Model Monitor User's Guide version 1.0

Troy Raeder
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
traeder@nd.edu

December 10, 2008

1 Introduction

Model Monitor (M^2) is a software tool for studying the effect of distribution shifts the performance of machine learning algorithms. Most practical work in data mining assumes that data distributions are *stationary*, which is to say they do not change over time. As such, classifiers tend to be most effective when the data on which they are evaluated (*test data*) is similar to the data on which the model was built (*training data*). In real life, however, shifts in distribution do occur, and a model that is robust to changes in distribution will be significantly more valuable in the long run than one which is not.

With this fact in mind, we designed Model Monitor around two basic tasks. First, it can be used to determine whether a distribution shift has occurred. Given separate train and test datasets (or a single dataset with a train-test split), Model Monitor will test each individual attribute of the dataset for distribution shift, using methods from [2], and report any features that have changed.

Second, Model Monitor can evaluate the robustness of a particular classifier to certain types of distribution shift. Using any commandline classifier, or a classifier from the WEKA machine learning environment, Model Monitor will run the classifier and report its performance under any one of a number of chosen performance measures or loss measures. In this way, researchers can better design classifiers that are resistant to distribution shifts, or practitioners can choose a model that performs well under an anticipated set of future conditions.

2 Installation and System Requirements

Model Monitor is written in Java, which means that it will run on any operating system. This section describes how to install and run the software, and acknowledges the third-party libraries that we used.

2.1 Installing and Running

To install Model Monitor, download the gzipped tar file from our website and unzip it into any directory. To run the program, type `java -jar m2.jar` from a terminal, within the directory where you installed it. To make use of Weka classifiers, you need to include the `weka.jar` file in the application's `lib/` directory. On Unix-based systems (or Windows systems that support it) it is sufficient to create a symbolic link in the `lib/` directory to wherever Weka resides on your system.

2.2 System Requirements

Model Monitor requires version 1.6 of the Java Runtime Environment (JRE). It is available from the Sun Microsystems java website at `http://java.sun.com`. Additionally, the distribution comes with three other Java libraries, which I will acknowledge here.

1. JFreeChart: We use JFreeChart for all the histograms and plots that we display. Source code and documentation is available at `http://www.jfree.org/jfreechart`.
2. JCommon: The JCommon library is required by JFreeChart and included in the basic JFreeChart distribution, so we include it here as well. Source code and documentation is available at `http://www.jfree.org/jcommon`.
3. SSJ: SSJ is a stochastic simulation library in Java, which we use for its implementations of Normal and Chi Square PDFs and inverse PDFs. Source code and documentation is available at `http://www.iro.umontreal.ca/~simardr/ssj/indexe.html`.

As mentioned before, our system supports the use of Weka classifiers. It is not required, but you may find its classifier implementations useful. Binaries, source code, and documentation are available at `http://www.cs.waikato.ac.nz/ml/weka/`.

3 Definitions

Before we start our introduction to the basic capabilities of Model Monitor and how to use them, we will define and explain some of the terms that we use throughout the following pages.

3.1 Basic Terminology

The following are basic definitions that should be familiar to most people in data mining / machine learning.

- **Feature** (or **Attribute**): A single piece of data, such as the age of a person or the cost of an item in a store, that is collected in a database. Thinking in relational database terms, an attribute is like a column of a relational database.

- **Numeric Attribute** (or **Continuous Attribute**): Any attribute, such as the age of a person, that can take on any numeric value. Common sense may limit the value in some cases (an age, for example, is probably a positive integer), but in theory the possible values of a numeric attribute are unbounded.

- **Nominal Attribute** (or **Categorical Attribute**): any attribute, such as the gender of a person, that has a pre-defined (usually small) set of valid values.

- **Instance** (or **Example**): A related set of feature values, such as the name, age, height, and weight of a person, or the name, cost, and UPC Code of a product in a store. Again, thinking in database terms, an instance is like a row in a relational database.

- **Dataset**: A related group of instances, meaning a group of instances that have all the same features. You can think of a dataset as a table in a relational database.

- **Classification Problem**: In general a classification problem is: Given a dataset of previously observed data and one attribute in particular that we want to be able to predict (the **class attribute**), build a model (i.e a tree or set of rules) that will accurately predict this attribute on future data. A simple example is a credit-scoring application: Given a set of attributes about prior loan customers (name, age, income, marital status, loan amount, ..., and whether they defaulted on the loan), build a model to predict whether future customers will default on their loans. By definition in a classification problem, the class attribute is *nominal*.

- **Two-Class Problem**: Any classification problem in which the class attribute is *binary*, meaning that it has only two possible values. Common examples are credit scoring (will a person default on a loan?) and disease prediction (will this person contract diabetes?).

- **Classifier**: A classifier is any algorithm for solving a classification problem. Most classification algorithms have two separate phases: 1) *Model Building*: Given a series of instances (*training set*), build a model (i.e. a tree or set of rules) that will predict a class for any unseen data instance. 2) Classification: Given a data instance in the same format as the training set, predict its class. Most classifiers provide, along with a prediction, an estimate of the probability that the given instance belongs to each class. These *probability estimates* are often used to measure a classifier's effectiveness.

- **Training Set**: The dataset used to build a classification model.
- **Test Set**: A dataset that is used to evaluate the effectiveness of a classifier. The test set should be different from the training set in order to accurately assess the generalization performance of the classifier: its ability to properly classify arbitrary instances of the problem.
- **Performance Measure**: Any numeric measurement of a classifier's performance on a dataset. The simplest is accuracy: the percentage of instances correctly classified. A **Loss Measure** is simply a performance measure that quantifies the failure, rather than the success, of the classifier. In other words, for loss measures, high numbers represent poor performance. A lengthier discussion of performance measures appears in section 3.3.
- **Hold-Out Validation**: A method of evaluating classifier performance in which some portion (usually at least 2/3) of the available data is used for training a classifier and the remaining portion is used for testing. The performance of the classifier is reported as its performance on the test set.
- **Cross-Validation**: A method of evaluating classifier performance in which the available data is divided into "folds". One-by-one, each fold is held out for testing, with the remaining folds used for training. The most popular variant is *10-fold cross-validation* where, at each step, 90% of the data is used for training and 10% for testing. The performance of the classifier is reported as the average of its performance over each of the folds. *5-by-2-fold cross-validation* runs two-fold cross-validation five times and averages all the results. The partition into folds is random at each iteration.
- **Bias** (or **Distribution Shift**): Any systematic change in the distribution of a feature, also known as *concept drift* or *concept shift*. A simple example is: if Americans truly are getting fatter and your dataset contains a person's weight, you will see the distribution of weights shift to the right over time. The term *bias* is usually reserved for sample selection biases, which are a very specific class of distribution shift. We (perhaps sloppily) use it to refer to any change in distribution. Much more information on biases is available in section 3.2.

The next couple sections outline and precisely define the biases and performance measures that we refer to throughout the later pages. The definitions are critical to using the tool, but can be skipped at first if you are just trying to familiarize yourself with the functionality.

3.2 Biases

here we define specifically and precisely the biases we have implemented in the current version of Model Monitor. Most of them are (necessarily) very specific implementations of a more general concept. For guidelines on changing the implementations to suit your needs, see section 6.

- **MAR (Missing at Random)**: MAR is a sample selection bias in which the probability that an instance is selected (alternatively, excluded) from the sample depends on the value of a particular attribute. After selecting MAR, select from the second dropdown the attribute along which to inject the bias. The parameter for MAR is the percentage of instances to remove. We implement MAR by sorting along the chosen attribute and removing the top $p\%$ of instances from that attribute.

- **MNAR (Missing not at Random)**: MNAR is a sample selection bias similar to MAR, except that the probability that an instance is selected depends on an unobserved feature of the data. Similarly to MAR, it removes a given percentage of the attributes along the chosen feature, but in this case the remaining instances have that feature marked as missing. This, in effect, simulates bias based on an unknown feature.

- **Mean Shift**: Mean shift will increase or decrease the value of a feature by a given number of standard deviations. The end effect is a shift of a feature's distribution in one direction. Unlike the sample selection biases, mean shift does not change the size of the dataset.

- **Random Noise**: Random noise will add noise to the chosen feature for a random subset of the instances. More specifically, a percentage of the instances (given as a parameter) will be chosen at random. For each of these instances, the value the chosen feature will be increased or decreased by a random number of standard deviations. The end effect of this bias tends to be a "normalization" of the distribution, meaning that the distribution looks more gaussian-shaped.

- **Prior Probability Shift**: Prior probability shift models a situation where the posterior distribution of positive and negative class values, conditioned on the feature values, stays the same but the frequency of positive and negative examples in the data changes. We implement prior probability shift by randomly removing either positive- or negative-class examples until the desired prior probability of positive examples is reached.

3.3 Performance Measures

Here we define and, give formulas for, the different classifier performance measures available in Model Monitor.

- **Accuracy**: The most basic performance measure, simply the percentage of test instances that the classifier has classified correctly.

- **AUROC**: The area under the receiver operating characteristic curve, which plots true positive rate against false positive rate. AUROC ranges from 0 to 1, with 1 being a perfect classifier, 0 a classifier that is wrong every time. An AUROC of 0.5 indicates a classifier that is approximately random. AUROC is often preferred over Accuracy for highly imbalanced

datasets because it more effectively captures the tradeoff between true positives and true negatives.

- **Brier Score**: The Brier score is the average squared difference between the probability estimates of a classifier and the actual true class. Brier score is a loss measure, with a score of 0 representing a perfect classifier (100% confidence in the right answer all the time) and 1 representing an incorrect classifier (100% confidence in the wrong answer).

- **Calibration Loss**: What we call Calibration Loss is really the Calibration component of the Brier Score as described above[1]. Calibration Loss is defined as $L_C = \frac{1}{n}\sum_{j=1}^{k} n_j(r_j - p_j)^2$ where k is the number of unique positive-class probabilities (p_j) generated on the testing set, n_j is the number of examples that are predicted positive with probability p_j and r_j is the fraction of the p_j-predicted examples that actually belong to the positive class. The intuition is that if a classifier is *well-calibrated*, i.e. unbiased, then of a set of examples that is predicted class-1 with probability p_j, exactly p_j of them should actually be class-1.

- **Refinement Loss**: Refinement Loss represents the refinement portion of the Brier Score as defined in [1]. It is intended to capture the portion of the loss that is due to variance in the classifier's probability estimates. Using the same conventions as above, Refinement Loss is mathematically defined as $L_R = \frac{1}{n}\sum_{j=1}^{k} n_j r_j(1 - r_j)$. Note that $n_j r_j$ is the raw number of p_j-predicted examples that belong to the positive class.

- **NCE Loss**: Negative Cross Entropy (NCE) is an information-theoretic measure of distributional divergence. Intuitively, it measures the number of extra bits required to encode the difference between the classifier's predictions and the true class labels. NCE loss is defined as $L_{NCE} = -\frac{1}{n}(\sum_{i \subset class_1} log(p_i) + \sum_{i \subset class_0} log(q_i))$ where p_i is the predicted probability that instance i belongs to class 1 and q_i is the predicted probability that i belongs to class zero.

- **Precision**: Precision is a measure that originated in the information retrieval community to measure the effectiveness of a search engine at returning relevant information. In this domain, precision is the fraction of documents retrieved by a search engine that are also relevant. For classifier evaluation, precision is defined as $P = \frac{TP}{TP+FP}$ where TP is the number of true positives (correct positive-class predictions) and FP is the number of false positives.

- **Recall**: Recall is similar to precision except that it measures proportion of relevant documents that a search engine can find. For a classifier, recall is defined as $R = \frac{TP}{TP+FN}$ where FN is the number of false negatives: positive-class examples that were predicted as negative.

- **F-Measure**: F-Measure measures a classifier's effectiveness at both precision and recall. It is possible to define an F-Measure that gives arbitrary

weight (i.e. importance) to either precision or recall. The measure we implement is known as the F_1-Measure because it gives equal importance to both precision and recall. The formula for F_1-Measure is: $F = \frac{2PR}{P+R}$ where P is precision and R is recall.

3.4 Mathematical Calculations and Statistical Tests

Model Monitor uses a number of calculations and statistical tests to either detect the presence of distribution shifts in data or to evaluate the performance of classifiers under distribution shift. Fuller treatments of these topics can be found in [2, 3], but we describe the basics here:

3.4.1 Hellinger Distance

The *Hellinger distance* (or *Bhatacharyya distance*) is a measure of the divergence between two distributions. Given two distributions X and Y, each divided into p bins, the Hellinger distance $d(X, Y)$ between X and Y is:

$$\sqrt{\sum_{i=1}^{p} (\sqrt{\frac{X_i}{|X|}} - \sqrt{\frac{Y_i}{Y}})^2} \tag{1}$$

If X and Y have completely identical distributions, meaning that the fraction of data points in each bin is identical, the $d(X, Y)$ will be zero. If the two distributions are completely divergent, meaning that X only has points in bins that Y has no points, we have that $d(X, Y) = \sqrt{2}$.

Hellinger distance is used to compare the distributions of features between training and test sets and quantify the extent to which a feature's distribution has changed. In practice, if the feature is continuous, we bin it. If the feature is nominal, the possible feature values form the bins. Unfortunately, it does not answer the question "is this change in distribution significant?". In order to determine whether the difference is beyond what can be attributed to random variation, we turn to the Komogorov-Smirnov test.

3.4.2 Kolmogorov-Smirnov Test

The *Kolmogorov-Smirnov test* (or KS test) of distributional divergence compares two distributions, X and Y, and tests the null hypothesis "X and Y come from the same distribution" against the alternative hypothesis "X and Y come from different distributions." The main advantage of the KS test over other methods is that it is non-parametric, meaning that it makes no assumptions about the underlying distributions of the data. To calculate the KS test statistic, we need to determine the *cumulative fraction function* for the distributions X and Y. The cumulative fraction function for distribution X, $CFF_X(x)$ is simply the fraction of instances in distribution X that are less than or equal to x. The KS test statistic, D, is defined as the largest vertical deviation between $CFF_X(x)$ and $CFF_Y(x)$ if they are plotted on the same axes. In other words:

$$D = max_x\{|CFF_X(x) - CFF_Y(x)|\} \tag{2}$$

It turns out that if the sizes of the two samples are reasonably large, (a rule of thumb is $N_e = \frac{n_1 n_2}{n1+n2} >= 4$, where n_1 and n_2 are the sizes of the two samples), then the distribution of D is such that

$$P(D > D_{obs}) = Q\left(D\left[\sqrt{N_e} + 0.12 + \frac{0.11}{\sqrt{N_e}}\right]\right) \tag{3}$$

where the function $Q(x)$ is defined as follows:

$$Q(x) = 2\sum_{j=1}^{\infty}(-1)^{j-1}e^{-2j^2x^2} \tag{4}$$

(see [4]).

Using the KS test, we can isolate features whose distribution has significantly changed since a classifier was trained. However, we may not care if the distribution of the data has significantly changed, as long as our classifier is performing well. To estimate whether distribution shift (or some other factor) has influenced the performance of our classifier, we turn to the Kruskal-Wallis test.

3.4.3 Kruskal-Wallis Test

The *Kruskal-Wallis test* (or KW test) is a non-parametric test for analysis of variance. For data samples split into g distinct groups, the Kruskal-Wallis test tests the null hypothesis "all group medians are equal" against the alternative hypothesis "at least one group median is different." The Kruskal-Wallis test statistic is calculated by sorting list of examples (of all groups combined) and ranking them. The smallest example gets a rank of 1 and the largest a rank of n, the total number of examples. In the event of a tie, all tied examples are given the average rank (i.e. if examples 3-5 are tied, they all get ranked 4). The KW test statistic is defined by

$$K = \frac{12}{n * (n+1)}\sum_{i=1}^{g} n_i(\bar{r_i} - \bar{r})^2 \tag{5}$$

where n is the number of examples across all groups, n_i is the number of examples in group i and $\bar{r_i}$ is the average rank of all the examples in group i. $\bar{r} = \frac{n+1}{2}$ is the average of all ranks.

If the sizes of the groups are all sufficiently large (all the $n_i > 5$) and the number of ties is small, then the K statistic is approximately chi-square distributed with $g - 1$ degrees of freedom. Thus, the KW p-value can be approximated by $P(\chi^2_{g-1} >= K)$.

In order to detect potential changes in classifier performance, we apply the KW test to classifier probability estimates. The classifier probability estimates for the test set without bias form one group and the probability estimates for

the test set with bias form another. The KW p-value tells us if the bias has significantly changed the classifier's predictions.

The chi-square approximation given above requires that relatively few ties exist in the data. With classifier probability estimates, the number of ties is usually quite large. To correct for ties, we need to divide the original value of K by the quantity

$$1 - \frac{\sum_{i=1}^{G} t_i^3 - t_i}{n^3 - n} \tag{6}$$

where G is the number of groups of tied ranks and n is the total number of examples across all groups. Once this correction is made, the chi-square test can be applied normally.

3.4.4 Friedman Test

The *Friedman test* is a non-parametric statistical test useful for determining statistical significance under repeated measures. Assume, for example, in an Olympic gymnastics competition, the same n judges rate k different gymnasts. We could wish to know, does one judge give consistently higher (or lower) ratings than the others?

The Friedman test can answer this question. Given a series of n blocks (judges), each with k samples (gymnasts), the Friedman test tests the null hypothesis that "the block effects are all the same" against the alternative hypothesis that "the block effects are not all the same." Another way to state the alternative hypothesis is that it makes a difference which block a measurement came from.

The procedure for the Friedman test is similar to that of the Kruskal-Wallis in that it involves the computation of ranks. The difference is that in the Friedman test, we do not rank all the samples together, but rather we compute separate ranks within each block. Let r_{ij} be the rank of the j'th sample in the i'th block. Define $R_j = \frac{\sum_{i=1}^{n} r_{ij}}{n}$, the average rank of the j'th sample across all blocks. The statistic

$$Q = \frac{12n}{k(k+1)} \left(\sum_{j=1}^{k} R_j - \frac{k(k+1)^2}{4} \right) \tag{7}$$

is chi-square distributed with $k - 1$ degrees of freedom. We reject the null hypothesis at the confidence level α if $P(\chi_k^2 >= Q) < \alpha$.

To apply the Friedman test to compare classifiers, we run each of the k the classifiers on n different datasets. Setting up the calculation with the classifiers as blocks and the datasets as samples, we can determine if the performance of the classifiers is significantly different across datasets.

However, knowing that the classifiers are significantly different doesn't tell us the whole picture. As a researcher, we usually want to know: by how much are we winning? In other words, which other classifiers is my classifier significantly

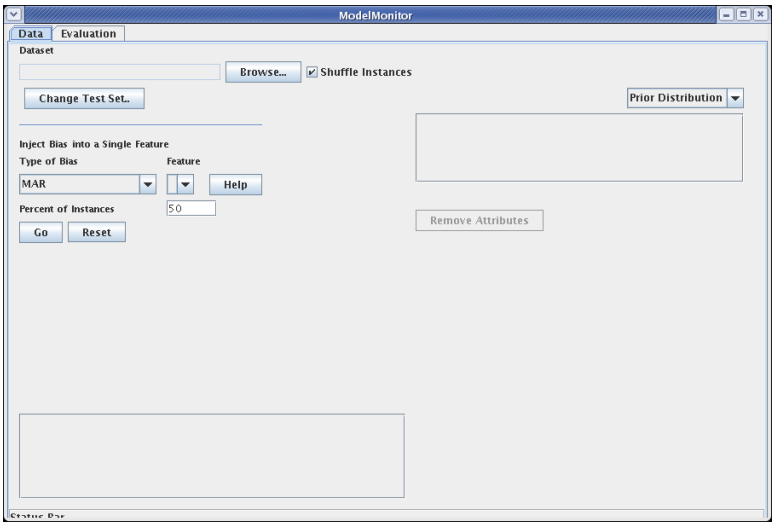

Figure 1: The user interface when it first loads.

better than? To answer this question, we need to perform the *Bonferroni-Dunn* post-hoc test. Given a series of classifiers that fails the Friedman test (i.e. is significantly different), the Bonferroni-Dunn test tells the *critical distance* between the average ranks of two classifiers that constitutes a statistically significant difference. Specifically, the Bonferroni-Dunn test computes the critical difference

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6n}} \tag{8}$$

Where q_α is the critical value for a two-tailed z-test at the significance level $\alpha/(k-1)$

We apply these tests to compare multiple classifiers under changes in distribution. The same bias is introduced into multiple datasets, and we compare the performance of a series of classifiers on those datasets. If any one classifier is significantly better than the others, it suggest that that particular classifier is more robust to the distribution shift in question.

4 Tutorial

Now that we have defined and explained the terminology we use throughout this manual, we will provide a detailed walkthrough of the basic features of Model Monitor and how to use them.

Model Monitor presents the user with a simple Java Swing GUI as the interface to all its features. The current interface consists of two tabs. The `Data` tab allows the user to select the dataset that will be used as the basis for any

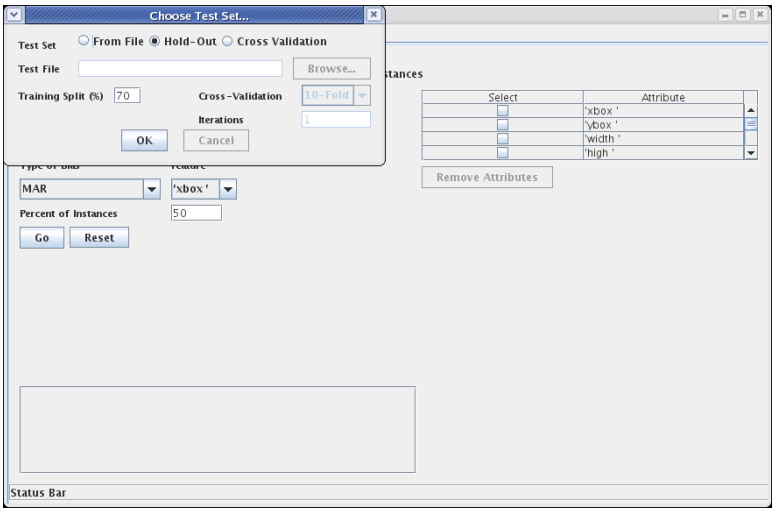

Figure 2: The dialog for selecting a validation method.

experiments. Once a dataset is chosen, the user can impart a series of distribution shifts in order to study the impact of distribution shifts on the dataset itself.

On the `Evaluate` tab, the user can specify any commandline classifier or WEKA classifier (if `weka.jar` is in the application classpath) and evaluate that classifier under a variety of different distribution shifts or biases.

4.1 Data Tab

The first step in the process is to select the dataset that will serve as the basis for the set of experiments. To do this, first click the "Browse..." button at the top of the screen (figure 1). This will bring up a file chooser that will let you browse for any C4.5 or ARFF dataset that you wish and that dataset will appear in the text box. You will notice that the "Shuffle Instances" checkbox is checked by default. All this means is that the order of the examples will be randomized after loading, which is only relevant if it is going to be broken into a train/test split or into folds for evaluation.

Once you have chosen a dataset, you can select the evaluation method (supplied test set, hold-out or cross validation) that will be used when you eventually run the classifier. Hold-out is selected by default. If you choose to use a separate test set, the original dataset will be used for training only. You have the opportunity to browse for this separate test set in the same manner as before.

As soon as the validation method is selected, the program will divide the dataset into a training and test set. For hold-out validation or a supplied test set, the division is obvious. For cross-validation, one fold will be chosen as the test set and the remaining folds as the training set. A histogram in the lower-

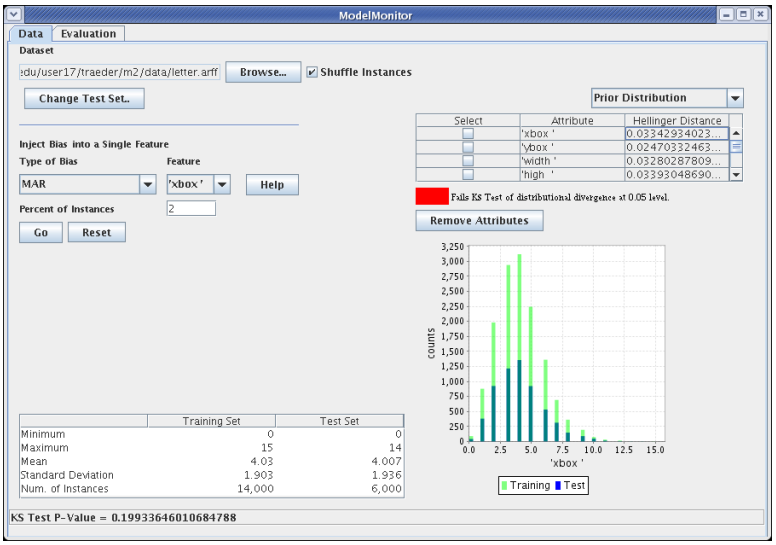

Figure 3: The user interface after loading a dataset. Information on the distribution of the first feature appears in the histogram and table.

right corner of the screen compares the distributions of the training and test sets. Light green bars represent the training set and blue bars represent the test set. The dark green areas are areas where the train and test sets overlap. Additionally, a table in the lower-left shows basic statistics for both training and test sets. Both these items are highlighted in figure 3.

Below the file-selection area, the user has the opportunity to try out various biases and distribution shifts on the data. Simply select a bias from the first dropdown, a feature along which to inject the bias from the second dropdown, enter the severity of bias in the text box and click `Go` (in figure 4, we have chosen to introduce the MAR sample selection bias, along the feature "high", over 20% of the instances). Details on the different available biases and their specific implementations are avaialble in section 3.2.

Once you click the `Go` button, two things happen: first, the test set (or one fold of a cross-validation) is altered according to the specified bias. Second, the table at the upper-right hand side of the screen populates with information. It shows, for each attribute other than the class attribute, the Hellinger distance between the original training set and the altered test set. In general, a large Hellinger distance indicates a significant difference between the train and test distributions of a feature[2]. Introducing multiple biases in succession will work as expected.

Depending on the severity of the bias you have introduced, you may notice that some of the cells in the Hellinger Distance table highlight in red. Red highlights indicate that the given feature has failed the Kolmogorov-Smirnov test of distributional divergence at the 95% confidence level. Figure 5 shows the

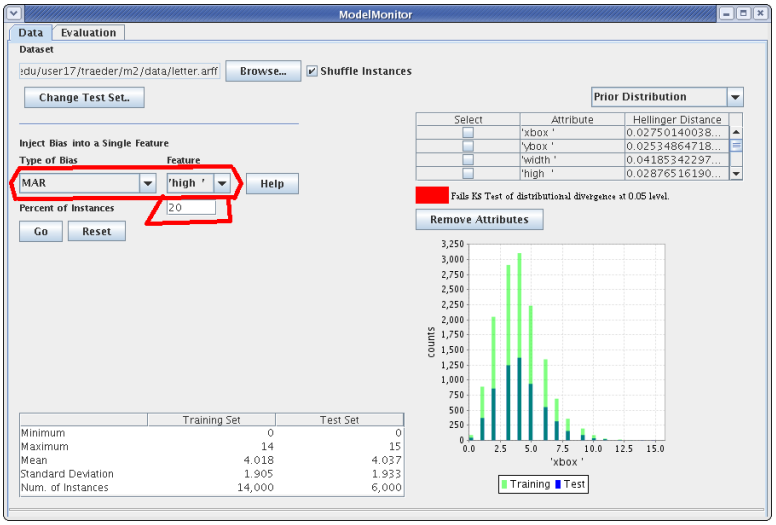

Figure 4: Introducing bias into a feature.

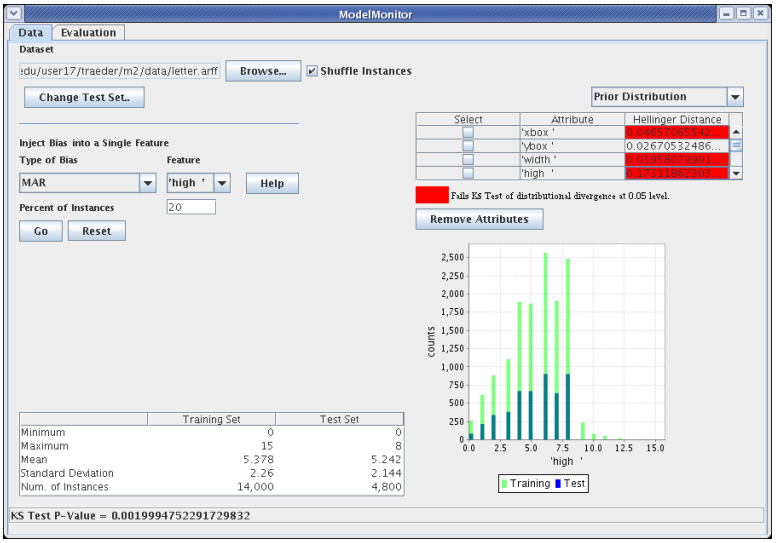

Figure 5: After bias has been introduced into a feature, the table and histogram change accordingly. Also, features that fail the KS test are highlighted red.

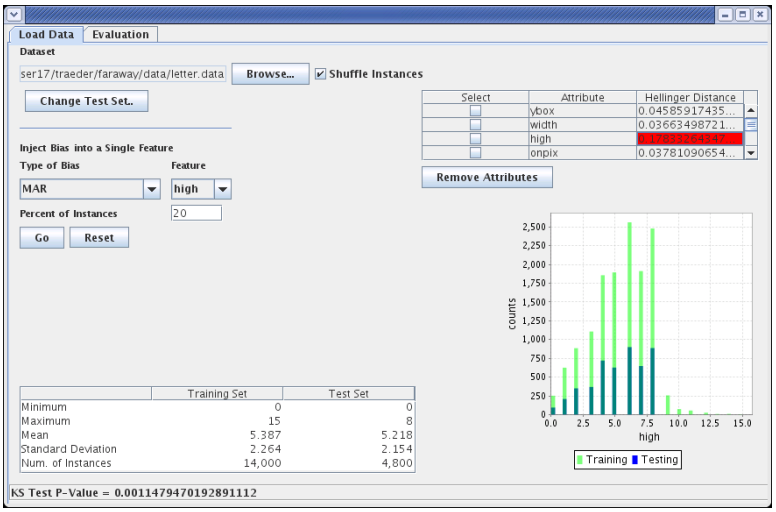

Figure 6: The test set distribution changes after bias.

impact of the MAR bias discussed above on the Letter dataset. In the table, the "high" attribute and two other attributes are highlighted red. The fact that "high" fails the KS test is expected, because we have drastically altered its distribtuion. The fact that two other attributes fail seems to indicate that this attribute (at least in the region we have removed) is somewhat correlated with the two other attributes.

The dropdown box above the table allows you to select which distribution to analyze. By default, Model Monitor displays information about the prior probability distribution of each feature, but we also provide the ability to visualize the class-conditional posterior distribution of the features. Changing the selection in this dropdown will cause the table and histogram below to update accordingly. KS p-values (and red colorings) are also updated.

Histogram Analysis The fact that a feature has failed the Kolmogorov-Smirnov test indicates that its distribution has significantly changed between training and test sets, but how has it changed? The histogram in the lower-right of the screen automatically updates to reflect the new test distribution. By clicking on different features in the table, the histogram and the table of statistics will show the distribution of that feature. This way, you can visualize how the train and test distributions differ. Clicking on the individual Hellinger distances will show the KS-test p-value for that feature in the status bar at the bottom of the screen. Figure 6 shows the distribution of the "high" attribute. We see that the top portion of the distribution is now missing.

Sometimes, you can see a very profound effect just in the histogram. In particular, with sample selection biases, one can often observe correlation between the features, meaning that there is a "gap" in the distribution of features other

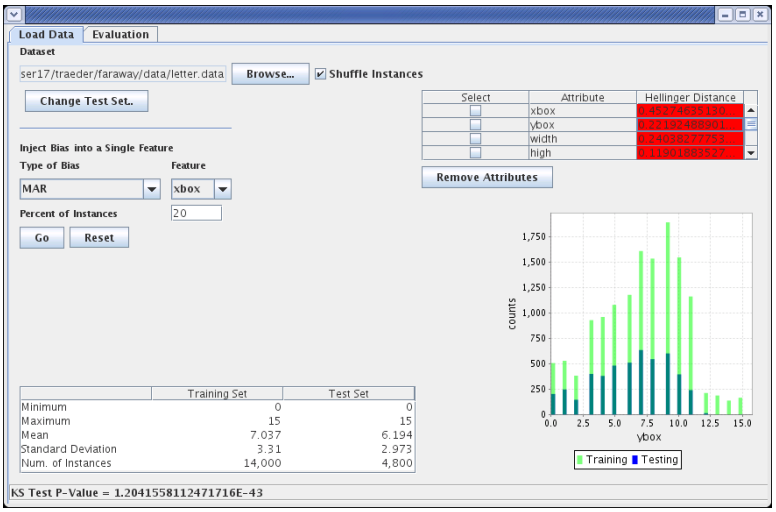

Figure 7: The dataset after 20% MAR on the attribute *xbox*. Many other features, apparently correlated with this one, fail the KS test.

than the ones along which the bias was originally injected.

We show an example in figure 7. Whereas 20% MAR on "high" has very little effect on other attributes, a 20% bias on "xbox" causes most of the other attributes to fail spectacularly. In particular, we see that the test set is missing its higher values of "ybox" and that the ybox feature fails the KS test at any reasonable level, with an astonishing p-value of 10^{-43}.

4.2 Evaluation

The `Evaluation` tab allows you to systematically evaluate the effect of a given distribution change on a particular classifier. The basic evaluation GUI presents the user with three options:

1. Evaluate classifier on dataset: Will build and evaluate the selected classifier on the dataset selected on the Data tab. The classifier will be built on the training set and evaluated on both biased and unbiased versions of the test set. If any attributes were removed on the Load Data tab, those attributes will not be considered for the classification model.

2. Evaluate classifier sensitivity: Will build and evaluate the selected classifier (on the dataset from the Load Data tab) under a well-defined range of distribution shifts. The user must specify the type of bias, the first bias value, the last bias value, and the difference between successive bias values. In the end, the classifier will be evaluated under each distribution shift within the range and the results are presented to the user as a graph.

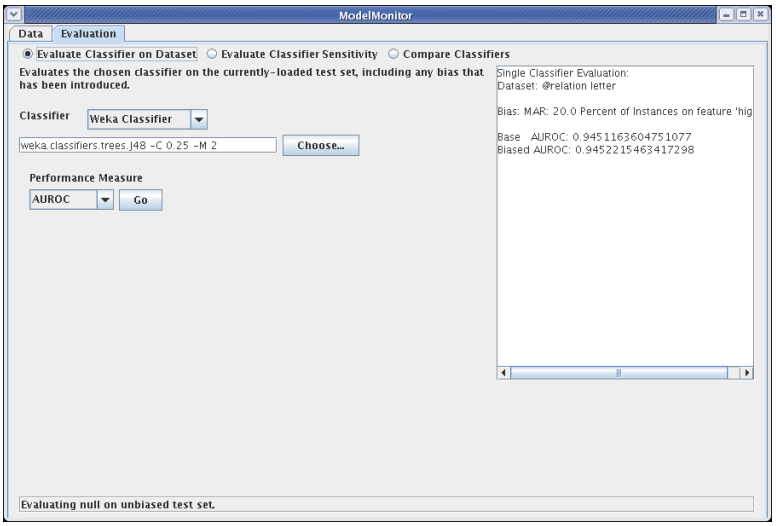

Figure 8: The basic GUI for running a single classifier, including results of a run.

To save chart space, only classifiers that fail the Kruskal-Wallis test (at the 0.05 significance level) are displayed.

3. Compare Classifiers: Will build and evaluate a series of classifiers on a series of datasets as specified by the user. Results are displayed in several forms, including tables and plots. See the `Configuration File` section for more information.

4.2.1 Running a Single Classifier

Continuing our example, we will see how the 20% MAR bias we introduced earlier impacts the performance of a classifier. We click over to the Evaulation tab. Figure 10 shows the single-evaluation GUI. We need to specify a classifier to use and a performance measure by which to evaluate it.

To choose a Weka classifier, select "Weka Classifier" from the dropdown menu and click the "Choose..." button. If the Weka Classifier option is not available in the dropdown menu, this means that the application was not able to find the `weka.jar` file on startup. Move `weka.jar` into the application's `lib/` directory and restart the application in order to fix the problem.

After clicking "Choose...", you will see the Weka classifier-choice popup menu. We pick J48, which is their Java implementation of C4.5. We choose AUROC as our evaluation metric and click "Go". The result of this quick little experiment appears in figure 10. The bias does not appreciably affect the performance of the classifier..

If, instead of a Weka classifier, you would rather run your own favorite classifier, you should select "External Classifier" from the dropdown and click

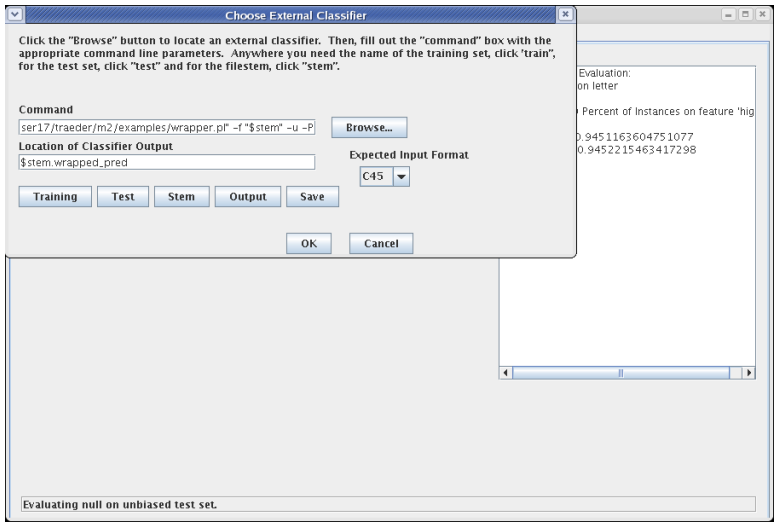

Figure 9: Dialog box for choosing a commandline classifier.

"Choose..." We provide a simple example here, but a more detailed instructions on integrating commandline classifiers is available in section 5.

Clicking the Choose button will bring up the classifier choice dialog presented in Figure 9. There are three controls in the dialog: the "Command" textbox will contain the full path to the classifier executable (or wrapper script), along with any necessary command line parameters. If the filename of the training or test sets (or the C4.5 filestem) is needed, you should use the Training, Test, and Stem buttons below. The Classifier Output File textbox will contain the location of a tab-delimited file containing probabilistic predictions for each instance. More details on the format of this file are available in Section 5. Finally, the dropdown box to the right selects the input format expected by the commandline classifier. Currently, the only formats supported are C45 names/data format and Weka ARFF format.

The process of entering a commandline classifier can be tedious, and so Model Monitor provides users the option of saving the information entered into this dialog. Clicking the "Save" button will prompt the user to name the current classifier. Enter a unique name and click "Ok". Now, an option with that name will be added to the dropdown box on the Evaluation tab. Any time you wish to use this classifier in the future, simply select it from the dropdown and click "Choose...". All the relevant fields will be populated. An example of this is illustrated in Figure **??**.

The perl script `examples/wrapper.pl` provides a perl wrapper for a C4.5 decision tree implementation. We put the commandline `perl examples/wrapper.pl -f $stem -u -P` into the Command. The $stem variable is replaced with the C4.5 filestem of the file where Model Monitor writes datasets after introducing bias. The `-u` option tells c4.5 to use the `.test` file for the specified filestem,

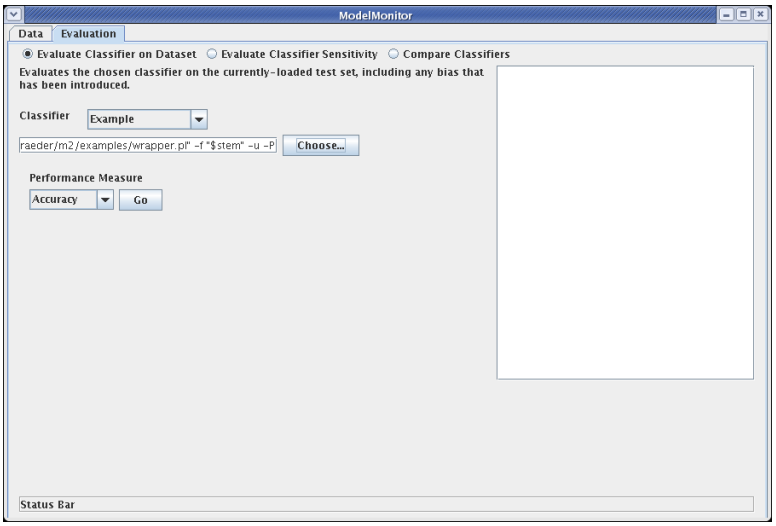

Figure 10: The commandline classifier is recalled by selecting its name from the dropdown and clicking "Choose."

and `-P` tells it to output probabilistic predictions.

In the Classifier Output File box, we need to provide the name of the file that will contain tab-separated probabilistic predictions on the test set (see below). The proper value will depend on your wrapper, but for `wrapper.pl` it is $stem.wrapped_pred, where $stem has the same meaning as before. The results are shown in figure 11. They are similar but not identical.

Remember now that two features appeared correlated with "high" in the region affected by the bias. We will investigate the impact of removing those (potentially redundant) features from the model in the hopes of improving performance. Navigating back to "Data", we remove the two offending attributes (figure 12).

Returning back to the Evaluation tab, we see that removing the attributes did indeed improve performance on this particular test set, improving both the AUROC under bias and the AUROC on the unbiased test set.

4.2.2 Sensitivity to Bias

The second use case, which we now explore, is the evaluation of a classifier's sensitivity to shifts in distribution. Suppose the user needs to find a classifier that is resilient to a particular bias (maybe they expect their customer's weight to increase over time, and they want to find a classifier that responds well in that scenario). For these kinds of inquiries, we provide a second interface, shown in figure 14.

The classifier choice options are the same as on the previous screen. However, in addition to those options, the user must now specify a bias and a range of

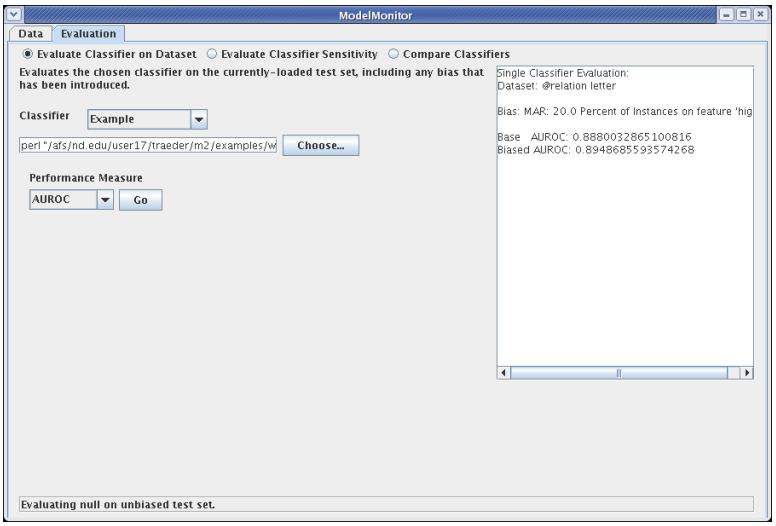

Figure 11: Simple example running a commandline classifier.

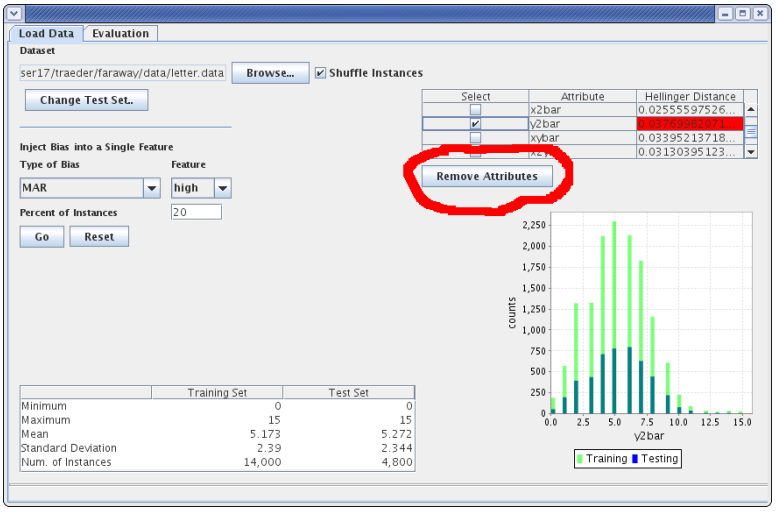

Figure 12: Removing attributes.

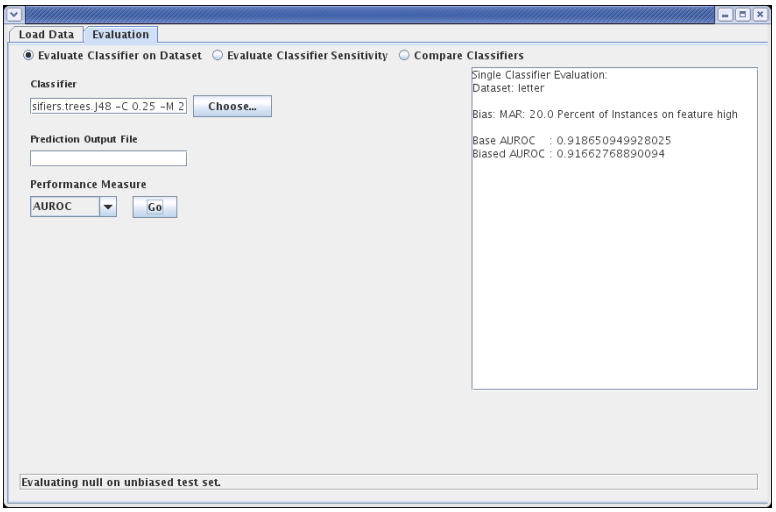

Figure 13: The results of a second run, after removing attributes.

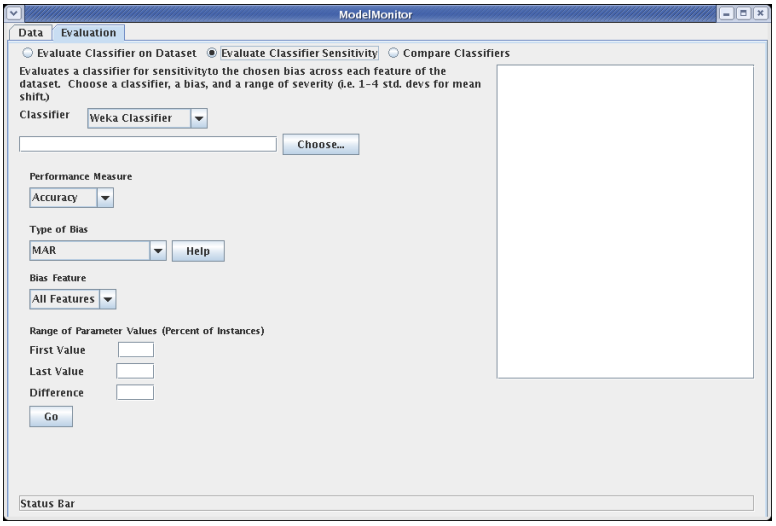

Figure 14: Basic interface for evaluating classifier sensitivity to bias.

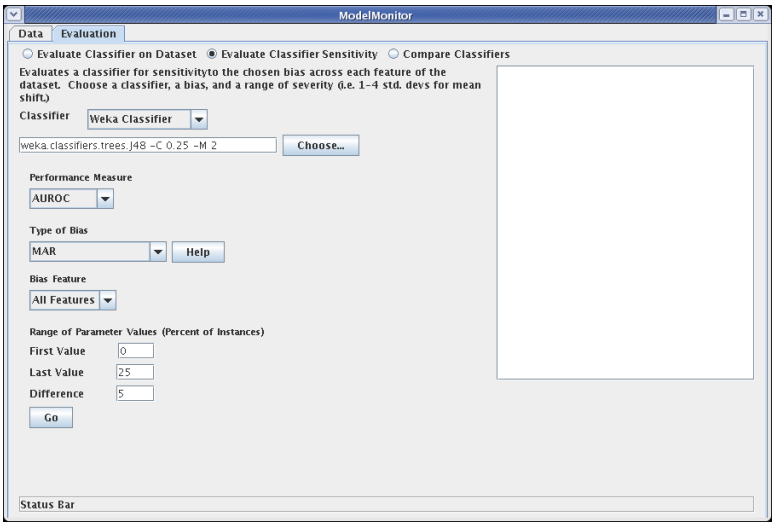

Figure 15: Sensitivity GUI with fields filled out.

severities for the bias. See section 3.2 for a description of all the different biases and what they mean. With the hope of illuminating this and making it clear, we'll dive right into an example. Look briefly at figure 15. We choose J48 as our classifier again, and consider MAR bias ranging from 0% of instances to 25% of instances in steps of 5%. What this means is that we introduce (0%, 5%, 10%, 15%, 20%, and 25%) MAR into each feature and present the impact of these changes to the user.

The ultimate output of this whole process appears shows the difference from baseline performance of the classifier (on the biased test set) as the increasingly severe bias (in this case, more instances removed by MAR) is introduced along each feature. To save chart space, only features that fail the Kruskal-Wallis test (at the $\alpha = 0.05$ confidence level) are displayed. The text area to the right displays progress information, including the performance of the classifier at each different level of bias. If any of the output lines has an asterisk, this means that the predictions of the classifier at this level of bias fail the Kruskal-Wallis test.

4.2.3 Comparing Classifiers

The final task on the Evaluation tab is the comparison of multiple classifiers on a collection of datasets. In this initial version, there is only limited GUI support for this feature. If you click the "compare classifiers" button on the evaluation tab, you will see a text box with a browse button. Clicking the button will allow you to browse for an XML configuration file that contains information about the classifiers, datasets, and biases to be used. This portion of the GUI is shown in figure 16. The format of the configuration files themselves (and exactly what the program does with them) is discussed in detail in section 5.1. Clicking

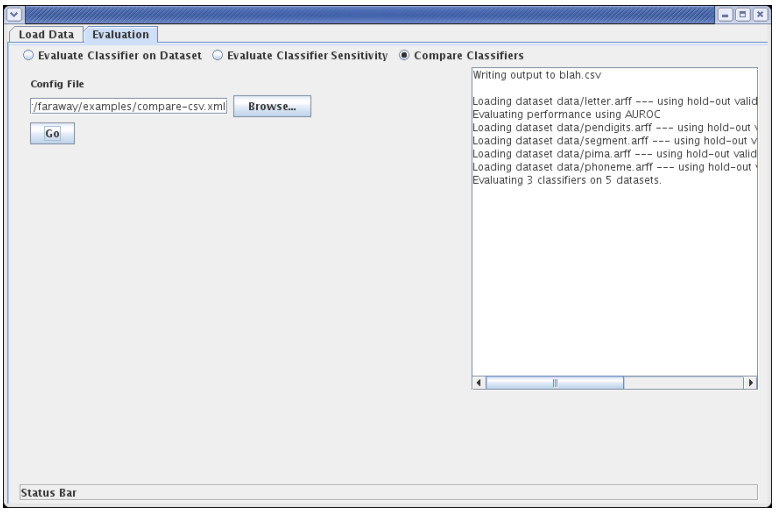

Figure 16: Panel for comparing classifiers.

"Go" will run the experiment set forth in the configuration file. The text area to the right of the screen displays progress information and shows the location of all the output files produced. The nature of the output is also discussed in section 5.1.

5 Commandline Classifiers

Model Monitor will interpret any classifier that does not begin with `weka.classifiers` as a commandline classifier, whether it is specified in the GUI or in a configuration file. The `Location of Classifier Output` box (or `predictions` attribute of the `Classifier` tag, in configuration files) is used only with commandline (non-Weka) classifiers. It specifies the location of a tab-delimited file that will contain, for each instance in the dataset, a list of probabilities associated with the various classes. For example, if a dataset has two instances, the first of which is predicted class 0 with probability 0.75 and the second predicted class 1 with probability 0.6, the output file should read:

```
0.75    0.25
0.4     0.6
```

In either the `Classifier` or `Location of Classifier Output` boxes, the following wildcards can be used to refer to the locations of various program-generated files.

- `$train`: The training set.
- `$test`: The test set.
- `$stem`: The file stem (i.e. filename without extension).

```
<ExperimentGroup outfile="out.html">
  <Experiment name="test1">
    <Dataset filename="letter.arff" trainingSplit="70"/>
    <Classifier name="C4.5" command="weka.classifiers.trees.J48"/>
    <Classifier name="Naive Bayes" command="weka.classifiers.bayes.NaiveBayes"/>
    <Classifier name="1R" command="weka.classifiers.rules.OneR"/>
    <Evaluator name="m2.evaluators.AUROCEvaluator"/>
    <Bias type="m2.bias.MARBias" start="0" stop="50" step="10"/>
    <Dataset filename="pendigits.arff" trainingSplit="70"/>
    <Dataset filename="segment.arff" trainingSplit="70"/>
    <Dataset filename="pima.arff" trainingSplit="70"/>
    <Dataset filename="phoneme.arff" trainingSplit="70"/>
  </Experiment>
</ExperimentGroup>
```

Figure 17: Sample configuration file.

- `$output`: The filename specified in the Prediction Output File box.

These options allow the tool to manage the partitioning of the data into training and testing portions as well as the injection if bias into the testing set. In most cases, specifying commandline without `$train` and `$test` or `$stem` will lead to woefully uninteresting output, as the classifier cannot be evaluated on the biased distribution.

Any filenames or pathnames with spaces in them must be quoted. This means using `&quot;` in configuration files and double quotes in the GUI. Model Monitor creates temporary output files (`$train`, `$test`) in the user's home directory. On Windows machines, these directories also often contain spaces. As a result, it is recommended that these wildcards always be quoted. The buttons provided in the GUI do this automatically.

5.1 Configuration Files

As an alternative to choosing classifiers, datasets, and biases in the GUI, the user may specify an XML configuration file which outlines a list of experiments that the user wishes to run. A sample configuration file appears in figure 17. This, along with other examples, appears in the `examples` directory of the distribution tarball. The meanings of each of the individual elements are described below.

Root Element Every XML document has a root element and the root element for configuration files is `ExperimentGroup`. It has one required attribute, `outfile`, which contains a valid filename where basic experimental results will be written. We currently support output in two formats, HTML and CSV. If the filename ends in ".htm" or ".html", HTML output will be produced, otherwise the output will be CSV. Any other output that the experiments generate

(described in greater detail below) will be placed in the same directory as the `outfile`. If the parent directory of the `outfile` does not exist, it will be created (assuming the user has permissions to do so).

The `ExperimentGroup` can contain only `Experiment` child elements.

Experiment element The `Experiment` element groups together all classifiers, datasets, and biases that are going to be considered as part of an experiment. Each classifier will be run on each dataset with each specified bias and several forms of resulting output (described below) will be written. Each experiment can have a name, which is used only to identify it in output. Additionally, each experiment can contain the following:

- **Classifier**: Specifying a name (for output) and a command (either commandline arguments or full WEKA classname and arguments). Each experiment needs at least one Classifier.
- **Evaluator**: Specifies an Evaluator class that will evaluate the performance of each classifier in the different scenarios. Each experiment needs exactly one evaluator.
- **Bias**: Specifies a bias or distribution shift to introduce into the data. The "type" attribute specifies the class name of the bias. "start", "stop", and "step" specify the severity of the bias. Specifically, the specify the beginning, end, and difference between successive elements in a series of increasingly severe biases that will be introduced into the test set. Each experiment can contain any number of biases.
- **Dataset**: Specifies a dataset. The appropriate attributes depend on what type of validation you want to do. For hold-out validation, the filename attribute specifies the dataset to use and the "trainingSplit" attribute specifies what percentage of the attributes the classifier should be trained on (with the remainder for testing). For cross-validation, you need to specify a "filename" as well as a number of "iterations" and "folds". For using a separate test file, you need only "train" to specify the name of the training file and "test" to specify the file used for testing.

.

Output Experiments produce three separate types of output. The first is a raw dump of the performance of each classifier on each dataset under each bias. This information goes in the file specified as output for the ExperimentGroup. The second form of output is a plot of classifier performance as increasingly severe bias is observed along each feature. An example of this type of plot appears in figure 18. To save chart space, features are only displayed if the classifier predictions are significantly different from their predictions on the unbiased test set (according to the Kruskal-Wallis test).

Figure 18: The performance of C4.5 on pendigits under a MAR bias of varying severity.

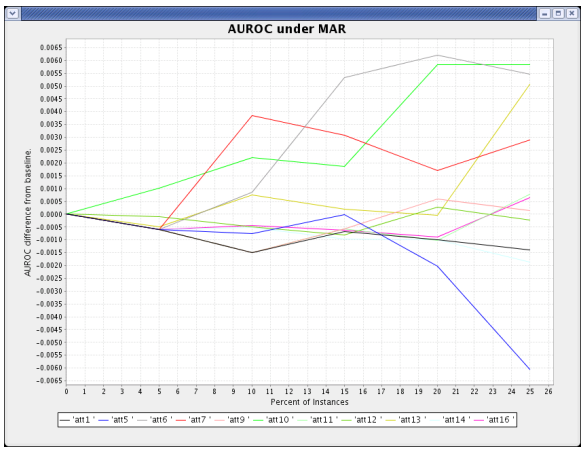

Figure 19: Example output comparing the performance of several classifiers across datasets under MAR bias.

Freature: Average of all.

Bias: MAR

	0.0	10.0	20.0	30.0	40.0	50.0
1R	2.6	2.6	2.6	2.6	**3.0**	**3.0**
C4.5	1.6	1.4	1.4	1.4	1.2	1.2
Naive Bayes	1.8	2.0	2.0	2.0	1.8	1.8
Friedman test p-value	0.2465	0.1652	0.1652	0.1652	0.0149	0.0149

Bonferroni-Dunn CD: 1.4175

The final form of output shows classifier performance across datasets. The performance of the classifiers under bias is compared (on several datasets) using the Friedman test. First, a certain type and severity of bias is introduced across each feature, and the performance If the Friedman test shows a statistically significant difference, we conduct the Bonferroni-Dunn post-hoc test to determine whether any of the classifiers are significantly better than the **first** Classifier specified in the configuration file. We present the *rank* for the performance of each classifier on each dataset. If any classifier is significantly different (according to the Bonferroni-Dunn test) than the base classifier, its rank in the table is bolded. An example of such a table is presented in figure 19.

The different classifiers are in rows and the severity of the MAR bias in columns. The number in each cell is the average rank of the classifier under the given bias. The p-value in the last row indicates that there is a significant

difference between the classifiers under 40% and 50% MAR. The fact that 1R is bolded in the final two columns indicates that it is significantly worse than C4.5, the reference classifier. The Bonferroni-Dunn critical distance, given at the bottom of the screen, is how far away the average ranks of two classifiers need to be in order for there to be a significant difference between them.

6 Extensibility

The configuration file mechanism means that our framework is naturally and easily extensible. While we plan to add more performance measures and biases/distribution shifts in future releases, individual users can easily add these things themselves. To add a new bias, you need to create a class that extends the `Bias` abstract class and defines the `injectBias` method. If you ensure that your new class is in the classpath, you can use it in a configuration file just like any of the built-in biases. One potential use for this is if, for example, you don't like our particular implementation of MAR (i.e. maybe you need the bottom instances removed rather than the top instances). Similarly, users can implement new performance measures by extending the `Evaluator` class and implementing the `evaluate` method.

7 Caveats and Limitations

Because Model Monitor is still under active development, there are a few performance issues about which users need to be aware.

1. The use of WEKA classifiers is currently memory-inefficient. In order to completely disassociate the tool from WEKA (it seemed unreasonable to require weka.jar for people who do not use WEKA), we use a different internal dataset representation than WEKA does. This means that when instantiating a WEKA classifier there are in fact two separate representations of the dataset in memory.

2. Even though WEKA does not require that the class attribute be the last attribute in a dataset, the tool currently assumes that it is.

3. The loading of large datasets is currently very slow. We are working to address this issue.

8 Comments

Please direct all comments, questions, and (especially) bugs to `traeder@nd.edu`.

References

[1] G. Blattenberger and F. Lad. Separating the Brier score into calibration and refinement components: a graphical exposition. *AM. STAT.*, 39(1):26–32, 1985.

[2] D. Cieslak and N. Chawla. Detecting Fractures in Classifier Performance. *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 123–132, 2007.

[3] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[4] W. Press et al. *Numerical recipes in C*. Cambridge University Press Cambridge, 1992.