(jOrder)
                    self.compare[i].append(dist)
                    for key in iOrder.diff_count:
                        if self.diff_count.has_key(key):
                            self.diff_count[key] += iOrder.diff_count[key]
                        else:
                            self.diff_count[key] = iOrder.diff_count[key]
    def count_all_inversions(self):
        self.diff_count = {}
        self.inv_count = {}
        for i in range(self.members):
            iOrder = self.orders[i]
            for j in range(i+1,self.members):
                jOrder = self.orders[j]
                dist = iOrder.count_all_inversions(jOrder)
                for key in iOrder.diff_count:
                    if self.diff_count.has_key(key):
                        self.diff_count[key] += iOrder.diff_count[key]
                    else:
                        self.diff_count[key] = iOrder.diff_count[key]
                for key in iOrder.inv_count.keys():
                    if self.inv_count.has_key(key):
                        ninv = iOrder.inv_count[key]
                        for nkey in ninv.keys():
                            if self.inv_count[key].has_key(nkey):
                                self.inv_count[key][nkey] += ninv[nkey]
                            else:
                                self.inv_count[key][nkey] = ninv[nkey]
                    else:
                        self.inv_count[key] = copy.copy(iOrder.inv_count[key])
```

```
class OrderSetSet:
    def __init__(self):
        self.ordersets = {}
        self.classes = []
        self.length = 0
    def addOrderSet(self, os, cls):
        self.ordersets[cls] = os
        self.classes.append(cls)                                    510
        self.length=os.length
    def load_data(self,filename):
        # place data into the structure
        self.ordersets = {}
        feat_list = []
        for line in open(filename):
            if line[0] != '#':
                if line[:5] == 'feats':
                    feat_list = line.split()
                    del feat_list[0]                                520
                else:
                    arr = line.split(' ')
                    cls = arr[0]
                    del arr[0]
                    odr = Order()
                    f_arr = map(float, arr)
                    #odr.init_rank(f_arr)
                    orank = {}
                    for key, val in zip(feat_list, f_arr):
                        orank[key] = val                            530
                    odr.init_order_rank(orank)
                    if (not self.ordersets.has_key(cls)):
                        odrs = OrderSet()
                        self.ordersets[cls] = odrs
                    self.ordersets[cls].add_order(odr)
                    self.length = odr.length
        self.classes = self.ordersets.keys()
    def get_an_order(self):
        return self.ordersets[self.classes[0]].orders[0]
    def gen_orders(self):                                           540
        for cls in self.classes:
            ords = self.ordersets[cls]
            for ord in ords.orders:
                yield (cls, ord)
    def del_feature(self, feature, update=True):
        for key in self.ordersets.keys():
            odrs = self.ordersets[key]
            odrs.del_feature(feature, update)
        self.length -= 1
```

```
    def del_class(self, cls):
        del(self.ordersets[cls])
        self.classes = self.ordersets.keys()
    def get_features_legend(self):
        keys = self.ordersets.keys()
        legend = self.ordersets[keys[0]].get_features_legend()
        return legend
    def total_concordance(self):
        total = 0.0
        count = 0
        for key in self.classes:
            odrs = self.ordersets[key]
            total += odrs.rho_concordance()
            count += 1
        total = total / count
        return total
    def total_diameter(self):
        total = 0
        for key in self.classes:
            odrs = self.ordersets[key]
            total += odrs.rho_diameter()
        return total
    def center_concordance(self):
        centers = OrderSet()
        for key in self.classes:
            odrs = self.ordersets[key]
            center = odrs.rho_center()
            centers.add_order(center)
        return centers.rho_concordance()
    def center_rho_distance_vector(self):
        dvec = []
        for keys in gen_choose(self.classes,2):
            odrs1 = self.ordersets[keys[0]]
            odrs2 = self.ordersets[keys[1]]
            c1 = odrs1.rho_center()
            c2 = odrs2.rho_center()
            dvec.append(c1.rho_dist(c2))
        return dvec
    def get_features(self):
        keys = self.classes
        odrs = self.ordersets[keys[0]]
        odr = odrs.orders[0]
        features = odr.order.keys()
        features.sort()
        return features
    def get_bool_features(self):
        ord = self.get_an_order()
        theFeats = ord.get_bool_features().keys()
```

```
        useFeats = []
        for feat in theFeats:
            if feat.find('=') >= 0:                                          600
                for cls, ord in self.gen_orders():
                    bfeats = ord.get_bool_features()
                    if bfeats[feat]:
                        useFeats.append(feat)
                        break
            else:
                useFeats.append(feat)
        return useFeats
    def print_quick_summary(self):
        print "Classes:", len(self.classes)                                  610
        print "Features:", self.length
        total = 0
        for key in self.classes:
            print "\tClass:", key, "\tMembers:",
            odrs = self.ordersets[key]
            cnt = len(odrs.orders)
            total += cnt
            print cnt
        print "Total instances:", total
    def print_summary(self):                                                 620
        for key in self.classes:
            odrs = self.ordersets[key]
            print "Class =", key
            #for odr in odrs.orders:
            #    print " ", odr.get_rank()
            print " Concordance =", odrs.rho_concordance()
            #print " Diameter = ", odrs.rho_diameter()
            #center = odrs.rho_center()
            #print " Center =", center.get_rank()
        print                                                                630
        print "      Concordance sum =", self.total_concordance()
        print "         Diameter sum =", self.total_diameter()
        print "Concordance of centers =", self.center_concordance()
    def print_data(self):
        feats = self.get_features()
        print "feats",
        for feat in feats:
            print feat,
        print
        cls_num = 0                                                          640
        for cls in self.classes:
            os = self.ordersets[cls]
            for ord in os.orders:
                r = ord.get_rank()
                print cls_num, ' '.join(map(str,r))
```

```
            cls_num += 1
    def print_orig_data(self):
        feats = self.get_features()
        print "feats",
        for feat in feats:
            print feat,
        print
        cls_num = 0
        for cls in self.classes:
            os = self.ordersets[cls]
            for ord in os.orders:
                print cls_num,
                for f in feats:
                    print ord.order[f],
                print
            cls_num += 1
    def print_bool_data(self, fh):
        bfeats = self.get_bool_features()
        title = "class\t" + '\t'.join(bfeats)
        fh.write(title)
        fh.write("\nd")
        for i in range(len(bfeats)):
            fh.write("\td")
        fh.write("\n")
        fh.write("class")
        for i in range(len(bfeats)):
            fh.write("\t")
        fh.write("\n")
        for cls in self.classes:
            os = self.ordersets[cls]
            for ord in os.orders:
                bf = ord.get_bool_features()
                fh.write(cls)
                for i in bfeats:
                    fh.write("\t"+str(bf[i]*1))
                fh.write("\n")
    def print_cont_data(self, fh):
        cfeats = self.get_features()
        title = "class\t" + '\t'.join(cfeats) + '\tcontclass'
        fh.write(title)
        fh.write("\nstring")
        for i in range(len(cfeats)):
            fh.write("\tc")
        fh.write("\tc")
        fh.write("\n")
        fh.write("i")
        for i in range(len(cfeats)+1):
            fh.write("\t")
```

```
        fh.write("class")
        fh.write("\n")
        cls_num = 0
        for cls in self.classes:
            cls_num += 1
            os = self.ordersets[cls]
            for ord in os.orders:                                    700
                bf = ord.get_rank()
                fh.write(cls)
                for i in bf:
                    fh.write("\t"+str(i))
                fh.write('\t'+str(cls_num))
                fh.write("\n")
    def print_svm_data(self, fh):
        cls_num = 0
        for cls in self.classes:
            cls_num += 1                                             710
            os = self.ordersets[cls]
            for ord in os.orders:
                fh.write(str(cls_num) + '\t')
                f = ord.order
                #f = ord.get_rank()
                idx = 0
                for k in f.keys():
                #for k in f:
                    idx += 1
                    fh.write(str(idx) + ':' + str(f[k]) + ' ')       720
                    #fh.write(str(idx) + ':' + str(k) + ' ')
                fh.write('\n')
    def totconc_del_order(self):
        del_feats = []
        subjects = copy.deepcopy(self)
        for i in range(self.length-2):
            old_subjects = copy.deepcopy(subjects)
            feats = subjects.get_features()
            max_concord = 0
            for feat in feats:                                       730
                subjects.del_feature(feat)
                concord = subjects.total_concordance()
                if (concord > max_concord):
                    max_concord = concord
                    max_feat = feat
                subjects = copy.deepcopy(old_subjects)
            del_feats.append(max_feat)
            subjects.del_feature(max_feat)
        return del_feats
    def centconc_del_order(self):                                    740
        del_feats = []
```

```
        subjects = copy.deepcopy(self)
        for i in range(self.length-2):
            old_subjects = copy.deepcopy(subjects)
            feats = subjects.get_features()
            cent_concord = 1
            for feat in feats:
                subjects.del_feature(feat)
                concord = subjects.center_concordance()
                if (concord < cent_concord):
                    cent_concord = concord
                    cent_feat = feat
                subjects = copy.deepcopy(old_subjects)
            del_feats.append(cent_feat)
            subjects.del_feature(cent_feat)
        return del_feats
    def ratio_del_order(self):
        del_feats = []
        subjects = copy.deepcopy(self)
        for i in range(self.length-2):
            old_subjects = copy.deepcopy(subjects)
            feats = subjects.get_features()
            ratio_min = 10000.0
            for feat in feats:
                subjects.del_feature(feat)
                tconcord = subjects.total_concordance()
                cconcord = subjects.center_concordance()
                ratio = cconcord / tconcord
                if (ratio < ratio_min):
                    ratio_min = ratio
                    ratio_feat = feat
                subjects = copy.deepcopy(old_subjects)
            del_feats.append(ratio_feat)
            subjects.del_feature(ratio_feat)
        return del_feats
    def feat_del_order(self):
        del_order = []

        os = copy.deepcopy(self)

        while os.length > 2:
            fs = FeatureSelect()

            for key in os.ordersets.keys():
                odrs = os.ordersets[key]
                for odr in odrs.orders:
                    fs.add_instance(key, odr.get_features())
                    fGroups = odr.get_features_groups()
```

```
            feat = fs.choose_deletion_group(fGroups)
            del_order.append(feat)
            os.del_feature(feat)

        return del_order
    def tau_compare_all(self):
        self.diff_count = {}
        for cls in self.classes:
            oset = self.ordersets[cls]
            oset.tau_compare_all()
            for key in oset.diff_count.keys():
                if self.diff_count.has_key(key):
                    self.diff_count[key] += oset.diff_count[key]
                else:
                    self.diff_count[key] = oset.diff_count[key]
        return self.diff_count
    def count_all_inversions(self):
        self.diff_count = {}
        self.inv_count = {}
        for cls in self.classes:
            oset = self.ordersets[cls]
            oset.count_all_inversions()
            for key in oset.diff_count:
                if self.diff_count.has_key(key):
                    self.diff_count[key] += oset.diff_count[key]
                else:
                    self.diff_count[key] = oset.diff_count[key]
            for key in oset.inv_count.keys():
                if self.inv_count.has_key(key):
                    ninv = oset.inv_count[key]
                    for nkey in ninv.keys():
                        if self.inv_count[key].has_key(nkey):
                            self.inv_count[key][nkey] += ninv[nkey]
                        else:
                            self.inv_count[key][nkey] = ninv[nkey]
                else:
                    self.inv_count[key] = copy.copy(oset.inv_count[key])
    def print_del_stats(self, del_order):
        os = copy.deepcopy(self)
        tc = os.total_concordance()
        cc = os.center_concordance()
        ratio = cc / tc
        tc_list = [tc]
        cc_list = [cc]
        ratio_list = [ratio]
        for feat in del_order:
            os.del_feature(feat)
            tc = os.total_concordance()
```

```
            cc = os.center_concordance()
            ratio = cc / tc
            tc_list.append(tc)
            cc_list.append(cc)
            ratio_list.append(ratio)
        print "tc:", tc_list, "cc:", cc_list, "ratio:", ratio_list
    def KL_dist(self, oss):
        F = []
        count = 0
        for cls, ord in self.gen_orders():
            I = Instance()
            I.cls = cls
            I.feats = str(ord.get_rank())
            F.append(I)
            count += 1
        Fg = []
        for cls, ord in oss.gen_orders():
            I = Instance()
            I.cls = cls
            I.feats = str(ord.get_rank())
            Fg.append(I)
        all = []
        all.extend(F)
        all.extend(Fg)
        itemCounts = {}
        for item in all:
            f = str(item.feats)
            cf = item.cls + '|' + f
            if not itemCounts.has_key(f):
                itemCounts[f] = 0
            if not itemCounts.has_key(cf):
                itemCounts[cf] = 0
            itemCounts[f] += 1
            itemCounts[cf] += 1
        P = {}
        for k in itemCounts.keys():
            P[k] = itemCounts[k] / float(count)
        val = 0.0
        for item1, item2 in zip(F, Fg):
            f = str(item1.feats)
            fg = str(item2.feats)
            subval = 0.0
            for cls in self.classes:
                cf = cls + '|' + f
                cfg = cls + '|' + fg
                if P.has_key(cf):
                    Pcf = P[cf] / P[f]
                    if P.has_key(cfg):
```

```
                    Pcfg = P[cfg] / P[fg]

                    subval += Pcf * math.log( Pcf / Pcfg )
            val += P[f] * subval
        return val                                                    890



##    def KL_dist(self, oss):
##        pc = ProbCount()
##        for cls, ord in self.gen_orders():
##            pc.addInstance(cls, ord.get_rank())
##        pc.print_data()

class ProbCount:                                                     900
    def __init__(self):
        self.cls = []
        self.ranks = []
        self.head = {}
        self.Pf = {}
        self.Pcf = {}
    def add_instance(self, cls, rank):
        self.cls.append(cls)
        self.ranks.append(rank)
        number = len(self.ranks)-1                                   910
        done = False
        cp = 0
        level = self.head
        prevlevel = level
        clsRank = (cls,number)
        for cp, key in enumerate(rank):
            if level.has_key(key):
                tp = type(level[key])
                if tp is types.ListType:
                    if cp >= len(rank)-1:                            920
                        level[key].append(clsRank)
                    else:
                        oldLeaf = level[key][0]
                        level[key] = {}
                        level = level[key]
                        nextkey = self.ranks[oldLeaf[1]][cp+1]
                        level[nextkey] = [oldLeaf]
                elif tp is types.DictType:
                    prevlevel = level
                    level = level[key]                               930
                else:
                    print "Error! Node data structure is neither list nor dict."
                    sys.exit()
```

```
            else:
                level[key] = [clsRank]
                break
    def print_data(self):
        print self.head
    def count(self):
        def subcount(level):
            for key in level.keys():
                node = level[key]
                tp = type(node)
                if tp is types.ListType:
                    count = len(node)
                    Pf = node[0][1]
                    print count, Pf, node
                    for item in node:
                        cls = item[0]
                        ord = item[1]
                else:
                    subcount(node)
        subcount(self.head)

class Instance:
    pass

class FeatureSelect:
    def __init__(self):
        self.instances = []
        self.count = 0
        self.featCount = 0
        self.K = 5 # default value
        self.features = []
        self.fGroups = {}
    def set_K(self, newK):
        self.K = newK
    def add_instance(self, cls, feats):
        i = Instance()
        i.cls = cls
        i.feats = feats
        if self.featCount == 0:
            self.featCount = len(feats)
        else:
            if self.featCount != len(feats):
                print "Unequal number of features!"
                sys.exit()
        self.instances.append(i)
        self.count += 1
    def add_OrderSetSet(self, oss):
        self.oss = copy.deepcopy(oss)
```

```
    self.features = oss.get_features()
def populate_instances(self):
    self.instances = []
    self.count = 0
    self.featCount = 0
    for key in self.oss.classes :
        odrs = self.oss.ordersets[key]
        for odr in odrs.orders:
            self.add_instance(key, odr.get_rank())
    self.features = self.oss.get_features()
def populate_instances_bool(self):
    self.instances = []
    self.count = 0
    self.featCount = 0
    for key in self.oss.classes :
        odrs = self.oss.ordersets[key]
        for odr in odrs.orders:
            self.add_instance(key, odr.get_bool_features())
    self.features = self.oss.get_features()
def print_all(self):
    for inst in self.instances:
        print inst.cls, inst.feats
def del_feature(self, feature):
    self.featCount -= 1
    for inst in self.instances:
        del(inst.feats[feature])
def choose_deletion(self, use_group=0):
    def Pr(cp):
        n = itemCounts[cp]
        c = string.split(cp,'|')
        d = itemCounts[c[1]]
        return n/d
    def get_ij(cp):
        [head, tail] = string.split(cp,'|')
        [first, second] = string.split(tail,',')
        [i,discard] = string.split(first,':')
        [j,discard] = string.split(second,':')
        return [i, j]
    def get_ijmn(cp):
        [head, tail] = string.split(cp,'|')
        [first, second] = string.split(tail,',')
        [a,discard] = string.split(first,':')
        [b,discard] = string.split(second,':')
        [i,j] = split.re('[<=]',a)
        [m,n] = re.split('[<=]',b)
        return [i, j, m, n]
    def get_subitem1(cp):
        [cls, cond] = string.split(cp,'|')
```

```
    [subitem, discard] = string.split(cond,',')
    return cls + '|' + subitem
def get_subitem2(cp):
    [cls, discard] = string.split(cp,'|')
    [discard, subitem] = string.split(cp,',')
    return cls + '|' + subitem
def D(cp1, cp2):
    mu = Pr(cp1)
    sigma = Pr(cp2)
    return mu * math.log(mu/sigma)
def gen_gfList(i):
    for k in gfList[i]:
        yield gfList[i][k]
def subfeats(dict, feats):
    newlst = []
    for feat in feats:
        newlst.append(dict[feat])
    return newlst
def combine(l1, l2):
    st = ''
    for a,b in zip(l1, map(str, l2)):
        st += a + ':' + b + ','
    return st[:-1]
def list_diff(l1, l2): # NB. l1 > l2
    h = {}
    diff = []
    for e in l2:
        h[e] = 0
    for e in l1:
        if not h.has_key(e):
            diff.append(e)
    return diff
itemCounts = {}
Dlist = {}
gfList = {}
if use_group:
    theFeats = self.oss.get_bool_features()
    #fGroups = ord.get_features_groups()
    #print fGroups
else:
    theFeats = self.features

for feat in theFeats:
    gfList[feat] = {}

# Count items
count = 0
for cls, ord in self.oss.gen_orders():
```

```
        count += 1
        for nfeats in [1,2,len(theFeats)]:
            for sf in gen_choose(theFeats, nfeats):
                if use_group:
                    sfeats = ord.get_bool_features()
                else:
                    sfeats = ord.order
                vals = subfeats(sfeats, sf)
                f = combine(sf,vals)
                cf = cls + "|" + f
                if not itemCounts.has_key(f):
                    itemCounts[f] = 0
                if not itemCounts.has_key(cf):
                    itemCounts[cf] = 0
                itemCounts[f] += 1
                itemCounts[cf] += 1
                if nfeats == 1:
                    gfList[sf[0]][f] = 0
                if nfeats == 2:
                    Dlist[cf] = 0

    # Compute probabilities
    for x in itemCounts.keys():
        itemCounts[x] /= float(count)

    # Compute Gamma
    gamma = zeroszeros(self.features)
    for item in Dlist.keys():
        if use_group:
            #print item
            [i,j,m,n] = get_ijmn(item)
            #print i, j , m, n
        else:
            [i,j] = get_ij(item)
        subitem = get_subitem1(item)
        gamma[i][j] += D(item, subitem)
        if use_group:
            gamma[m][n] += D(item, subitem)
        subitem = get_subitem2(item)
        gamma[i][j] += D(item, subitem)
        if use_group:
            gamma[m][n] += D(item, subitem)

    gammaSort = {}
    gammaProbs = {}
    for key in gamma.keys():
        sg = gamma[key]
        ssg = sg.keys()
```

```
        ssg.sort(lambda a,b: cmp(sg[a], sg[b]))
        gammaSort[key] = ssg[:self.K]
        gammaProbs[str(ssg[:self.K])] = ssg[:self.K]

    deltaG = {}
    for key in gammaSort:
        fG = gammaSort[key]
        fG.sort()
        itemCounts = {}
        DList = {}
        count = 0
        for cls, ord in self.oss.gen_orders():
            tfG = copy.copy(fG)
            count += 1
            sfeats = ord.order
            # M = f_M
            vals = subfeats(sfeats, tfG)
            f = combine(tfG,vals)
            cf = cls + "|" + f
            if not itemCounts.has_key(f):
                itemCounts[f] = 0
            if not itemCounts.has_key(cf):
                itemCounts[cf] = 0
            itemCounts[f] += 1
            itemCounts[cf] += 1
            # M = f_M, F_i = f_i
            tfG.append(key)
            tfG.sort()
            vals = subfeats(sfeats, tfG)
            oldF = f
            f = combine(tfG,vals)
            DList[f] = oldF
            cf = cls + "|" + f
            if not itemCounts.has_key(f):
                itemCounts[f] = 0
            if not itemCounts.has_key(cf):
                itemCounts[cf] = 0
            itemCounts[f] += 1
            itemCounts[cf] += 1
        # Compute probabilities
        for x in itemCounts.keys():
            itemCounts[x] /= float(count)
        deltaG[key] = 0.0
        for fmfi in DList.keys():
            fm = DList[fmfi]
            subval = 0.0
            for cls in self.oss.classes:
                cfmfi = cls + "|" + fmfi
```

```
                cfm = cls + "|" + fm
                if itemCounts.has_key(cfmfi):
                    subval += D(cfmfi, cfm)
            subval = subval * itemCounts[fmfi]
            deltaG[key] += subval

    min_val = 1000000.0
    for key in deltaG:
        if deltaG[key] < min_val:
            min_val = deltaG[key]
            min_dG = key
    return min_dG
def center_dist_vec_diff(self, del_feats):
    oss_test = copy.deepcopy(self.oss)
    for feat in del_feats:
        oss_test.del_feature(feat)
    v1 = self.oss.center_rho_distance_vector()
    v2 = oss_test.center_rho_distance_vector()
    ans = 0.0
    for a,b in zip(v1,v2):
        ans += math.pow(a-b,2)
    return math.sqrt(ans)
def KL_dist(self, del_feats):
    self.oss2 = copy.deepcopy(self.oss)
    for feat in del_feats:
        self.oss2.del_feature(feat)
    val = self.oss.KL_dist(self.oss2)
    return val
def gen_KL_dist_start(self):
    self.oss_orig = copy.deepcopy(self.oss)
    self.populate_instances()
    F = self.instances
    for i in F:
        print i.cls, i.feats
    sys.exit()
    for feat in del_feats:
        self.oss.del_feature(feat)
    self.populate_instances_bool()
    self.oss = copy.deepcopy(self.oss_orig)
    Fg = self.instances
    all = []
    all.extend(F)
    all.extend(Fg)
    itemCounts = {}
    for item in all:
        f = str(item.feats)
        cf = item.cls + '|' + f
        if not itemCounts.has_key(f):
```

```
            itemCounts[f] = 0
        if not itemCounts.has_key(cf):
            itemCounts[cf] = 0
        itemCounts[f] += 1
        itemCounts[cf] += 1
    P = {}
    for k in itemCounts.keys():
        P[k] = itemCounts[k] / float(self.count)
    val = 0.0                                                    1230
    for item1, item2 in zip(F, Fg):
        f = str(item1.feats)
        fg = str(item2.feats)
        subval = 0.0
        for cls in self.oss.classes:
            cf = cls + '|' + f
            cfg = cls + '|' + fg
            if P.has_key(cf):
                Pcf = P[cf] / P[f]
                if P.has_key(cfg):                               1240
                    Pcfg = P[cfg] / P[fg]

                    subval += Pcf * math.log( Pcf / Pcfg )
        val += P[f] * subval
    return val
def gen_CDV(self, from_start=True):
    def euclid_dist(vec1, vec2):
        ans = 0.0
        for a,b in zip(vec1,vec2):
            ans += math.pow(a-b,2)                               1250
        return math.sqrt(ans)
    v1 = self.oss.center_rho_distance_vector()
    oss_prev = copy.deepcopy(self.oss)

    length = oss_prev.length
    while length > 2:
        feats = oss_prev.get_features()
        vmin = float(sys.maxint)
        print "# Dist: ",
        for feat in feats:                                       1260
            oss_test = copy.deepcopy(oss_prev)
            oss_test.del_feature(feat)
            v2 = oss_test.center_rho_distance_vector()
            v = euclid_dist(v1, v2)
            print v,
            if v < vmin:
                vmin = v
                v2min = v2
                fmin = feat
```

```
            print
            yield fmin
            oss_prev.del_feature(fmin)
            length = length - 1
            if not from_start:
                v1 = v2min
    def gen_Spoilers(self):
        def max_inv(dict):
            maxk = dict.keys()[0]
            max = dict[maxk]
            for k in dict.keys():
                if dict[k] > max:
                    max = dict[k]
                    maxk = k
            return maxk
        def delfeat(feat, diffs, invs):
            del diffs[feat]
            idict = invs[feat]
            for k in idict:
                diffs[k] -= idict[k]
            del invs[feat]
            for k in invs.keys():
                idict = invs[k]
                del idict[feat]
        self.oss.count_all_inversions()
        dc = self.oss.diff_count
        ic = self.oss.inv_count
        length = self.oss.length
        while length > 2:
            #print dc
            #print ic
            df = max_inv(dc)
            yield df
            delfeat(df, dc, ic)
            length -= 1


class GenOrder:
    def __init__(self):
        self.n = 6          # Length of ranks
        self.c = 3          # Number of classes
        self.m = 5          # Members of each class
        self.max_rc = 0.3 # Max rho concordance of initial classes
        self.min_rc = 0.6 # Minimum rho concordance of each class
        self.max_rcc = 0.8 # Maximum rho concordance of each class
        self.swaps = 5     # Maximum number of swaps used to generate new class members
    def unique(self, ranks):
        uniq = {}
```

```
        for rank in ranks:
            s = str(rank)
            if uniq.has_key(s):
                return False
            uniq[s] = 1
        return True
    def rand_interleave(self, r1, r2):
        ch_arr = []
        ans = []
        for i in range(len(r1)):
            ch_arr.append(1)
        for i in range(len(r2)):
            ch_arr.append(2)
        while ch_arr:
            elem = random.choice(ch_arr)
            if elem == 1:
                ans.append(r1.pop(0))
            else:
                ans.append(r2.pop(0))
            ch_arr.remove(elem)
        return ans
    def gen_redundant(self):
        def ptr2ord(p, o):
            newo = []
            for i in p:
                newo.append(o[i])
            return newo

        if self.n % 2 == 1:
            print "To generate redundant data, n must be even.\n"
            sys.exit()
        size = self.n / 2
        ac = AlphaCount()
        seed = ac.range(self.n)
        random.shuffle(seed)
        seed1 = seed[:size]
        seed2 = seed[size:]
        not_unique = True
        ptrs = range(size)
        while not_unique:
            rand_ranks = []
            for x in range(self.c):
                random.shuffle(ptrs)
                rand_ranks.append(copy.copy(ptrs))
            not_unique = not self.unique(rand_ranks)
        oss = OrderSetSet()
        cn = ac.range(self.c)
        cn = map(string.upper,cn)
```

```
        whole_set = OrderSetSet()
        for ptr, name in zip(rand_ranks, cn):
            os = OrderSet()
            s1 = ptr2ord(ptr, seed1)
            s2 = ptr2ord(ptr, seed2)                                    1370
            #print s1, s2
            for x in range(self.m):
                a1 = copy.copy(s1)
                a2 = copy.copy(s2)
                new = self.rand_interleave(a1, a2)
                ord = Order()
                ord.init_order(new)
                os.add_order(ord)
            whole_set.addOrderSet(os, name)
        return whole_set                                                1380
    def gen_known(self, known_portion):
        ac = AlphaCount()
        seed = ac.range(self.n)
        random.shuffle(seed)
        seed_known = seed[:known_portion]
        seed_unknown = seed[known_portion:]
        not_unique = True
        while not_unique:
            odrset = OrderSet()
            for x in range(self.c):                                     1390
                random.shuffle(seed_known)
                odr = Order()
                odr.init_order(seed_known)
                odrset.add_order(odr)
            not_unique = not self.unique(odrset.get_ranks())
        #for od in odrset.orders:
        #    sys.stderr.write("# " + str(od.get_order()) + '\n')
        #sys.stderr.write("Centers have been choosen, rc=" + str(rc) + "\n")

        oss = OrderSetSet()                                             1400
        cn = ac.range(odrset.members)
        cn = map(string.upper,cn)
        whole_set = OrderSetSet()
        for od, name in zip(odrset.orders, cn):
            os = OrderSet()
            for x in range(self.m):
                random.shuffle(seed_unknown)
                add_this = copy.copy(seed_unknown)
                new = self.rand_interleave(od.get_order(), add_this)
                ord = Order()                                           1410
                ord.init_order(new)
                os.add_order(ord)
            whole_set.addOrderSet(os, name)
```

```
        return whole_set
    def generate(self, V, R, E):
        size = V + R + E
        ac = AlphaCount()
        seed = ac.range(size)
        random.shuffle(seed)
        vS = seed[:V]                                                    1420
        rS = seed[V:V+R]
        eS = seed[V+R:]

        # Compose the V's
        not_unique = True
        while not_unique:
            rand_ords = []
            for x in range(self.c):
                random.shuffle(vS)
                rand_ords.append(copy.copy(vS))                          1430
            not_unique = not self.unique(rand_ords)
        # Generate class names
        cn = ac.range(self.c)
        cn = map(string.upper,cn)

        # Add in the R's
        init_set = {}
        for cls, od in zip(cn, rand_ords):
            init_set[cls] = []
            for i in range(self.m):                                      1440
                init_set[cls].append(copy.copy(od))

        for r in rS:
            pair_with = random.choice(vS)
            for cls in init_set.keys():
                ords = init_set[cls]
                for od in ords:
                    i = od.index(pair_with)
                    rnd = random.choice([0,1])
                    od.insert(i+rnd,r)                                   1450

        # Add in the V's
        whole_set = OrderSetSet()
        for cls in init_set.keys():
            os = OrderSet()
            ords = init_set[cls]
            for od in ords:
                random.shuffle(eS)
                add_this = copy.copy(eS)
                new = self.rand_interleave(od, add_this)                 1460
                ord = Order()
```

```
                ord.init_order(new)
                os.add_order(ord)
            whole_set.addOrderSet(os, cls)
        return whole_set
    def generate_random(self):
        def swap(rank, a, b):
            temp = rank[a]
            rank[a] = rank[b]
            rank[b] = temp
        def gen_new(odr, nswaps):
            newOdr = Order()
            rank = odr.get_order()
            swaps = random.randrange(nswaps)
            ln = odr.length
            for x in range(swaps):
                y = random.randrange(ln)
                z = random.randrange(ln)
                if (y != z):
                    swap(rank,y,z)
            newOdr.init_order(rank)
            return newOdr

        seq = range(self.n)
        rc = 100.0
        while rc > self.max_rc:
            odrset = OrderSet()
            for x in range(self.c):
                random.shuffle(seq)
                odr = Order()
                odr.init_rank(seq)
                odrset.add_order(odr)
            rc = odrset.rho_concordance()
        sys.stderr.write("Centers have been choosen, rc=" + str(rc) + "\n")

        ac = AlphaCount()
        cn = ac.range(odrset.members)
        cn = map(string.upper,cn)
        whole_set = OrderSetSet()
        for odr, name in zip(odrset.orders, cn):
            os = OrderSet()
            os.add_order(odr)
            whole_set.addOrderSet(os,name)
        for key in whole_set.classes:
            os = whole_set.ordersets[key]
            #sys.stderr.write(str(os.orders[0].get_rank()) + "\n")
            while os.members < self.m:
                newOdr = gen_new(os.orders[0], self.swaps)
                os.add_order(newOdr)
```

```
                rc = os.rho_concordance()
                if rc < self.min_rc or rc > self.max_rcc:
                    #sys.stderr.write("Order rejected, rc=" + str(rc) + "\n")
                    os.del_last()
                else:
                    sys.stderr.write("Order accepted, rc=" + str(rc) + "\n")
                    #sys.stderr.write(str(newOdr.get_order()) + "\n")
            sys.stderr.write("Class " + key + " complete\n")
        return whole_set

if __name__ == '__main__':
    # Test the module
    o1 = Order()
    o2 = Order()

    #o1.init_order_rank({'x':20, 'y':40, 'z':60})
    #o2.init_order_rank({'x':1, 'y':2, 'z':3})

    #oset = OrderSet()
    #oset.add_order(o1)
    #oset.add_order(o2)

    #o3 = oset.rho_center()
    #print o1.order, o1.rank
    #print o2.order, o2.rank
    #print o3.order, o3.rank


    o1.init_rank([3.5,5.5,1,5.5,7,2,3.5,8])
    o2.init_rank([7,2,1,8,4.5,3,4.5,6])

    print o1.get_order(), o1.get_rank()
    print o2.get_order(), o2.get_rank()
    print o1.tau_dist(o2), "Kd:", o1.kendall_dist(o2), "Num ties:", o1.num_ties, o2.num_ties

    o1.reverse()
    print o1.get_order(), o1.get_rank()
    print o2.get_order(), o2.get_rank()
    print o1.tau_dist(o2), "Kd:", o1.kendall_dist(o2), "Num ties:", o1.num_ties, o2.num_ties

    #print o1.get_rank()
    #print o2.get_rank()

    #inv = o1.kendall_dist_ties(o2)
    #print inv, o1.diff_count

    #inv = o1.kendall_dist_noties(o2)
    #print inv, o1.diff_count
```

```
#genOrd = GenOrder()
#genOrd.n = 6
#genOrd.m = 3
#genOrd.c = 3
#oss = genOrd.gen_known(3)
#oss = genOrd.generate(3,1,2)
#feats = oss.get_features()
#print "feats",
#for feat in feats:
#    print feat,
#print
#oss.print_data()
#sys.exit()
#print "#\n# CDV"
#fs = FeatureSelect()
#fs.add_OrderSetSet(oss)
#for feat in fs.gen_CDV():
#    print "# ", feat

#print "#\n#"
#print "# Spoilers"
#while oss.length > 2:
#    spoilers = oss.tau_compare_all()
#    print "# ", spoilers
#    sk = spoilers.keys()
#    sk.sort(lambda x,y: cmp(spoilers[y],spoilers[x]))
#    feat = sk[0]
#    oss.del_feature(feat)
#    print "# ", feat

#fs = FeatureSelect()
#fs.add_OrderSetSet(oss)
#for feat in fs.gen_Spoilers():
#    print "# ", feat

#pc = ProbCount()
#for cls, ord in oss.gen_orders():
#    pc.add_instance(cls,ord.get_rank())
#pc.print_data()
#pc.count()
```

A.2 C Code

```
#include <stdlib.h>
#include <stdio.h>
#include <getopt.h>
#include <math.h>
#include <limits.h>
/*#include <mcheck.h>*/
#include "lex.h"
#include "order_fs.h"

struct option long_options[] =
{
  {"spoilers",      ARG_NONE, NULL, 's'},
  {"center-vector", ARG_NONE, NULL, 'c'},
  {"kl-dist",       ARG_NONE, NULL, 'k'},
  {"help",          ARG_NONE, NULL, 'k'},
  {NULL,            0,        NULL, 0}
};

/* Flex variables */
extern int yylex();
extern FILE *yyin;
extern int yyrestart(FILE*);
extern char *yytext;

/* stupid global var because not every system has qsort_r */
double *compare_rank;

int main (int argc, char *argv[]) {
  char *filename, *progname;
  int opt_index, option, errcnt=0;
  char fs_mode='s';
  OrderSetSet *orig_oss;
  OrderSetSet *copy_oss;

  progname=(char *)malloc((strlen(argv[0])+1)*sizeof(char));
  strcpy(progname, argv[0]);

  while ((option = getopt_long(argc, argv, "sckh", long_options, &opt_index)) != EOF) {
    if (option == 0)
      option = long_options[opt_index].val;

    switch(option) {
    case 's':
      fs_mode='s'; break;
    case 'c':
      fs_mode='c'; break;
    case 'k':
      fs_mode='k'; break;
```

```
    case 'h':
      return usage(progname);
    default:
      errcnt++; break;
    }
  }
  if (errcnt > 0) {
    fprintf (stderr, "Try `%s --help' for more information\n", argv[0]);
    return EXIT_FAILURE;
  }

  if (optind >= argc) {
    usage(progname);
    return EXIT_FAILURE;
  }

  filename=(char *)malloc((strlen(argv[optind])+1)*sizeof(char));
  strcpy(filename, argv[optind]);

  orig_oss = load_file(filename);
  /* print_OrderSetSet(orig_oss); */

  switch(fs_mode) {
  case 's':
    genSpoilers(orig_oss);
    break;
  case 'c':
    copy_oss = load_file(filename);
    /* mtrace(); */
    genCenterDistanceVector(orig_oss, copy_oss);
    break;
  case 'k':
    printf("Not implmented yet!\n"); break;
    break;
  }

  /*
  muntrace();
  free_OrderSetSet(orig_oss);
  free_OrderSetSet(copy_oss);
  */
  free(progname);
  free(filename);
  return EXIT_SUCCESS;
}

int genCenterDistanceVector(OrderSetSet *oss, OrderSetSet *copy) {
  double *cdv_orig, *cdv_current;
```

```
  double dist, min_dist;
  int min_feat;
  int len, i, j;
                                                                        100
  len = oss->num_classes * ( oss->num_classes - 1 ) / 2;
  cdv_orig = CenterDistanceVector(oss);

  while (oss->num_feats > 2) {

    min_dist = LONG_MAX;
    for (i=0; i<oss->num_feats; i++) {
      copy_OrderSetSet(oss, copy);
      del_feature(i, copy);
      cdv_current = CenterDistanceVector(copy);                         110
      dist = euclid_distance(len, cdv_orig, cdv_current);
      if (dist < min_dist) {
        min_dist = dist;
        min_feat = i;
      }
      free(cdv_current);
    }
    printf("%s\n", oss->features[min_feat]);
    del_feature(min_feat, oss);
  }                                                                     120

  free(cdv_orig);
  return EXIT_SUCCESS;
}

double *CenterDistanceVector(OrderSetSet *oss) {
  Order **centers;
  double *cdv;
  int vec_length;
  int i, j;                                                             130
  int cent_num = 0;

  /* Initialize central rank vectors */
  vec_length = oss->num_classes * ( oss->num_classes - 1 ) / 2;
  centers = (Order **)malloc(sizeof(Order *) * (oss->num_classes));
  for (i=0; i<oss->num_classes; i++) {
    centers[i] = rho_center(&oss->osets[i]);
  }

  /* Compute CDV */                                                     140
  cdv = (double *)malloc(sizeof(double)*vec_length);
  for (i=0; i < (oss->num_classes-1); i++) {
    for (j=i+1; j < oss->num_classes; j++) {
      cdv[cent_num++] = rho_distance(centers[i], centers[j]);
```

```
    }
  }

  /* Free Memory */
  for (i=0; i<oss->num_classes; i++) {
    free(centers[i]->rank);
    free(centers[i]->order);
    free(centers[i]);
  }
  free(centers);

  return cdv;
}

int genSpoilers(OrderSetSet *oss) {
  int val;

  while (oss->num_feats > 2) {
    val = get_highest_spoiler(oss);
    /* printf("Del: %s\tFeats Left:%d\n", oss->features[val], oss->num_feats-1); */
    printf("%s\n", oss->features[val]);
    del_feature(val, oss);
    /* print_OrderSetSet(oss); */
  }
  return EXIT_SUCCESS;
}

double rho_distance(Order *o1, Order *o2) {
  int i;
  double ans = 0.0;

  for (i=0; i < o1->length; i++) {
    ans += pow((o1->rank[i] - o2->rank[i]), 2.0);
  }

  return 1.0 - ((6.0*ans)/(pow(o1->length, 3) - o1->length));
}

double euclid_distance(int length, double *r1, double *r2) {
  int i;
  double ans = 0.0;

  for (i=0; i<length; i++) {
    ans += pow((r1[i]-r2[i]), 2.0);
  }

  return sqrt(ans);
}
```

```
int ord_compare(const int *a, const int *b) {

  if (compare_rank[*a] < compare_rank[*b] ) {
    return −1;
  }
  if (compare_rank[*a] > compare_rank[*b] ) {
    return 1;                                                    200
  }
  return 0;
}

int Count(int sort_me[], int l, int m, int r, double pos[], int spoilers[]) {
  int *left, *right;
  int l_len, r_len;
  int lp=0, rp=0, right_merged=0;
  int idx, ans;
                                                                 210
  /* Create left array */
  l_len = m−l+1;
  left = (int *)malloc(sizeof(int)*l_len);
  memcpy(left, &sort_me[l], sizeof(int)*l_len);

  /* Create right array */
  r_len = r−m;
  right = (int *)malloc(sizeof(int)*r_len);
  memcpy(right, &sort_me[m+1], sizeof(int)*r_len);
                                                                 220
  for(idx=l; idx<=r; idx++) {
    if (lp >= l_len) {
      sort_me[idx] = right[rp++];
    } else if (rp >= r_len) {
      ans = left[lp++];
      sort_me[idx] = ans;
      spoilers[ans] += right_merged;
    } else {
      compare_rank = pos;
      ans = ord_compare(&left[lp], &right[rp]);                  230
      switch(ans) {
      case −1:
        ans = left[lp++];
        sort_me[idx] = ans;
        spoilers[ans] += right_merged;
        break;
      case 0:
        sort_me[idx] = left[lp++];
        break;
      case 1:                                                    240
```

```
        ans = right[rp++];
        sort_me[idx] = ans;
        spoilers[ans] += (l_len - lp);
        right_merged++;
        break;
      }
    }
  }

  free(left);
  free(right);
}

int QuickCount(int sort_me[], int left, int right, double pos[], int spoilers[]) {
  int mid;

  if (left < right) {
    mid = (left + right) / 2;
    QuickCount(sort_me, left, mid, pos, spoilers);
    QuickCount(sort_me, mid+1, right, pos, spoilers);
    Count(sort_me, left, mid, right, pos, spoilers);
    return EXIT_SUCCESS;
  }
  return EXIT_SUCCESS;
}

int count_spoilers(Order o1, Order o2, int *spoiler_count) {
  if (o1.has_ties || o2.has_ties) {
    return count_spoilers_ties(o1, o2, spoiler_count);
  } else {
    return count_spoilers_noties(o1, o2, spoiler_count);
  }
}

int count_spoilers_ties(Order o1, Order o2, int *spoiler_count) {
  int i, j;

  for (i=0; i<o1.length; i++) {
    for (j=i+1; j<o1.length; j++) {
      if ((o1.rank[i] != o1.rank[j]) || (o2.rank[i] != o2.rank[j])) {
        if ((o1.rank[i] < o1.rank[j]) != (o2.rank[i] < o2.rank[j])) {
          spoiler_count[i]++;
          spoiler_count[j]++;
        } else {
          spoiler_count[i]--;
          spoiler_count[j]--;
        }
      }
```

```
    }
  }
  return EXIT_SUCCESS;
}

int count_spoilers_noties(Order o1, Order o2, int *spoiler_count) {
  int len;
  int *srt;

  /* initialize working sort vector */
  len = o1.length;
  srt = (int *)malloc(len*sizeof(int));
  memcpy(srt, o1.order, sizeof(int)*len);

  QuickCount(srt, 0, len-1, o2.rank, spoiler_count);

  free(srt);
  return EXIT_SUCCESS;
}

Order *rho_center(OrderSet *os) {
  int i, j;
  Order *center;

  center = (Order *)malloc(sizeof(Order));
  center->length = os->orders[0].length;
  center->rank = (double *)malloc(sizeof(double)*center->length);
  center->order = (int *)malloc(sizeof(int)*center->length);
  for (i=0; i<center->length; i++) {
    center->rank[i] = 0.0;
  }

  /* add all ranks */
  for (i=0; i<os->num_ranks; i++) {
    for (j=0; j<center->length; j++) {
      center->rank[j] += os->orders[i].rank[j];
    }
  }

  /* normalize ranks */
  make_order(center);
  normalize_rank(center);

  return center;
}

int max_index(int length, int vec[]) {
  int i, maxV;
```

```
  int maxI;

  maxV = vec[0];
  maxI = 0;
  for (i=1; i<length; i++) {
    if (vec[i] > maxV) {
      maxV = vec[i];
      maxI = i;
    }
  }
  return maxI;
 }

int get_highest_spoiler(OrderSetSet *oss) {
  int class, len, i;
  int ord1, ord2;
  int *spoilers;
  int max;

  /* initialize spoiler counts */
  len = oss->num_feats;
  spoilers = (int *)malloc(len*sizeof(int));
  for (i=0; i<len; i++) {
    spoilers[i] = 0;
  }

  for(class=0; class < oss->num_classes; class++) {
    for(ord1=0; ord1 < oss->osets[class].num_ranks-1; ord1++) {
      for(ord2=ord1+1; ord2 < oss->osets[class].num_ranks; ord2++) {
        count_spoilers(oss->osets[class].orders[ord1], oss->osets[class].orders[ord2], spoilers);
      }
    }
  }

  /*
  printf("Spoilers: ");
  for (i=0; i<len; i++) {
    printf("%3d ", spoilers[i]);
  }
  printf("\n");
  */

  max = max_index(len, spoilers);
  free(spoilers);
  return max;
}

OrderSetSet* load_file(char *fname) {
```

```
int type;
int num_feats=0;
int cur_feat=0;
int cur_ord=0;
int rank_pos;
double rank_val;                                                    390
int class;
OrderSetSet *oss;
double iptr;

if ((yyin=fopen(fname, "r" )) == NULL ) {
  printf("The file %s could not be opened\n", fname);
  return NULL;
}

/* get initial length */                                            400
while (yylex() != FEATS) { /* skip junk */ }

while (yylex() == WORD) {
  num_feats++;
}

/* create the structure */
oss=(OrderSetSet *)malloc(sizeof(OrderSetSet));
oss->num_classes = 0;
oss->num_feats = num_feats;                                         410
oss->features = (char **)malloc(num_feats*sizeof(char *));

rewind(yyin);
yyrestart(yyin);

while (type = yylex()) {
  switch(type) {
  case FEATS:
    while ((type = yylex()) == WORD) {
      oss->features[cur_feat] = (char *)malloc(MAX_FEATURE_STRING*sizeof(char));   420
      strcpy(oss->features[cur_feat], yytext);
      cur_feat++;
    }
    break;
  case CLASS:
    sscanf(yytext, "%d", &class);
    rank_pos = 0;
    if (oss->num_classes < class+1) {
      oss->num_classes++;
      cur_ord = 0;                                                  430
    }
    oss->osets[class].orders[cur_ord].length = num_feats;
```

```
      oss->osets[class].orders[cur_ord].has_ties = 0;
      oss->osets[class].orders[cur_ord].rank = (double *)malloc(num_feats*sizeof(double));
      oss->osets[class].orders[cur_ord].order = (int *)malloc(num_feats*sizeof(int));
      while ((type = yylex()) == NUMBER) {
        sscanf(yytext, "%lf", &rank_val);
        oss->osets[class].orders[cur_ord].rank[rank_pos] = rank_val;
        if (modf(rank_val, &iptr) > 0) {
          oss->osets[class].orders[cur_ord].has_ties = 1;                    440
        }
        rank_pos++;
      }
      make_order(&oss->osets[class].orders[cur_ord]);
      cur_ord++;
      oss->osets[class].num_ranks = cur_ord;
      break;
    case EOL:
      printf("EOL\n");
      break;                                                                 450
    default:
      printf("How did I get here?\n");
    }
  }

  fclose(yyin);
  return oss;
}

int copy_OrderSetSet(OrderSetSet *oss, OrderSetSet *new_oss) {               460
  int i, j, num_feats;

  num_feats = oss->num_feats;

  /* create the structure */
  new_oss->num_classes = oss->num_classes;
  new_oss->num_feats = oss->num_feats;

  for (i=0; i<num_feats; i++) {
    strcpy(new_oss->features[i], oss->features[i]);                          470
  }

  for(i=0; i < oss->num_classes; i++) {
    new_oss->osets[i].num_ranks = oss->osets[i].num_ranks;
    for(j=0; j < oss->osets[i].num_ranks; j++) {
      new_oss->osets[i].orders[j].length = num_feats;
      memcpy(new_oss->osets[i].orders[j].rank, oss->osets[i].orders[j].rank, sizeof(double)*num_feats);
      memcpy(new_oss->osets[i].orders[j].order, oss->osets[i].orders[j].order, sizeof(int)*num_feats);
    }
  }                                                                          480
```

```
  return EXIT_SUCCESS;
}

int free_OrderSetSet(OrderSetSet *oss) {
  int i, j;

  for(i=0; i < oss->num_classes; i++) {
    for(j=0; j < oss->osets[i].num_ranks; j++) {
      free(oss->osets[i].orders[j].rank);                                    490
      free(oss->osets[i].orders[j].order);
    }
  }

  for (i=0; i<oss->num_feats; i++) {
    free(oss->features[i]);
  }

  /* delete the structure */
  free(oss->features);                                                       500
  free(oss);

  return EXIT_SUCCESS;
}

int del_feature(int feat, OrderSetSet *oss) {
  int class;
  int ord_num;
  int f;
  double val;                                                                510
  char *tmp_str;
  Order *ord;
  int passed=0;

  /* free(oss->features[feat]); */
  tmp_str = oss->features[feat];
  for(f=feat; f < (oss->num_feats-1); f++) {
    oss->features[f] = oss->features[f+1];
  }
  oss->features[oss->num_feats] = tmp_str;                                   520
  oss->num_feats--;

  for(class=0; class < oss->num_classes; class++) {
    for(ord_num=0; ord_num < oss->osets[class].num_ranks; ord_num++) {
      ord = &oss->osets[class].orders[ord_num];
      ord->length--;
      val = ord->rank[feat];
      /* delete from rank */
```

```
      for (f=0; f<ord->length; f++) {
        if (f>=feat) ord->rank[f] = ord->rank[f+1];
        if (ord->rank[f] == val) ord->rank[f] -= 0.5;
        else if (ord->rank[f] > val) ord->rank[f] -= 1.0;
      }
      /* delete from order */
      for (f=0; f<ord->length; f++) {
        if ((passed) || (ord->order[f]==feat)) {
          ord->order[f] = ord->order[f+1];
          passed = 1;
        }
        if (ord->order[f] > feat) ord->order[f] -= 1;
      }
      passed = 0;
    }
  }
  return EXIT_SUCCESS;
}

int make_order(Order *ord) {
  int i;

  for(i=0; i< ord->length; i++) {
    ord->order[i] = i;
  }
  compare_rank = ord->rank;
  qsort(ord->order, ord->length, sizeof(int), (void *)ord_compare);

  return EXIT_SUCCESS;
}

int normalize_rank(Order *ord) {
  int i, j;
  int p1, p2;
  int start=0;
  int stop;
  double val;


  for (i=1; i<ord->length; i++) {
    p1 = ord->order[i-1];
    p2 = ord->order[i];
    if (ord->rank[p1] < ord->rank[p2]) {
      stop = i;
      val = (start + stop + 1.0) / 2.0;
      for (j=start; j<stop; j++) {
        ord->rank[ord->order[j]] = val;
      }
```

```
      start = i;
    }
  }

  stop = i;
  val = (start + stop + 1.0) / 2.0;
  for (j=start; j<stop; j++) {
    ord->rank[ord->order[j]] = val;
  }

  return EXIT_SUCCESS;
}

int print_Order(Order *ord) {
  int i;

  printf("Rank:");
  for (i=0; i<ord->length; i++) {
    printf("%3.1f ", ord->rank[i]);
  }
  printf("\tOrder:");
  for (i=0; i<ord->length; i++) {
    printf("%d ", ord->order[i]);
  }
  printf("\n");
  return EXIT_SUCCESS;
}


int print_OrderSetSet(OrderSetSet *oss) {
  int class;
  int ord;
  int feat;

  printf("Features: ");
  for(feat=0; feat < oss->num_feats; feat++) {
    printf("%s ", oss->features[feat]);
  }
  printf("\n");

  for(class=0; class < oss->num_classes; class++) {
    printf("Class: %d\n", class);
    for(ord=0; ord < oss->osets[class].num_ranks; ord++) {
      printf("%4d, rank: ", ord);
      for(feat=0; feat < oss->num_feats; feat++) {
        printf("%3.1f ", oss->osets[class].orders[ord].rank[feat]);
      }
      printf("  order: ", class, ord);
```

```
      for(feat=0; feat < oss->num_feats; feat++) {
        printf("%d ", oss->osets[class].orders[ord].order[feat]);
      }
      printf("\n");
    }
  }
  return EXIT_SUCCESS;
}

int usage(char *progname) {
  fprintf(stderr, "Usage: %s [options] filename\n\n", progname);
  fprintf(stderr, "     -s | --spoilers\n");
  fprintf(stderr, "     -c | --center-vector\n");
  fprintf(stderr, "     -k | --kl-dist\n");
  fprintf(stderr, "     -h | --help\n");

  return EXIT_SUCCESS;
}
```

Bibliography

[1] M.N. Arbeitman, E.E.M. Furlong, F. Imam, E. Johnson, B.H. Null, B.S. Baker, M.A. Krasnow, M.P. Scott, R.W. Davis, and K.P. White. Gene expression during the life cycle of *Drosophilia melanogaster*. *Science*, 297:2270–2275, September 2002.

[2] K.J. Arrow. *Social Choice and Individual Values*. Yale University Press, second edition, Oct 1970.

[3] Andreas D. Baxevanis and B. F. Francis Ouellette, editors. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Wiley-Interscience, New york, N.Y., second edition, 2001.

[4] A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):245–271, 1997.

[5] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Computational Learing Theory*, pages 144–152, 1992. URL `citeseer.ist.psu.edu/boser92training.html`.

[6] S. Brenner, M. Johnson, J. Bridgham, G. Golda, D.H. Lloyd, D. Johnson, S. Luo, S. McCurdy, M. Foy, M. Ewan, R. Roth, D. George, S. Eletr, G. Albrecht, E. Vermaas, S.R. Williams, K. Moon, T. Burcham, M. Pallas, R.B. DuBridge, J. Kirchner, K. Fearon, J. Mao, , and K. Corcoran. Gene expression analysis by massively parallel signature sequencing (MPSS) on microbead arrays. *Nature Biotechnology*, 18:630–634, 2000.

[7] S. Brenner, S.R. Williams, E.H. Vermaas, T. Storck, K. Moon, C. McCollum, J.I. Mao, S. Luo, J.J. Kirchner, S. Eletr, R.B. DuBridge, T. Burcham, , and G. Albrecht. *In vitro* cloning of complex mixtures of DNA on microbeads: physical separation of differentially expressed cDNAs. *Proceedings of the National Academy of Science USA*, 97, 2000.

[8] A. Malcolm Campbell and Laurie J. Heyer, editors. *Discovering Genomics, Proteomics, and Bioinformatics*. Benjamin Cummings, San Francisco, C.A., 2002.

[9] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[10] G. F. Chen and M. Inouye. Suppression of the negative effect of minor arginine codons on gene expression; preferential usage of minor codons within the first 25 codons of the *Escherichia coli* genes. *Nucleic Acids Research*, 18(6):1465–1473, 1990.

[11] W.W. Cohen, R.E. Schapire, and Y. Singer. Learning to order things. *Journal of Artificial Intelligence Research*, Vol. 10:pages 243–270, 1999.

[12] S. J. Coughlan, V. Agrawal, and B. C. Meyers. A comparison of global gene expression measurement technologies in *Arabidopsis thaliana*. *Comparative and Functional Genomics*, 5:245–252, 2004.

[13] T.M. Cover and J. Thomas. *Elements of Information Theory*. John Wiley and Sons, 1991.

[14] J. Demsar and B. Zupan. Orange: From experimental machine learning to interactive data mining, white paper. White paper, Faculty of Computer and Information Science, University of Ljubljana, 2004. Software available at `http://www.ailab.si/orange`.

[15] Peter Fishburn. Binary probabilities induced by rankings. *SIAM Journal on Discrete Mathematics*, 3(4):478–488, 1990.

[16] Adriana Garay-Arroyo, José M. Colmenero-Flores, Alejandro Garciarrubio, and Alejandra A. Covarrubias. Highly hydrophilic proteins in prokaryotes and eukaryotes are common during conditions of water deficit. *The Journal of Biological Chemistry*, 275(8):5668–5674, feb 2000.

[17] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, May 2000.

[18] A. Gionis, T. Kujala, and H. Mannila. Fragments of order. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 129–136. ACM Press, 2003.

[19] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, November-December 1984.

[20] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[21] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Satistical Learning*. Springer-Verlag, 2001.

[22] T. Kamishima. Nantonac collaborative filtering: Recommendation based on order responses. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 583–588. ACM Press, 2003.

[23] T. Kamishima and S. Akaho. Learning from order examples. In *IEEE International Conference on Data Mining*, pages 645–648. IEEE Computer Society, 2002.

[24] Samuel Karlin and Jan Mrázek. Predicted highly expressed genes of diverse prokaryotic genomes. *Journal of Bacteriology*, 182(18):5238–5250, sep 2000.

[25] Samuel Karlin, Jan Mrázek, Allan Cambell, and Dale Kaiser. Characterizations of highly expressed genes of four fast-growing bacteria. *Journal of Bacteriology*, 183(17):5025–5040, aug 2001.

[26] M. Kendall and J.D. Gibbons. *Rank Correlation Methods*. Oxford University Press, 1990.

[27] B. S. Kim, S. Y. Rha, G. B. Cho, and H. C. Chung. Spearman's footrule as a measure of cDNA microarray reproducibility. *Genomics*, 84:441–448, 2004.

[28] Donald E. Knuth. *The Art of Computer Programming*, volume 3 - Sorting and Searching, pages 11–22. Addison Wesley, second edition, 1998.

[29] D. Koller and M. Sahami. Toward optimal feature selection. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'96)*, pages 284–292, July 1996.

[30] Jack Kyte and Russel F. Doolittle. A simple method for displaying the hydropathic character of a protein. *Journal of Molecular Biology*, 157:105–132, 1982.

[31] M. Last, A. Kandel, and O Maimon. Information-theoretic algorithm for feature selection. *Pattern Recognition Letters*, 22(6/7):799–811, 2001.

[32] G. Lebanon and J. Lafferty. Cranking: Combining rankings using conditional probability models on permutations. In *ICML*, pages 363–370, 2002.

[33] G. Lebanon and J. Lafferty. Conditional models on the ranking poset. In *Advances in Neural Information Processing Systems 15*, pages 431–438, 2003.

[34] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1988.

[35] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge discovery and Data Mining*, pages 161–168. ACM Press, 2000.

[36] John I. Marden. *Analyzing and Modeling Rank Data*. Chapman & Hall, London, UK, first edition, 1995.

[37] B. C. Meyers, D. K. Lee, T. H. Vu, S. S. Tej, S. B. Edberg, M. Matvienko, and L. D. Tindell. *Arabidopsis* MPSS: An online resource for quantitative expression analysis. *Plant Physiology*, 135:801–813, June 2004.

[38] B. C. Meyers, S. S. Tej, T. H. Vu, C. D. Haudenschild, V. Agrawal, S. B. Edberg, H. Ghazal, and S. Decola. The use of MPSS for whole-genome transcription analysis in *Arabidopsis*. *Genome Research*, 14(8):1641–1653, August 2004. Data availiable at `http://mpss.udel.edu/at/`.

[39] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[40] Bernard Monjardet. Concordance between two linear orders: The Spearman and Kendall coefficients revisited. *Journal of Classification*, 14:269–295, 1997.

[41] Jan Mrázek, Devaki Bhaya, Arthur R. Grossman, and Samuel Karlin. Highly expressed and alien genes of the *synechocystis* genome. *Nucleic Acids Research*, 29(7):1590–1601, 2001.

[42] Scott D. Patterson. Mass spectrometry and proteomics. *Physiological Genomics*, 2(2):59–65, March 2000.

[43] Naren Ramakrishnan and Ananth Y. Grama. Data mining applications in bioinformatics. In Robert L. Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and Raju R. Namburu, editors, *Data Mining for Scientific and Engineering Applications*, pages 125–140. Kluwer Acedemic Publishers, 2001.

[44] H.A. Rothe. *Sammlung combinatorisch-analytischer Abhandlungen*, volume 2, pages 263–305. Leipzig, 1800.

[45] Y. Sai, Y. Yao, and N. Zhong. Data analysis and mining in ordered information tables. In *Proceedings of the IEEE International Conference on Data Mining*, pages 497–504, 2001.

[46] A. Sen. Social choice theory. In K.J. Arrow and M.D. Intriligator, editors, *Handbook of Mathematical Economics*, volume 3, chapter 22. Elsevier, 1981.

[47] D. J. Slotta, L. S. Heath, N. Ramakrishnan, R. Helm, and M. Potts. Clustering mass spectrometry data using order statistics. *Proteomics*, 3(9):1687–1691, September 2003.

[48] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101, 1904.

[49] P.T. Spellman, G. Sherlock, M.Q. Zhang, V.R. Iyer, K. Anders, M.B. Eisen, P.O. Brown, D. Botstein, and B. Futcher. Comprehensive identification of cell cycle–regulated genes of the yeast *Saccharomyces cerevisiae* by microarray hybridization. *Molecular Biology of the Cell*, 9:3273–3297, December 1998.

[50] Han Tao, Christoph Bausch, Craig Richmond, Frederick R. Blattner, and Tyrrell Conway. Functional genomics: Expression analysis of *Escherichia coli* growing on minimal and rich media. *Journal of Bacteriology*, 181(20):6425–6440, oct 1999.

[51] Roman L. Tatusov, Michael Y. Galperin, Darren A. Natale, and Eugene V. Koonin. The COG database: a tool for genome-scale analysis of protein functions and evolution. *Nucleic Acids Research*, 28(1):33–36, 2000.

[52] Roman L. Tatusov, Eugene V. Koonin, and David J. Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–637, 1997.

[53] R. A. VanBogelen, K. Z. Abshire, A. Pertsemlidis, R. L. Clark, and F. C. Neidhardt., editors. Escherichia coli *and* Salmonella*: cellular and molecular biology*, chapter 115: Gene-protein database of *Escherichia coli* K-12, edition 6, pages 2067–2117. ASM Press, Washington, D.C., 2nd edition, 1996.

[54] Russel D. Wolfinger, Greg Gibson, Elizabeth D. Wolfinger, Lee Bennett, Hisham Hamadeh, Pierre Bushel, Cynthia Afshari, and Richard S. Paules. Assessing gene significance from cdna microarray expression data via mixed models. *Journal of Computational Biology*, 8(6):625–637, 2001.

[55] H. Zhang, C. Yu, and B. Singer. Cell and tumor classification using gene expression data: Construction of forests. *PNAS*, 100(7):4168–4172, April 2003.

[56] X.L. Zhang, H. Begleiter, B. Porjesz, W. Wang, and A. Litke. Event related potentials during object recognition tasks. *Brain Research Bulletin*, 38(6):531–538, 1995.

Vita

Douglas J. Slotta

Research Interests

Bioinformatics, algorithms, scientific programming, parallel computing

Education

Virginia Polytechnic Institute and State University Blacksburg, Virginia

Ph.D., Computer Science and Applications, May 2005

Dissertation Topic: "Evaluating Biological Data Using Rank Correlation Methods"

Advisor: Lenwood S. Heath

M.S., Computer Science and Applications, May 2001

Thesis Topic: "Structural Design Using Cellular Automata"

Advisor: Layne T. Watson

Ohio Northern University Ada, Ohio

B.S., Computer Science, May, 1997

Academic Experience

Virginia Polytechnic Institute and State University Blacksburg, Virginia

Graduate Student **January, 1999 - present**
Experience included Ph.D. and Masters level research, course work, research, and consulting projects.

Instructor **August 2000 - August, 2001**
Taught CS1004, Computer Literacy and several lab sections of CS2204 Unix.

Teaching Assistant **January, 1999 - 2002**
Duties at various times have included office hours and leading weekly computer lab exercises.

Publications

"Convergence analysis for cellular automata applied to truss design," Douglas J. Slotta, Brian Tatting, Layne T. Watson, Zafer Gürdal, S. Missoum, Engineering Computations: Int J for Computer-Aided Engineering, vol. 19, iss. 8, pp. 953-969(17)

"Computational Approaches to Combining Predictive Biological Models," Douglas J. Slotta, Lenwood S. Heath, Naren Ramakrishnan, Rich Helm, and Malcolm Potts, Proceedings of the High Performance Computing Symposium, Advanced Simulation Technologies Conference (HPC 2002), 2002, pp. 75-80.

"Clustering Mass Spectrometry Data using Order Statistics," Douglas J. Slotta, Lenwood S. Heath, Naren Ramakrishnan, Rich Helm, and Malcolm Potts, Proteomics 3 (9), 2003, pp. 1687-1691.

Papers in review

"Algorithms for Feature Selection in Rank Order Spaces," Douglas J. Slotta, John Paul C. Vergara, Naren Ramakrishnan, and Lenwood S. Heath, Data Mining and Knowledge Discovery, accepted with revisions March 2005.

"Evaluating Arabidopsis MPSS Using Rank Correlation Methods," Douglas J. Slotta and Lenwood S. Heath.

Conference Presentations

"Clustering Mass Spectrometry Data using Order Statistics," Douglas J. Slotta, Lenwood S. Heath, and Naren Ramakrishnan. First Annual Proteomics Data Mining Conference, Duke University, Durham, NC, September, 2002.

"Structural Optimization Using Cellular Automata," Douglas J. Slotta, Brian Tatting, Zafer Gürdal, and Layne T. Watson, First SIAM conference Computational Science and Engineering, September 2000.

Professional Experience

Incode Corporation (now Honeywell POMS) Blacksburg, Virginia

Programmer **June, 1997 - October, 1999**

Maintained software for Pharmaceutical manufacturing process control. Designed software interface for Programmable Logic Controllers.

End User Computing Toledo, Ohio

Programmer **January, 1996 - May, 1997**

Consulting Firm. Created custom database solutions for clients.

United States Marine Corps

Maintenance Management Specialist **July, 1988 - July, 1992**

Performed computerized maintenance management. Served in Operation Desert Storm/Shield. Awarded Navy Achievement Medal for performance of duties during Operation Desert Shield. Also cleared minefields.