Analyzer: Integrated Tools to Guide the Application of Machine Learning

by

Christopher Clark

Supervised by Oren Etzioni

A senior thesis submitted in partial fulfillment of

the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2013

Presentation of work given on _____6/6/2013_____

Thesis and presentation approved by _____

Date _____

1 Introduction

As has often been repeated, we live in an information rich age where data analysis is becoming increasingly important. Machine learning is now used routinely by companies to create products such as targeted ads, personalized search results, fraud detection systems, and personalized user recommendations to name just a few. The prevalence of machine learning means more and more engineers are being called upon to use it in industry settings. However implementing such solutions often requires considerable expertise and experience. While it is true that many libraries exist that contain implementations of wide arrays of machine learning algorithms individual practitioners are still left with a host of decisions to make in regards to what data to collect, what features to use, and how to apply these algorithms, all of which will play a crucial role in how successful they will be. Practitioners also face a number of pitfalls when applying these algorithms which can easily delay or stall efforts to complete the task if not successfully avoided. My research explores the possibility of providing additional software tools to aid a practitioner's ability to make these implementation decisions in a less error prone and more effective way.

1.1 Machine Learning as an Iterative Process

Machine learning problems have often been generalized as an attempt to construct an accurate prediction hypothesis based on a number of example data points and labels. However, in practice those looking to apply machine learning will need to consider a large number of additional factors to be successful. In general we might expect engineers to solve problems using a cycle as shown in figure 1. First data is collected from the outside world through sensors, surveys, web site monitoring, or other means. Then this raw data is transformed into a form that can be inputted into generic machine learning algorithms, typically this involves reducing it to a set of feature vectors and label values. Next users must choose a classifier to use and set any parameters it might have before finally performing some kind of evaluation as to how successfully the classifier they selected was. The fact that this chart is a cycle reflects the fact that machine learning problems are often approached in an iterative fashion, as engineers use the results from previous iterations to determine what changes in their work flow to attempt next. Depending on the situation certain steps might be out of the control of engineers, but often times, and especially in industry settings, there will be room to revise almost any part of this process.

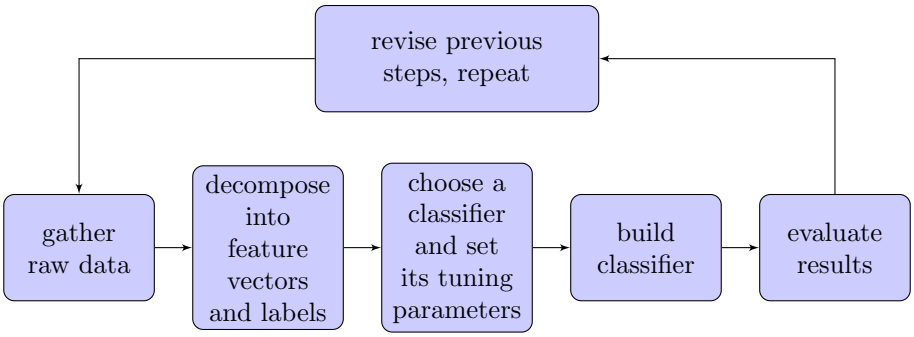

Figure 1: Iterative Procedure for Machine Learning

1.2 Machine Learning Implementation Challenges

Much research has been done on step 2 in attempts to improve what classification algorithms are available. This sort of research includes seeking new algorithms that can be applied in general settings as well as constructing algorithms specialized to perform well in particular cases, such as cases where there is a large amount of data. Work has also been done on feature engineering in certain domains; some easy examples are the well known techniques of word stemming or using tf-idf for the purposes of representing documents. However asides from these efforts it is often left to the user's individual expertise and intuitions to complete the remaining steps.

Yet these additional steps play a crucial role in the success or failure of machine learning projects. For a quick example consider the importance feature construction had in winning the Netflix challenge. Aside from using many sophisticated machine learning algorithms Koran[1] additionally attributes their success to their effort in feature creation. For example, the realization that users rate movies differently when rating them in quick succession rather than one at a time. Koren states that the addition of this feature, even in a relatively basic algorithm, yielded a baseline predictor with a prediction accuracy significantly better than that of the original Netflix Cinematch algorithm. In short using the right features can be more important than using a complicated, highly optimized classification algorithm. Additionally, outside of competition settings data scientists often have some control over how and what data is collected, which can have an even larger impact on their ability to build successful models. Making good choices in this regard can make or break a machine learning project, so tools that could improve practitioner's ability to complete these steps could have a dramatic impact on their ability to solve real world problems.

Asides from these challenges users face an additional set of pitfalls and practical considerations when executing these steps that, while they might in general be avoided by experienced users, can seriously hinder their progress. To name a few, these might include failing to notice something trivial, like the fact a feature is completely uniform or nearly perfectly correlated with another feature making it redundant. It might also include failing to notice that an algorithm is overfitting and could be more accurate with increased regularization, or failing to check if additional data would help by building a learning curve before spending time and energy collecting more data. Finally less experienced users may be unfamiliar with what classification algorithms they should try and how to fine-tune their parameters.

1.3 Solution Design

To explore our ability to help engineers with these issues I have constructed an extension to the well-known data mining framework Weka. The extension aims to help users solve machine learning problems more effectively by aiding them in making some of the critical choices they need to make when implementing their solutions. The extension, called Analyzer to reflect its very general nature, provides an additional tab to Wekas explorer GUI that gives users access to a number of additional data analysis tools. This set of tools includes one called MisclassificationMiner which is designed to help users with the task of feature construction and data collection. In addition it includes a number of tools aimed at addressing some of the difficulties discussed in the previous section.

2 Related Work

Some previous work has been done in this area. Patel et al. [2] conducted a study interviewing a number of software engineers in order to further examine what difficulties are encountered in practice when applying machine learning techniques. The study included interviews with 11 engineers experienced with machine learning and a study of how 10 less experienced participants went about tackling a machine learning problem in a controlled environment. They concluded that the application of machine learning techniques is hindered by three obstacles (1) difficulty pursuing statistical machine learning as an iterative and exploratory process, (2) difficulty understanding relationships between data and the behavior of statistical machine learning algorithms, and (3) difficulty evaluating the performance of statistical machine learning algorithms and

techniques in the context of applications. My work will attempt to provide tools to alleviate both (1) and (2) in a domain and classifier general way.

Many of the same researchers collaborated to construct a pair of systems, Prospectus [3] and Gestalt [4], that attempt to accomplish similar goals. Gestalt aims to address (1) by allowing users to easily implement and analyze a classification pipeline through a single framework, from decomposing raw data into features to building a model and evaluating it. Prospectus attempts to address (2) by allowing users to easily train a medley of different classification models and apply them to their data using leave-one-out cross validation. The user can then examines the subset of the data set where many of the classifiers struggled to make accurate predictions, the intuition being that data points in that subset are points where the features space is inadequate to classify those points and are thus points the user should focus on when making feature engineering decisions. Users can also examine points where many different classifiers came to similar, but wrong, classifications. Patel et al. showed such points are more likely to be points that are mislabeled. Both systems received positive feedback from test subjects using them.

Analyzer attempts to reach similar goals. Like Gestalt, Analyzer attempts to make the process of classification easier. Unlike Gestalt the focus will be on helping the user make effective choices rather than fully encapsulate a machine learning pipeline. In a similar manner to Prospectus Analyzer also includes tools to help users understand why and where their classifier is struggling. Unlike Prospectus misclassification miner will do this by finding clusters of examples that the user's classifier is struggling with rather than just finding all such examples.

The concept of finding clusters of hard to classify examples has been used before. Dekel et al [5] used the concept as a way to automatically determine where in the feature space the construction of piecewise classifiers would benefit their learning algorithm. My work, while also based on the notion of the importance of clusters of misclassified examples, differs in that it aims to use that information to benefit users rather than an automated process.

3 Analyzer Overview

The functionality in Analyzer can be broken down into two parts. First, some more basic tools are added that have well known implementations and are built to help less experienced users better understand their data, choose which classifier to use, and avoid various mistakes, as well as provide shortcuts for more experienced users. Second, some more advanced functionality is included that aims to aid users when attempting to tackle the difficult, but important, task of feature engineering.

First I will cover the more basic tools. These tools typically gather some statistical information about the user's data and then provide some basic interpretation of those statistics to aid users who might be less familiar with the implications of the information returned. Some additional benefit to making these tools is simply that they will now be highly visible to users, so users who are unaware of these techniques might learn of them. Afterwards I will cover MisclassifcationMiner in more detail. All tools include documentation on how they should be used and what their purpose is.

3.1 DataCheck

DataCheck calculates a number of simple statistics on the user's data and attempts to bring any important facts the user should be aware of to their attention. This includes 1) Checking for features that have the same value for all or almost all examples and alerting the user to their presence. 2) Checking for features that are perfectly or almost perfectly correlated and alerting the user. 3) Checking for examples that are duplicated, or examples that are overtly noisy or are identical except for class value and alerting the user. 4) Checking for examples or attributes that are almost entirely composed of unknown values. Alerting users to the presence of duplicated examples, concentrations of unknown values, and closely correlated features can alert users to weaknesses or problems in the data gathering and formatting phases. This tool also outputs some statistics, such as the correlation matrix, the top most correlated features, and number of unknown values. Basic interpretations of these results are included, such as informing a user that a feature is redundant if it is perfectly correlated with another feature and should be removed to speed up computation of any remaining steps.

3.2 OverfitCheck

OverfitCheck answers the question Is my classifier overfitting?" It simply trains a given classifier and compares its accuracy on the test and train data. This is easily done using Wekas API, but this tool can complete this action in a single step and will give the user a direct 'yes' or 'no' answer to the question. It also allows a user to run this operation over multiple trials to get a more robust result.

3.3 GenerateLearningCurve

GenerateLearningCurve builds a learning curve for a specified classifier. Again doing this is possible using Wekas interface if one is willing to step through enough menus, but there is benefit to making it more readily available. Also, to the best of my knowledge, Weka's base model does not include functionality to visualize the results which is included here (an example is in the appendix). GenerateLearningCurve will also try to directly tell the user whether gathering additional data would improve their classifier and how much improvement they should expect.

3.4 EntropyVsAccuracy

EntropyVsAccuracy builds a graph of the entropy vs. accuracy of a user's examples as was done by Prospectus. Users can use this to seek out potentially mislabeled data points or hard to classify as outlined by Petal et al. [3].

3.5 ClassifierSearch

ClassifierSearch, answers the question what classifier should I use, it runs a multitude of classifiers with a variety of settings on the user's training data, timing out ones that take an excessively long time to train. The user can input the classifiers they want to try as a list, but including a default list or possibly a number of recommended lists to be used in various situations (this might include a list of classifiers to try for regression, for classification, for large datasets, for smaller datasets etc.) can provide a way for less knowledgeable users

to search for the most effective classifier.[1]

3.6 MisclassificationMiner

Analyzer also introduces a new tool, MisclassificationMiner, that attempts to help users with feature construction. Doing so in a domain general way is quite challenging as often this would often involve the application of domain knowledge and individual ingenuity. It is true that some automated feature construction system exists, however they tend to work through brute force, searching through many different possible feature combinations to find ones that might help. Due to the exponential number of ways features can be recombined such a system will not be able to search even a significant fraction of the possibilities expect in extremely limited cases. More importantly, a fully automated domain general system cannot, by definition, leverage any sort of insights into the domain that the user might have, which is potentially a highly valuable resource. It is, for example, very unlikely that such an automated system could have suggested to the participants in the Netflix challenge that users rate movies differently if they are rating many movies in bulk. Additionally an automated system could not determine what new data to collect simply because, until the data is collected, the system has no way to evaluate the usefulness of that data. This is in contrast to a human who might have some intuitions as to which pieces of data will be useful based on their background knowledge of the task.

Given these difficulties, rather than attempting to automate some of these steps, MisclassificationMiner instead attempts to provide insights into the data that might help humans complete them more effectively. What sort insights might be useful? Prospectus operated under the hope that allowing users to focus on examples that have proven hard to classify would be helpful in this regard. This has some clear intuitive appeal, there is no need for a user to try and improve performance on examples the system is already getting right so this will allow user's to focus their attention on the places improvement is possible.

This approach does have some disadvantages. If the dataset is large the user might be unable to manually review even a small fraction of the difficult to classify examples, and even if a large number of points are manually reviewed some important information about them might be missed by the reviewer. MisclassificationMiner takes a different tack by trying to identify subsets of the feature space were the user's classifier is performing especially poorly. Ideally a user, once aware there is a problem with a specific region, will be

[1] Not fully implemented in current model

able to better direct their own knowledge and skill to discovering why that is and relieving the problem. There is also some benefit to knowing about these sort of subspace even if the user is unable to make any improvements. At the least the user will have a deeper understanding of how their classifier is behaving and the nature of their dataset.

Since we ultimately aim to be informative to users we want to be able to express these subsets in a human readable way. Clusters of the sort that are generate by kMeans or Gaussian Mixture Models, which are described using mathematical formulas that take every feature as input, are thus not suitable as they quickly become too complex for people to acquire an intuitive grasp of what they are describing. MisclassificationMiner instead expresses subsets using a conjunction of binary rules. For example, in a feature space with numeric features f1, f2, f3 we might produces a rules such as:

$$(f1 > 10 \ AND \ f2 < 0 \ AND \ f1 < 20)$$

Specifying a subset including all examples with a f1 value between 10 and 20 and a f2 value larger than zero. I hypothesize that being alerted to the presence of these subsets in this manner will, in some cases, be helpful to a user considering the problem of feature engineering.

4 Implementation

4.1 Interface

Analyzer, as mentioned previously, is an additional tab in Wekas explorer interface. It is designed to imitate the layout of the Explorer's other tabs so as to be more familiar to users who are already comfortable with Weka, shown in figure 2. Like those other tabs the Analyzer tab includes an area to select what tool the user wishes to use and a menu to customize the settings of that tool at the top. Also, like other Explorer tabs, Analyzer includes a combo-box to allow the user to select what featured should be used as the class of their data. The bottom panel will display status updates to indicate what calculation the system is currently working on. The large area in the middle will display textual information the tools output. The area to the middle-left will show a list of tools that have finished running. Clicking on the results in this pane will allow users to open any additional graphics or visualizations the tool produced.

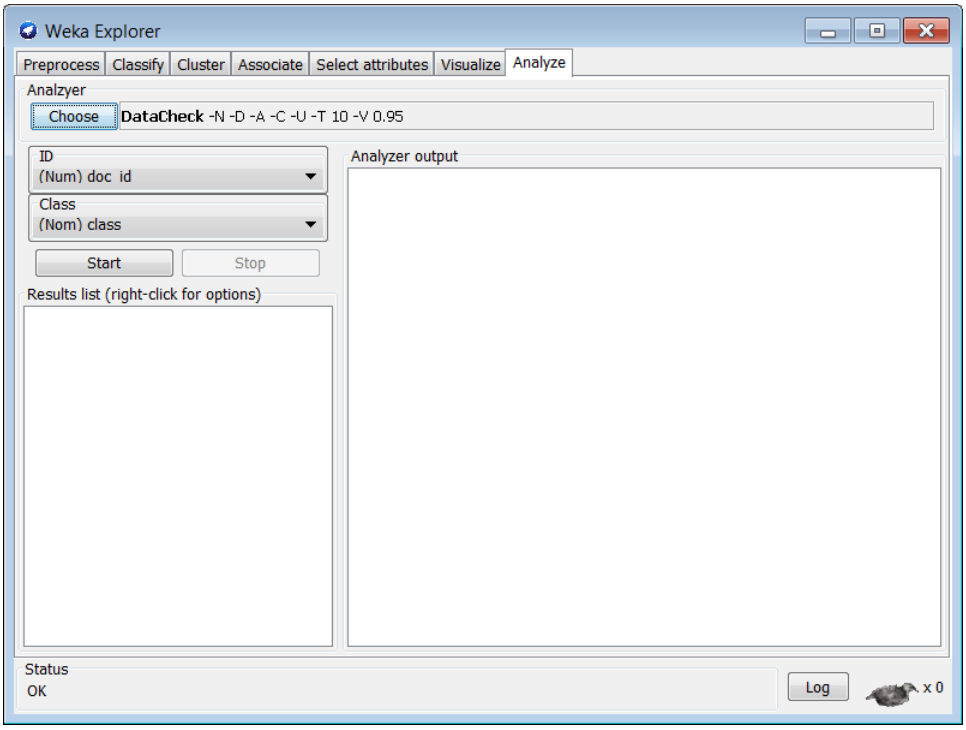

Figure 2: The Analyzer Tab

One significant change compared to Weka's ordinary interface is the additional "ID" combo-box which allows the users to select an attribute to be used as an ID for their examples. This attribute can be numeric or a string as long as each example's ID value is unique. This allows users to easily keep track of individual examples of their data. For example, if an ID attribute is specified, the "DataCheck" tool can list the IDs of examples that are duplicates of each other allowing a user to cross reference those IDs to find any additional information they have about those examples.

The implementation of the more basic tools is well known and does not need to be reviewed. Screenshots of the interface can be found in the appendix. The implementation of MisclassificationMiner is reviewed next.

4.2 MisclassificationMiner

MisclassificationMiner operates in three steps. First the data is split into two groups, the 'misclassified' and the 'correctly classified' group. Next a list of possible of 'rules' that can be used to split that data into subsets are compiled. Finally we will search through the possible conjunction of those rules to find ones that

isolate spaces containing many examples in the misclassified set but few examples in the correctly classified set. MisclassificationMiner uses an algorithm derived from Segal's [5] GoldDigger algorithm.

4.2.1 Dividing the Data

Assume we have a set of n data-points $\{x_1, x_2, ...x_n\}$ with labels $\{y_1, y_2, ...y_n\}$, we will first divide them into M and C, or the misclassified and correctly classified set. This will be done by dividing the data into some number of folds, building a classifier on all but one of the folds, and using the trained model to label the remaining fold, and repeating until all data points have been labeled. We then repeat the process i number of times resulting in i number of labels for each example. Finally we specify some cutoff, c, and specify that all points that were labeled incorrectly more than c times are put into the misclassified group and the rest put into the correctly classified group. If the labels are numeric we say that a classifier mislabeled a data-point if its estimate was greater the some cutoff of the data-point's true label. Building the classifier multiple times allows us to focus on examples the classifier is getting wrong due to bias rather than of variance. The intuition being that such examples are more likely to have a systematic reason for being hard to classify, so a user is more likely to be able to ascertain a useful insight from examining them.

One of the main advantages of using binary groups is that it will allow us to apply data mining algorithms to the result. It is, however, somewhat of a simplification as further computation will treat all examples in each group in an identical manner ignoring that fact that examples within each group might have considerable variance in classifier accuracy. It is left to future work to determine whether better results can be found with an algorithm that does not require making this kind of split.

4.2.2 Defining the Rule

Next we need to create a set of "rules", R, that partitions the feature space. I define a 'rule' for a particular domain to be a function that returns either true or false for any data point inside that domain. Therefore a rule, r, is function, $r(x_i) = \{True|False\}$. I will say r 'covers' an example, x_i, if $r(x_i) = True$. Rules are created on a feature by feature basis. For a discrete feature, F, with possible values $\{w_1, w_2...w_n\}$ we create rules $\{F = w_1, F! = w_1, F = w_2, F! = w_2..F = w_n, F! = w_n\}$. Numeric values are more challenging. In general for a numeric attribute k the strategy here is to find a series of 'breakpoints' $\{b_1, b_2...b_n\}$ and from

there derive rules $\{k >= b_1, k < b_1, k >= b_2, k < b_2..k >= b_n, k < b_n\}$. To create a list of breakpoints we can use an approach similar to what has been implemented in ID3 and other decision tree algorithms, that of scanning through a sorted range of values the that features takes on and adding all the cutooffs where both the feature's value change and the example's split by that cutoff's class value change to the list of break points. Since for large datasets this can produce an overwhelming number of rules proportional to the number of examples we offer an alternate approach where we separate the range of values that that feature takes on into a arbitrary number of quantiles and use those as our breakpoints. Since additional rules will cost us performance steps are taken to avoid creating redundant rules. For example, if a categorical feature only has two values we need only create two rules rather than four.

4.2.3 Ranking Rules

Having defined a list of basic rules to work with we now define a ranking criteria for rules. Laplacian Accuracy is used, which is calculated as

$$L(r, U) = \frac{n + k * u}{m + k}$$

where r is a rule, U is a set of examples, n is the number of misclassified examples in U that rule r covers, m is the total number of examples that rule covers in U , u is the total number of misclassified example divided by the total number of examples in U , and k is a constant. If k is zero we see that our formula reduced to n/k, meaning we will value rules that cover a high percentage of misclassified examples. Since a score of 1.0, the highest possible, is trivially achieved by a rule that covers a single misclassified example we add the additional factor of k. k acts much like a having a prior belief that r has an accuracy of u with confidence proportional k. If a rule covers no examples (so $m = n = 0$) then the Laplacian accuracy will become u, which is the accuracy we would expect a rule that simply returns true for all examples or a random subset of examples to achieve. Meanwhile as rules cover larger amounts of examples the k factor will get washed out, so for a rule with accuracy, accuracy referring to n/m, greater than u covering more examples will increase its score. By altering k we can adjust the trade off our algorithm makes between choosing rules the are highly accurate or cover a large number of instances.

4.2.4 Constructing Conjunctions

Based on our rules we can easily create new rules that are conjunctions of other rules. The conjunction of rules $\langle r_1, r_2, ..., r_n \rangle$ is simply a rule that returns true iff $\langle r_1, r_1, ..., r_n \rangle$ return true. Now we can define a search space of all possible conjunctions as drawn from the set R. Since this space is far too large to search exhaustively we instead use a greedy beam search with our initial state being the empty rule, or a rule that returns true of every example. We stop searching a path in the event adding any additional rules to the current conjunction would not increase the score of our rule.

The full pseudocode for Greedy Beam Search is as follows:

Algorithm 1 Greedy Beam Search

```
Let R = {r_1, r_1, ..., r_|R|} be a set of rules
Let B = ⟨b_1, b_2, ..., b_|B|⟩ be a list of conjunctions, we initialize each b_i to be the empty rule.
Let C be an empty, reverse Priority Queue
Let D = ⟨d_1, d_2, ..., d_|D|⟩ be list of booleans, initialized to True except d_0 = False
while an element in D is False do
    for i ∈ ⟨1, 2, ..., |B|⟩ such that D[i] is True do
        for r ∈ R do
            n = the conjunction of r and B[i]
            add n to C with score L(n)
            If size(C) > |B| pop an element from C
        end for
    end for
    Let B2 = copy(B)
    for i ∈ ⟨1, 2, ..., |C|⟩ do
        Let c = pop(C)
        if c∃B2 then
            D[i] = True
            B[i] = c
        else
            D[i] = False
    end if
  end for
end while
return The highest scoring rule in C
```

We want this algorithm to find multiple rules, however if we just run it multiple times it will simply return the same conjunction each time. To avoid this problem we remove all misclassified examples that a previous rule has covered from the dataset, forcing each iteration to attempt to isolate different misclassified examples. Eventually this search will return the empty rule (which will cover all remaining examples) or will have

covered all examples. In either case we return the rules found thus far and terminate. Note we will always terminate because a rule must score higher than the empty rule to be selected (since we start with the empty rule, and proceed greedily). The empty rule will, referring to the section on Rule Ranking, get a score of u. To get a superior score a rule must cover at least one misclassified example if there are any such examples left. Thus whenever we add a rule, if there were misclassified examples, we will reduce the number of misclassified examples in our dataset. Thus we will eventually cover all misclassified examples and terminate.

4.2.5 Overfitting

Rules tend to overfit, for instance adding a rule that covers all examples but a single correctly classified example will often be beneficial since doing so will reduce m, the number of examples covered, without reducing n, the number of misclassified examples covered. Such a rule likely reflects random chance rather than an actual pattern in the data. Even though these rules are not intended for classification adding superfluous rules can mislead users who might overestimate the significance of their presence. MisclassificationMiner allows users to see a breakdown of the usefulness of their individual clauses to help alleviate this problem but it is still beneficial to take some steps to reduce overfitting. I explored two potential regularization methods taken from Segal.

Regularizing rules involves adding a penalty term to the rule evaluation function which increases linearly with the number of clauses in the conjunction. Our rule scoring function becomes

$$L(r) = \frac{n + k * u}{m + k} - f * Len(r)$$

where $Len(r)$ is the number of clauses in r and f is some constant that determines how much to penalize long rules.

Pruning rules is an alternate tactic based on holding out part of the data as a validation set. Now once the rule has been constructed we can use the remainder of the data to iteratively remove clauses from the rule if doing so would increase the rule's score on the validation set. The pruning algorithm is as follows.

Algorithm 2 Prune Rule

Given a conjunction r composed of clauses $\langle c_1, c_2, ..., c_{|r|} \rangle$
Let $hasImproved = True$
while $hasImproved$ **do**
  $hasImproved = False$
  Let $bestRule = r$
  for $c \in r$ **do**
    if $Score(r$ without clause $c) > Score(bestRule)$ **then**
      $bestRule = r$ without clause c
      $hasImproved = True$
    end if
  end for
  $r = bestRule$
end while
return r

4.2.6 Bit Vector Optimization

This algorithm can be slow as we need to score many different conjunctions and scoring a conjunction involves iterating through every single example to check if it is covered by each clause of that conjunction and if it is in the misclassified or correctly classified set. A significant speed up to this process can be achieved using bit vectors, an approach also suggested by Segal. In this case we assume we have some fixed ordering of our examples $\langle x_1, x_2, ...x_n \rangle$, then for each rule we construct a bit vector where bit i is set iff the rule returns true for example x_i. We additionally create a bit vector with bit i set iff example x_i is in the misclassified set. Now to score a conjunction we AND the bit vectors of all the rules involved. The cardinality of the resulting vector (the number of set bits) is the number of examples the rules covers and the cardinality of that bit vectored ANDed with the misclassification bit vector is the number of misclassified examples. Additionally, since we build up rules clause by clause, we can use a bit vector to cache all examples that the conjunction currently covers saving us having to repeatedly complete the same AND operation.

4.2.7 Clustering, an Alternative Approach

I experimented briefly with using clustering to detect these groups of misclassified examples rather than rule searching. Although the results will be less human readable we might be willing to make this sacrifice if the resulting clusters are much better. Clustering algorithms have the advantage of being able to express clusters in ways other than using binary cutoffs on feature values. To try this out I ran a standard Gaussian

Mixture Model clustering algorithm over just the examples in the misclassified set, the results were used in the evaluation as a contrast to rule searching.

5 Evaluation

Since this system is built to be run with a human element it is decidedly difficult to test. Ideally human trials could be done to see if users found it beneficial to use the system. However given time constraints an approximation that attempts to show the rules MisclassificationMiner generates do isolate examples that are especially import for users to focus on. The metric is the same as used by [3] to evaluate prospectus.

5.1 Datasets

We use the well known Flowers and 20 Newsgroups datasets. The 20 Newsgroups data set, which consists of about 20,000 newsgroup documents divided almost evenly into 20 different UseNet discussion newsgroups. The classification task is to determine which newsgroup a given article was posted in. The documents were reduced to feature vectors using a bag of words with word stemming approach, the same features as used by Patel et al. to evaluate Prospectus. The flowers dataset consists of images of 17 types of flowers. Feature used were precomputed shape, color and texture features provided by Nilsback et al. [6].

5.2 Procedure

We create two version of each dataset, a before dataset and an after dataset. The original datasets are defined to be the after datasets. To create the before datasets we remove a number of features from the after datasets. For Newsgroups we remove 75

5.3 Metric

Now we define the accuracy gain for an example to be the accuracy rating of that example in the after dataset minus the accuracy rating in the before dataset. Accuracy gain measures the extent to which the

introduction of additional features was able to improve our ability to classify that example. We would like to be able direct users towards examples with high accuracy gain as those examples are ones that can be improved with additional feature engineering, and thus ones users might find useful to focus on when embarking on feature engineering.

To evaluate MisclassificationMiner I measure the average accuracy gain of the examples that are covered by the rules generated by MisclassificationMiner and compare this score with another potential method of detecting high accuracy gain examples, that of simply selecting the examples with very low accuracy ratings on the before dataset as was done by Prospectus. There is a trade off between selecting a large number of examples and selecting a set of examples where the average accuracy gain is very high. It is easier to select a high average accuracy gain set of points if only few points need to be selected, but having more points might be of more use to the user. To illustrate this the results are shown on a curve in figures 3 and 4.

5.4 Results

There are four lines on the graph, corresponding to four techniques used to select sets of points that might have high accuracy gain. On the graph the x-axis corresponds to the size of the sets selected and the y-axis corresponds to the average accuracy gain of the points in that set. I used each technique to select sets of examples of different sizes and calculate the average accuracy gain of the points in that set to create data points. The "unsure" line was built by adding all points with a 0 accuracy rating to a set to create the first data point, then additionally adding all points with a 1/15 accuracy rating to that subset to get the second data point, and so on. The Rule (Regularized) and Rule (Pruned) correspond to using the rule's generated through using pruning and through using regularization. The rules generated were scored based on their accuracy, or the number of misclassified examples covered in the training data divided by the number of examples covered in the training data. Then all the examples covered by the highest scoring rule were used to generate the first data point, all the examples covered by either the first or second highest scoring rule were used to generate the second data point and so on. Finally the clustering line corresponds to using the subsets generated by the clustering algorithm, here we add examples to the subset in an order depending on how distant they are from the nearest cluster generated.

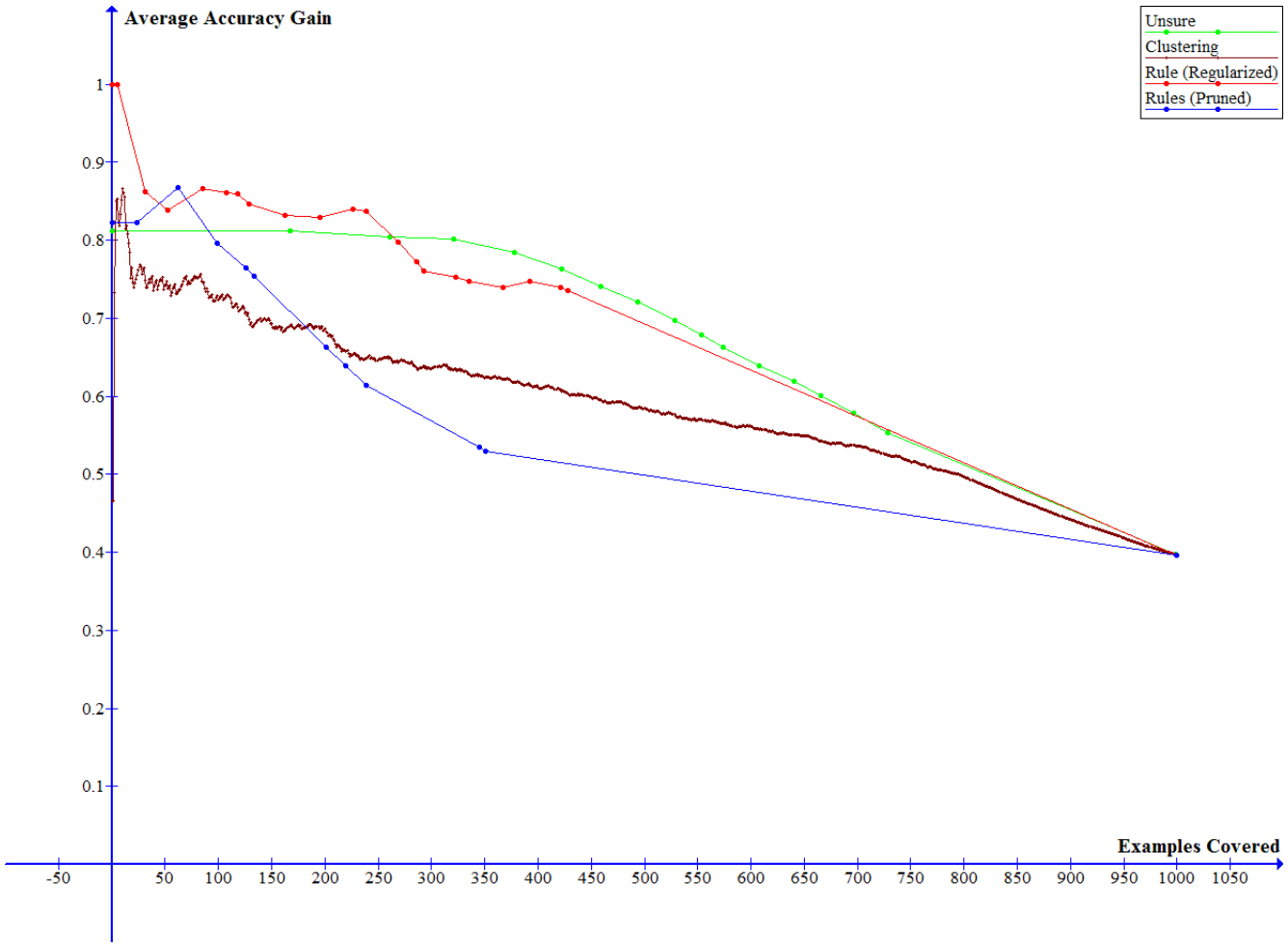

Figure 3: Accuracy Gain vs. Size on the Newsgroup Dataset

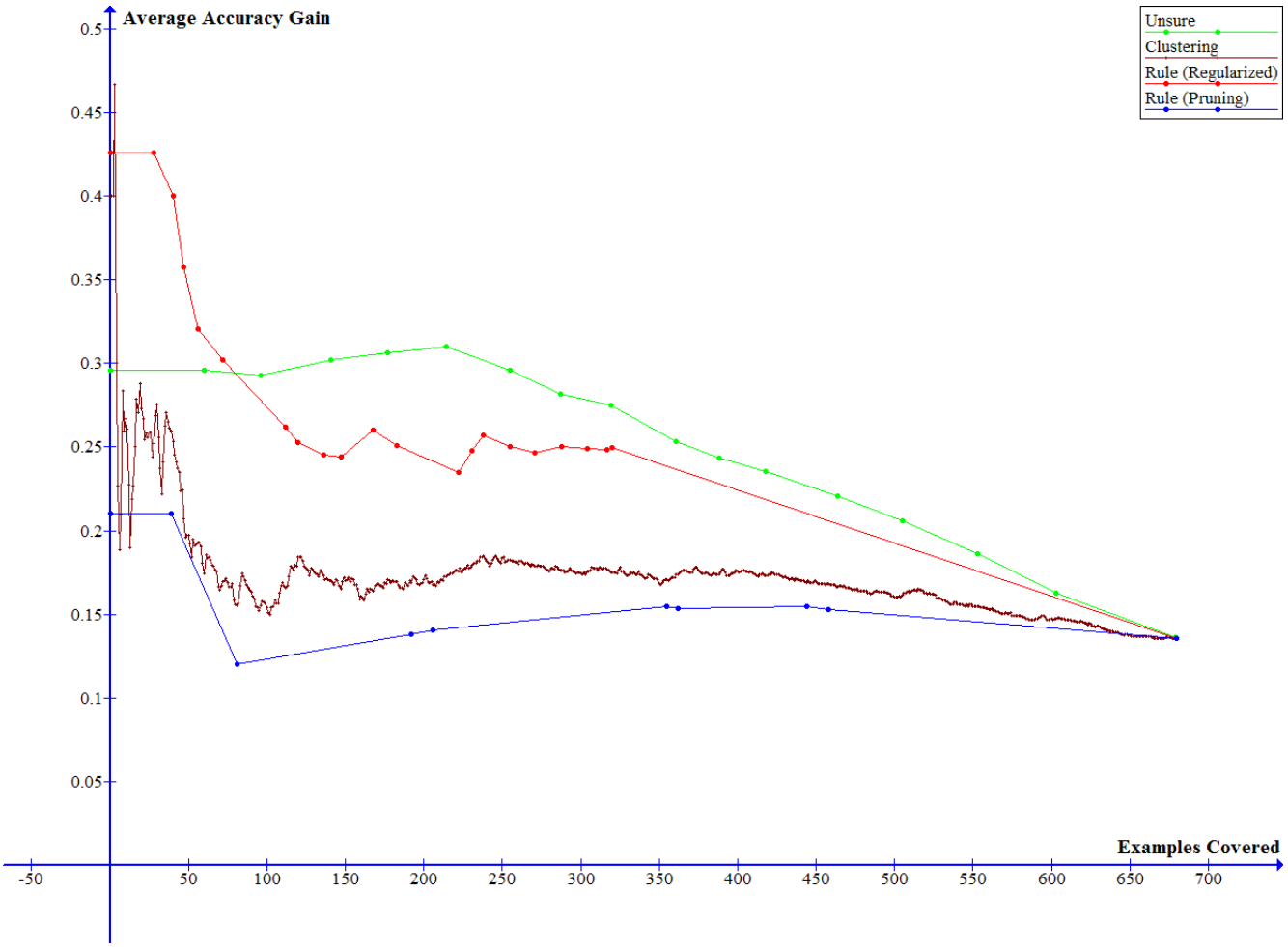

Figure 4: Accuracy Gain vs. Size on the Flowers Dataset

5.5 Example Output

Below we display some of the top rules used in this process generated on the 'before' datasets.

Table 1: Newsgroup Rules

Rule Number	Rule
1	(eng >= 1.000) AND (class == sci.crypt) AND (com><2.000) AND (alt <1.000) AND (black <1.000))
2	((class == sci.med) AND (pat <1.000) AND (cook <2.000) AND (perform <2.000) AND (infect <1.000))
3	((work <1.000) AND (post >= 2.000) AND (disc <1.000) AND (ca><1.000) AND (fal <1.000) AND (class = comp.sys.mac.hardware) AND (class != rec.sport.baseball) AND (ind <3.000))
4	((class == sci.electronics) AND (frequ <1.000) AND (volt <1.000) AND (gordon <1.000) AND (sim <3.000))
5	((class == rec.motorcycles) AND (rear == 0.000) AND (jonathan <1.000) AND (biker <1.000))

5.6 Discussion

For both datasets examples covered by the highest scoring rules had very high accuracy gain indicating they might indeed be important examples for a human users to examine. Significantly, even compared to selecting examples that had a zero accuracy rating, examples covered by the first few highest ranking rules

Table 2: Flowers Rules

Rule Number	Rule
1	((color675 <1.537) AND (color274 >= 1.883) AND (texture497 <0.528))
2	((texture556 >= 1.458) AND (texture286 >= 0.515))
3	((color237 <1.752) AND (color086 <1.779))
4	((color548 >= 1.955) AND (texture225 >= 0.619))
5	(color576 textless 0.839)

had superior accuracy gain. This lends credence to the notion that difficult to classify examples that are similar and can be grouped together in this manner are also examples that are easier to make accuracy gains for and thus, we hope, are useful for a user to examine. According to this metric pruning was an inferior strategies compared to using regularization when generating the rules and both were better than using clustering. The accuracy gain of the conjunctions declines quite quickly indicating that the conjunctions being found diminished in quality after the first few were used.

When considering the rules themselves, for the flower's dataset the rules are challenging to interpret since it is not clear how to interpret the features themselves. This reveals one of the weaknesses of this algorithm, if the features used are not human readable the benefits of attempting to define a subset of the space in a human readable way is lost. On the other hand it is still possible for a user to gain some benefit from examining the individual examples so the utility of the system is not completely lost.

For the newsgroups dataset the rules have the potential to be more helpful. Many of them agree with our intuitions, it makes sense that the classifier has a hard time classifying documents about electronics if that document does not contain words stemming from "volt" for "frequ". This knowledge might direct a user to examine those documents in particular in search of words that would be helpful when trying to complete this classification task.

5.7 Conclusion and Future Work

Human trials are a clear next step. The metric used above only measures one aspect of what was implemented, the tools outside of MisclassificationMiner have not been tested. Further the helpfulness of MisclassificationMiner has not been directly measured. A strong testing method would be to present users with an extended classification problem, taking raw data from a domain they have some familiarity with and arriving at a classification model. Ideally in such a dataset feature engineering would have a potentially high impact on classifier accuracy. By comparing user's ability to complete the task with and without the benefit of using Analyzer we would get a superior metric of the utility of the system.

Additionally a number of optimizations are possible. Clustering techniques could be improved considerably and might prove to be superior to rule building if retested, although they would still lack the advantage of having human interpretable rules. There is also room to optimize the rule search strategy in a way that better accounts for the variable accuracy rating of different examples. A better way to rate rules might be also be helpful, the graphs reveal several cases when adding a rule we rated as worse then the previous rules increased the average accuracy gain of the data points indicating the rule was actually better than some of the rules before it. Another direction that might have potential but has not been fully explored is using the rules generated as new features in the hopes of improving classifier accuracy.

Finally a potentially useful thing to do would be to add additional tools to Analyzer that would help users compare subsets of points isolated by rules with the points outside of that subset. This might including the ability to show statistics, such as the information of features among the points within the subset rather than without, that could help a user understand why the classifier was failing in that area.

Never-the-less the initial results show some promise, since feature engineering is an important aspect of machine learning building tools that can help users do this part well is of considerable importance. Although the remaining parts of this project are untested I remain optimistic that they have the potential to aid users to a considerable extent when completing machine learning projects.

5.8 Acknowledgments

Many thanks to Oren Etzioni for collaborating with me on this project.

6 Appendix

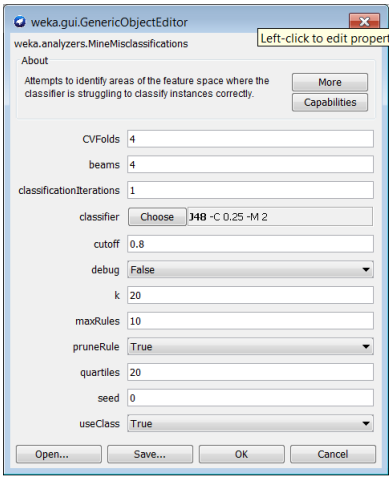

Figure 5: Tool Parameters Menu

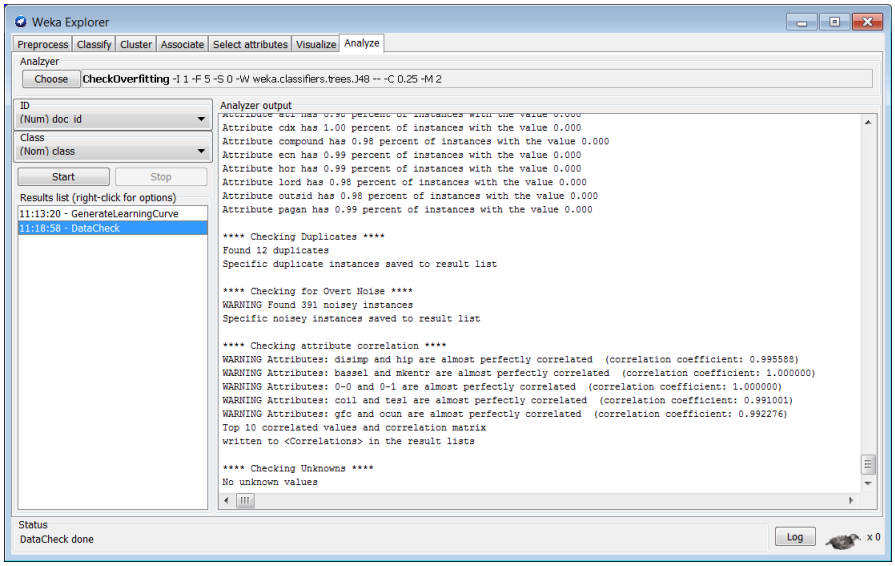

Figure 6: Example Tool Output

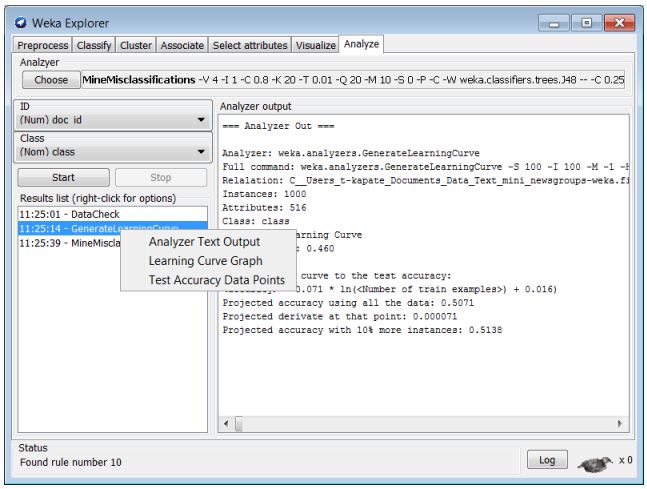

Figure 7: Result Menu

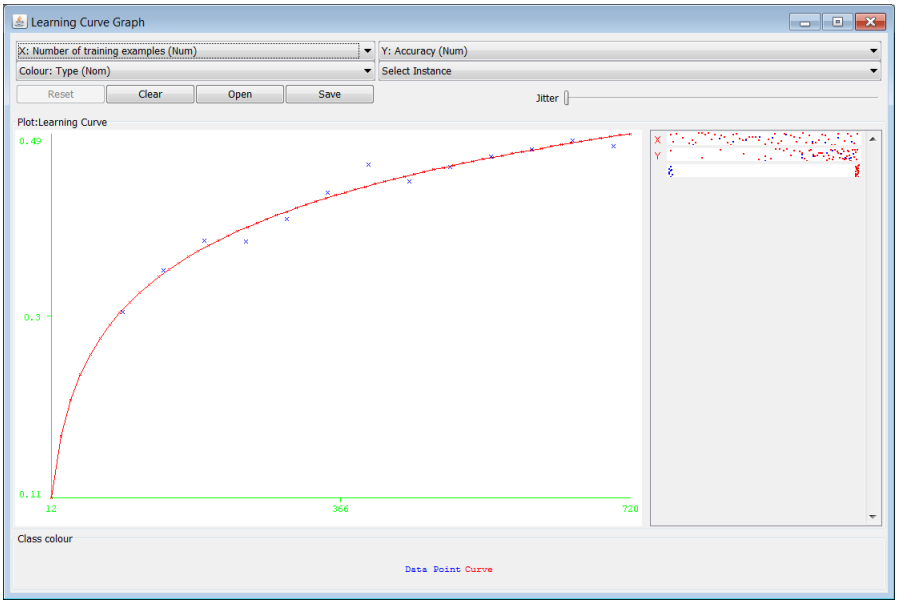

Figure 8: Learning Curve Graph

7 References

[1] Koren, Yehuda. "The Bellkor Solution to the Netflix Grand Prize." Netflix prize documentation (2009).

[2] Patel, Kayur, et al. "Investigating statistical machine learning as a tool for software development." Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems. ACM, 2008.

[3] Patel, Kayur, et al. "Using multiple models to understand data." Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two. AAAI Press, 2011.

[4] Patel, Kayur, et al. "Gestalt: integrated support for implementation and analysis in machine learning." Proceedings of the 23nd annual ACM symposium on User interface software and technology. ACM, 2010.

[5] Dekel, Ofer, and Ohad Shamir. "Theresa Hole in My Data Space: Piecewise Predictors for Heterogeneous Learning Problems." Proceedings of the International Conference on Artificial Intelligence and Statistics. Vol. 15. 2012.

[6] Segal, Richard B. Machine learning as massive search. Diss. 1997.

[7] Mierswa, Ingo, et al. "Yale: Rapid prototyping for complex data mining tasks." Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006.