End-User Feature Engineering in the Presence of Class Imbalance

Ian Oberst[1], Travis Moore[1], Weng-Keen Wong[1], Todd Kulesza[1], Simone Stumpf[2], Yann Riche[3], Margaret Burnett[1]

[1]School of EECS
Oregon State University
Corvallis, OR
{obersti, moortrav, wong, kuleszto, burnett}@eecs.oregonstate.edu

[2]Centre For HCI Design
City University London
London, U.K.
Simone.Stumpf.1@city.ac.uk

[3]Riche Design
Seattle, WA
yann@yannriche.net

ABSTRACT
Intelligent user interfaces, such as recommender systems and email classifiers, use machine learning algorithms to customize their behavior to the preferences of an end user. Although these learning systems are somewhat reliable, they are not perfectly accurate. Traditionally, end users who need to correct these learning systems can only provide more labeled training data. In this paper, we focus on incorporating new features suggested by the end user into machine learning systems. To investigate the effects of user-generated features on accuracy we developed an auto-coding application that enables end users to assist a machine-learned program in coding a transcript by adding custom features. Our results show that adding user-generated features to the machine learning algorithm can result in modest improvements to its F1 score. Further improvements are possible if the algorithm accounts for class imbalance in the training data and deals with low-quality user-generated features that add noise to the learning algorithm. We show that addressing class imbalance improves performance to an extent but improving the *quality* of features brings about the most beneficial change. Finally, we discuss changes to the user interface that can help end users avoid the creation of low-quality features.

Author Keywords
Feature engineering, class imbalance, end-user programming, machine learning.

ACM Classification Keywords
H5.2. Information interfaces and presentation (e.g., HCI): User Interfaces, I.2.6 Artificial Intelligence: Learning.

INTRODUCTION

Many intelligent user interfaces, such as recommender systems and email classifiers, use machine learning algorithms to customize their behavior to the preferences of an end user. We term this personalized system a *learned program* because it represents a set of decision-making behaviors for classification, generated by a probabilistic machine learning algorithm, for a specific end user. Each of these users has their own idiosyncrasies, so only they can provide the "gold standard" for the deployed learning system. These systems sometimes make mistakes; they recommend a movie the user dislikes, or judge mail from a family member to be SPAM. Improving the predictive accuracy of such learning systems poses the following challenge: how does an *end user*, who is typically not an expert in machine learning, improve a learned program's predictions?

End users possess far more knowledge about how to fix a learned program than the traditional method of simply labeling examples. In our past work [26] we found that the most common type of corrective feedback users wished to provide is the ability to create and delete features used by the machine learning algorithm.

Feature engineering is the process of designing features for use by a machine learning algorithm. A *feature* is a characteristic of training data instances that is informative for predicting a class label. Good feature engineering is critical to a machine learning algorithm's performance. Typically a machine learning expert, in conjunction with a domain expert, performs the feature engineering before the system is deployed to end users. Thus, by the time users see the system the features will have already been permanently set. In this paper we investigate how *end users* can contribute to the feature engineering effort, adding to and deleting from the features used by the machine learning algorithm *after deployment*.

Our past work on using learned programs to classify email [17] demonstrated that *class imbalance* is a major problem. Class imbalance affects any domain where classes are unevenly distributed in the data. In a common case of class imbalance, the majority of the data points belong to one dominant class. Although class imbalance occurs frequently in many real-world data sets, machine learning researchers often overlook it even though standard learning algorithms typically perform poorly in the face of this problem. When one category of data is over-represented in the training set, this same over-representation cascades into the program's predictions, and the learned program often just predicts the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2009, April 4–9, 2009, Boston, MA, USA.
Copyright 2009 ACM 978-1-60558-246-7/08/04…$5.00

most dominant class seen in the training data. In order to address this problem, we explored how the learned program, with features engineered by end users, can alleviate this problem using three methods for dealing with class imbalance.

In order to explore these ideas we implemented a learning system to assist with *coding*, a familiar task in the fields of psychology, social science, and HCI. When coding, researchers categorize segmented portions of verbal transcripts of a research study into codes as part of an empirical analysis process. We developed a prototype known as *AutoCoder* that trains on previously coded segments of a transcript and predicts the codes the user would likely assign to future segments.

The coding domain is especially interesting for two reasons. First, performing the coding task manually is extremely tedious and time consuming, which means the user has a strong incentive to improve the accuracy of the learned program. Second, coding tasks are often different for each study, meaning the user must engage in feature engineering. Since each study is so unique and tailored to the verbal data collected, machine learning experts do not have the information they would need to adequately complete feature engineering before deployment.

The coding domain is also extremely challenging for machine learning. First, because coding is so time consuming, it is not viable for researchers to manually create a large training set of labeled segments for each study. Consequently, systems like the AutoCoder must learn from a limited amount of training data. Second, class imbalance in coding is the norm. Codes are seldom evenly distributed within a verbal transcript. Even worse, researchers interacting with the AutoCoder may generate a class imbalance by focusing on only one code at a time, searching out segments corresponding to just that code as a cognitive strategy. Finally, the auto-coding task is a complex learning problem because sequential relationships exist between segments and assigned codes, and these play a role in the user's code selection.

This paper explores the impact of incorporating user-generated features into an auto-coder system. We present and evaluate three approaches for dealing with class imbalance. Finally, we identify two characteristics of detrimental user-defined features, and propose ways to leverage these characteristics to encourage end-user feature engineering that will improve the learning system.

RELATED WORK

A variety of methods, spanning the spectrum of formality and richness, have been proposed to obtain user feedback. These methods include interactions in natural language [2], editing feature-value pairs [6,21] and *programming by demonstration* [9,19]. Once the user feedback has been obtained, it needs to be incorporated into the machine learning algorithm. Traditional techniques for incorporating a user's feedback allow the user to interactively create new training examples [12] or to label informative training examples as chosen by the learning algorithm [7]. These approaches interact with the user through labels on the training examples.

A different way to incorporate feedback is to allow *rich feedback*, which involves incorporating more expressive forms of corrective feedback that can cause a deeper change in the underlying machine learning algorithm. Forms of rich feedback include constraint-based approaches [8,15,25], similarity metric learning from rich user feedback [13], clustering documents using a set of user-selected representative words for each class [20], and allowing the user to directly build a decision tree for the data set with the help of visualization techniques [30].

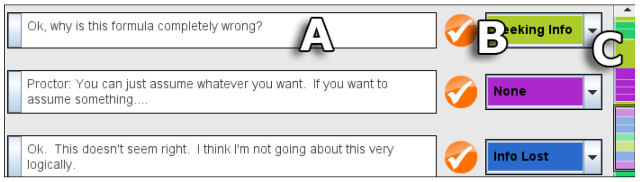
Several methods have also allowed rich feedback in the form of allowing the user to modify features. Our past work [17,27,28] allows end users to re-weight features used by the learning algorithm. Raghavan and Allan [23] propose using a human expert to identify training instances and features that the machine learning algorithm then focuses on. Unlike our current work, their features are predefined and not created by the end user. Furthermore, they do not deal with class imbalance. Roth and Small [24] allow users to replace features corresponding to words or phrases with an abstraction called a Semantically Related Word Lists (SRWL). A SRWL consists of a list of words that are semantically related to each other. For instance, the words North, East, South and West can be replaced by a SRWL corresponding to "Compass Direction". The user can also refine the SRWLs used in the application. In our work, we enable end users to create features formed by combinations of words and/or punctuation, rather than replacing words or phrases with their SRWL abstraction.

Focusing on the particular domain of auto-coding, the TagHelper auto-coding system is highly accurate with extensive training data, but not for categories with few training examples [11]. Obtaining enough training data, however, is problematic in many cases because manual coding is so time-consuming. Our research addresses this issue by attempting to reduce the need for large training data by instead fine-tuning the features.

METHODOLOGY

The AutoCoder Prototype

We developed an AutoCoder prototype that allows users to code segmented text transcripts (Figure 1 A) using a



Ok, why is this formula completely wrong? — eeking Info

Proctor: You can just assume whatever you want. If you want to assume something.... — None

Ok. This doesn't seem right. I think I'm not going about this very logically. — Info Lost

Figure 1: The basic AutoCoder prototype showing a series of segments (A), their corresponding codes (B), and an overview of the transcript's codes (C).

predefined code-set (Figure 1 B). Codes shown on the interface are colored to give users an overview of their coding activity. These colors are replicated in the navigation scrollbar to provide a summary of each code's occurrence over the whole transcript (Figure 1 C). The user is able to manually assign a code to each segment; after a set number of segments have been coded, the program will attempt to predict the codes of the remaining unlabeled segments. The learned program updates its predictions as the user provides label corrections and suggests new features. To agree with a prediction, the user is able to approve it by placing a checkmark. Approved predictions are then added to the set of training data to improve the accuracy of future machine predictions.

In addition to this basic ability to code a transcript, we devised widgets that explain the program's predictions and allow users to suggest features to the learned program.

Explaining Predictions

We designed the AutoCoder such that it could describe the most relevant pieces of information leading to its decisions. AutoCoder provides a list of **Machine-generated Explanations** (Figure 2 W2) for each prediction, informing users of the features that most influenced its decision. A computer icon reminds users that these explanations are based upon machine-selected features.

End users can select any segment to view explanations about its current prediction, resulting in the Machine-generated Explanation with the highest-weighted features appearing beneath the segment text. The AutoCoder sorts the explanations list according to the most influential features; interested users can click a button to progressively show more. If a user disagrees with the influence of a feature, she can delete the feature, thereby excluding it from the learned program's future predictions.

A **Prediction Confidence** (Figure 3 W3) widget displays a pie graph outlining the program's probability of coding a given segment using any of the possible codes. A confident prediction is indicated by a predominantly solid color within the pie graph, while a graph containing an array of similarly sized colors signifies that the program cannot confidently determine which code to apply. If the program does not display a high confidence in its prediction, it is a cue to end users that they may want to implement measures to improve the program's logic.

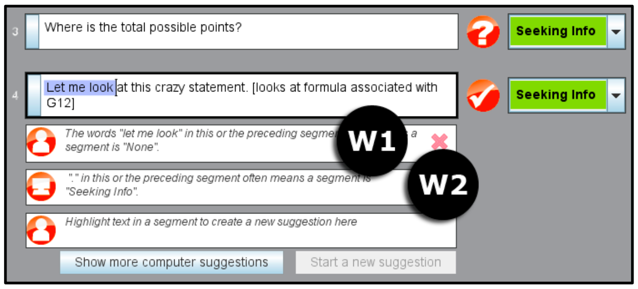
The **Popularity Bar** (Figure 3 W6) specifically addresses the problem of class imbalance. It represents the proportions of each code amongst the user-coded and machine-predicted segments. The left bar represents code popularity among user-coded segments, i.e., the proportion of codes the user has manually selected. The right bar contrasts code popularity among machine-predicted segments, i.e., the proportion of codes the machine is predicting for remaining segments.

End-User Feature Engineering

We gave end users the ability to explain to the machine why a segment should be coded in a particular manner and, in doing so, add their own new features to the program. These **User-generated Suggestions** (Figure 2 W1) are integrated into the learned program's logic as features for determining the codes of the remaining segments. They are distinguished from Machine-generated Explanations by a "human" icon.

By selecting a sequence of text, users are able to create features spanning adjacent segments to model relationships between words and segments, as well as including non-consecutive portions of text in a single segment. An example of a possible user-generated suggestion that incorporates such a relationship is: "*'?' in the preceding segment followed by 'OK' in this segment often means this segment is coded as 'Info Gained'*". We also allow users to add features consisting of single words or combinations of words and punctuation to the program. Figure 2, for example, shows a user creating a word combination feature by selecting a contiguous portion of text.

To reflect how a user suggestion impacts the overall coding task, we designed an **Impact Count Icon** (Figure 3 W4) to reflect how many predictions are affected by each suggestion. For instance, a suggestion stating that *"?"* implies a particular code will impact each segment containing a *"?"*, likely resulting in a high impact count. A suggestion with a high impact count is important because it will affect many predictions. To indicate which predictions are affected by a suggestion, AutoCoder highlights the relevant segments and their corresponding sections in the scrollbar.



3 Where is the total possible points?

Seeking Info

4 Let me look at this crazy statement. [looks at formula associated with G12]

Seeking Info

The words "let me look" in this or the preceding segme [illegible] a segment is "None".

"." in this or the preceding segment often means a segment is "Seeking Info".

Highlight text in a segment to create a new suggestion here

Figure 2: An AutoCoder User-generated Suggestion (W1) and Machine-generated Explanation (W2).

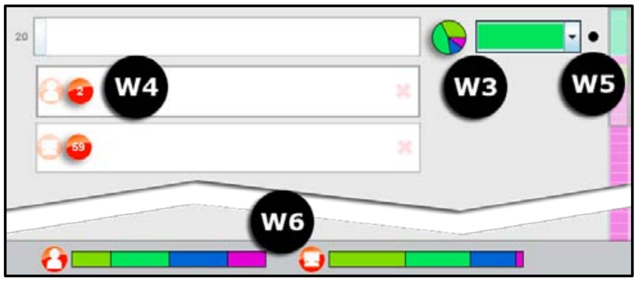


Figure 3: The Prediction Confidence widget (W3), Impact Count Icons (W4), Change History Markers (W5), and Popularity Bar (W6).

To help users understand the specific implications of their recent actions, **Change History Markers** (Figure 3 W5) provide feedback on where changes in the program's predictions have recently occurred. A black dot is displayed adjacent to the most recently altered predictions. A similar black mark is shown beside each segment's respective section in the scrollbar, giving the user an overview of every change throughout the program. As the user makes changes that *do not* alter the machine's prediction for a segment, these marks gradually fades away.

Data Collection

We collected data via a user study [16] that used the AutoCoder prototype to explore which types of information are thought helpful and understandable by end users attempting to debug machine-learned programs. This study also provided log data that we used off-line to investigate how to improve the underlying machine learning algorithm by incorporating user-defined features.

We recruited 74 participants (40 males, 34 females) from the local student population and nearby residents for our study. None possessed experience with machine learning algorithms and only one had previous experience with a task similar to coding.

All participants received a tutorial consisting of a 30-minute introduction to coding, the coding set itself, and the prototype's functionalities. This was followed by two 20-minute sessions where users coded a transcript with the prototype. A different transcript was used during each session. We assigned the transcript order randomly across our participants.

In addition to a set of questionnaires to obtain subjective user responses to the prototype, we logged each participant's interactions with the software, including the features that they suggested to the program.

Machine Learning Algorith

Our auto-coding task is an example of a *sequential supervised learning task* [10] in which the order of the data plays an important role. Sequential supervised learning algorithms leverage the sequential relationships in data to improve prediction accuracy. To see these sequential relationships in our auto-coding domain, consider the screenshot in Figure 2. Our AutoCoder represents a transcript as a sequence of segments, shown on the left, and a corresponding sequence of codes, shown on the right. In order to predict the code for a segment, the machine learning algorithm needs to consider the text in that segment. However, the text in neighboring segments often influences the code for the current segment. For instance, the text in both segments 3 and 4 may influence the code assigned to segment 4. To deal with these neighboring relationships, we consider the text in a *sliding window* of segments that influence the current code, where in our implementation this sliding window includes the current segment and the previous segment. The underlying machine learning algorithm needs to convert the text in the sliding window into features. We take the union of the words in the current and previous segments and produce a bag-of-words representation from this union. This bag-of-words representation results in features corresponding to the presence/absence of individual words in the sliding window.

In addition to neighboring segments, neighboring codes are also informative for predicting the code for the current segment. As an example, the "Seeking Info" code in segment 3 may make it more likely for segment 4 to be assigned a "Seeking Info" code. As a result, we incorporate the code assigned to the previous segment as a feature for the sequential supervised learning algorithm.

Baseline Algorithm

Sophisticated sequential supervised learning algorithms such as Hidden Markov Models (HMMs) [22] and Conditional Random Fields (CRFs) [18] are unsuitable in our situation for two reasons. First, predictions made by these approaches are difficult to explain to an end user, especially which features matter most, which is an important step in helping end users create predictive features. Second, sophisticated supervised learning approaches such as CRFs and HMMs involve computationally expensive learning and inference processes.

We chose a much simpler approach by using the "recurrent sliding window" technique [10] that converts a sequential supervised learning problem into a standard supervised learning problem. This conversion takes the features derived from the sliding window, along with the code from the previous segment, and creates a training example for a standard supervised learning algorithm. This conversion makes the learning and inference process much more efficient and it allows the use of standard machine learning algorithms, such as naïve Bayes, that can easily generate explanations understandable to end users. In our experiments, we used a *Recurrent Sliding Window Naïve Bayes* classifier (abbreviated RSW-NB), which assumes that the features are conditionally independent given the class label. We use the RSW-NB classifier, without user-defined features, as the baseline classifier in our experiments.

Incorporating User-Defined Features

The most straightforward way to incorporate user-defined features is to add them to the list of features used by the RSW-NB algorithm. This approach, however, does not cause the learning algorithm to pay special attention to the user-defined features since they are treated as equal to "regular" bag-of-words features. In order to make the algorithm heed the user, we implemented a variant of RSW-NB called *Weighted RSW-NB* (WRSW-NB), which causes the underlying algorithm to put more emphasis on the user-defined features for the specified class. We make the following assumption when a user adds a feature X for code y: the presence of feature X is important for predicting code y. This extra emphasis can be easily achieved with naïve

Bayes by assigning maximal weight to the parameter *P(X = present | Code = y).*

Class Imbalance Correction

The most common approaches for dealing with class imbalance for machine learning algorithms involve oversampling and/or undersampling from the training data. These methods vary according to how data points are over/undersampled. We applied the three methods described below. All three class imbalance methods exhibit variability in the predictions due to randomness in the sampling of data points. In order to make the predictions less volatile, we run the machine learning algorithm with the class imbalance correction 100 times and take a majority vote among the runs for the class label.

Over/Undersampling

The first method of correcting for class imbalance consists of applying a mixture of both oversampling and undersampling [3]. This method works by computing the mean number of data points in the minority classes, where a minority class is any class whose number of data points is less than the class with the largest number of data points. Each class in the training data is then oversampled or undersampled until the number of data points in that class is equal to the me

We chose to use a mixture of both undersampling and oversampling because of the amount of class imbalance that exists in our data. Some minority classes have so few data points that undersampling the majority classes to this extreme would produce insufficient training data for the classifier. We instead chose to compute the mean over all the minority classes and then oversample all classes that were smaller than the mean and undersample all classes that were larger than the mean. This prevented the majority classes from being undersampled too harshly while allowing us to still oversample the minority classes.

SMOTE

The SMOTE algorithm [3,4] has previously been shown to work effectively in dealing with class imbalance on continuous data sets from the UCI Repository [1]. SMOTE works by taking a target data point and using its *k* nearest neighbors to construct new synthetic data points. SMOTE was originally designed for continuous features, though a version for nominal features was proposed in [3,5], but their method only allows for the creation of a single distinct synthetic data point in terms of its features. In order to create synthetic data points with more diverse features, we modified the algorithm to randomly select, with replacement, a subset of the *k* nearest neighbors to create a panel of "voters" used to create the synthetic data point. The panel of "voting" neighbors votes on each feature to determine if it is present or absent in the synthetic data point. In a tie, the presence or absence of the feature is determined randomly.

By randomly selecting the size of the panel to be between 1 and *k*, we can then generate multiple synthetic data points. We also chose to use Hamming Distance [14] as our distance metric for the *k* nearest neighbor algorithm as we found it performed better on our data than the metric originally used by Chawla et al. [4].

Extreme SMOTE

We created a variant of SMOTE called Extreme SMOTE that oversamples the minority classes by randomly selecting from data points the actual *k* nearest neighbors rather than creating synthetic data points.

Off-line Evaluation of Classifiers

We developed eight classifiers that incorporate user features (w/UF suffix). Six of the eight classifiers adopt class imbalance corrections. The eight classifiers are:

- RSW-NB w/UF
- RSW-NB w/UF + Over/undersampling
- RSW-NB w/UF + SMOTE
- RSW-NB w/UF + Extreme SMOTE
- WRSW-NB w/UF
- WRSW-NB w/UF + Over/undersampling
- WRSW-NB w/UF+ SMOTE
- WRSW-NB w/UF + Extreme SMOTE

We evaluated each of these against the Baseline algorithm, a RSW-NB without user features or class imbalance correction.

We used the data logs from our user study for training and testing the classifiers. The data of the first transcript that participants coded was used as a training set while the second transcript was used as a test set. Each classifier was trained on and tested against the data from a single participant.

We pre-processed the training data so that only segments with explicit user labels, either assigned by users themselves or predictions by the classifier that were verified by the user, were used. Since the number of bag-of-words features (i.e. features based on individual words) in a transcript is very large, we applied feature selection using information gain to reduce the number of features. We chose to use the 75 most informative bag-of-words features, since an empirical evaluation of the average effect proved to be the highest at that value. We did not perform feature selection of the user-defined features; instead, we incorporated all the user-defined features into the classifier.

Evaluation Metric

Due to class imbalance, using a metric of prediction accuracy can be misleading. A "dummy" classifier that simply predicts the dominant class in the training data can appear to have very good accuracy. A better evaluation metric is precision and recall. We can summarize the tradeoff between precision and recall as a single number in the form of an F1 score, where F1 = (2*precision*recall)/(precision+recall).

A further decision is how to compute overall precision and recall when there are more than two classes. Macro-averaging is a process of averaging an evaluation metric by class [29]. For example, macro-averaged F1 is computed by calculating the F1 score for each class, then taking the average of those values across all classes. Another alternative is micro-averaging, which is skewed in favor of more dominant classes and is thus unsuitable in our class-imbalanced data. For the evaluation of our classifiers, we calculated the difference in macro-averaged F1 scores between a classifier and the Baseline. We call this metric *macro-F1 delta* for brevity. To evaluate the classifiers' overall performance, we took the mean of the macro-F1 deltas over all participants.

RESULTS

Adding User Features and Addressing Class Imbalance

The simplest way to involve the user in feature engineering is to start paying attention to the user's suggestions. To measure the impact of adding user-defined features (without any class imbalance corrections), we compared the mean macro-F1 deltas for RSW-NB w/UF and WRSW-NB w/UF (Figure 4). RSW-NB w/UF shows a modest improvement (+0.9%) while WRSW-NB w/UF shows a decrease (-2.4%) from the Baseline algorithm. This result means that placing strong emphasis on user-defined features (as done in WRSW-NB w/UF) can cause incorrect predictions, if class imbalance resulting from users' tasks or behaviors is ignored.

It is therefore vital to consider addressing class imbalance. Figure 5 illustrates the effects of user feature engineering when class imbalance corrections are performed. Any methods that we chose for RSW-NB w/UF to address class imbalance resulted in an increase in the mean macro-F1 delta, where the simplest approach of over/undersampling resulted in a significant improvement against the Baseline (Wilcoxon: V=32, N=73, *p*=0.0245). In contrast, more complex approaches are needed to improve the mean macro-F1 delta score for WRSW-NB w/UF: only SMOTE and Extreme SMOTE resulted in increases in performance. Furthermore, just by addressing class imbalance a poor algorithm (such as WRSW-NB w/UF in Figure 4) can perform better. Thus, methods to address class imbalance problems need to be carefully chosen, but Extreme SMOTE appears to yield reliable improvements for both weighted and unweighted algorithms when taking user-defined features into account.

Even though the mean macro-F1 increases or decreases appear modest, individual users may still benefit from large gains. For example, WRSW-NB w/UF+SMOTE could lead some individual users to benefit from gains as large as 30%, far outstripping the mean macro-F1 delta over all users of only 0.1% (Table 1). On the other hand, some users suffered macro-F1 decreases as low as -17%. In general, algorithms with class imbalance corrections tend to produce much bigger changes affecting individuals than if class imbalances corrections were not applied.

Applying class imbalance corrections can address some problems, but not all. For example, if we compare one of the best performing classifiers, RSW-NB w/UF + over/undersampling with the related RSW-NB w/UF (without class imbalance corrections) only four more participants would have experienced macro-F1 increases due to class imbalance corrections. Taking this and the previous result together, we would like to make use of the more pronounced changes affecting individuals when class imbalance corrections are applied, while at the same time ensuring that more participants see more correct predictions.

Influence of Individual User Features

If we could remove noise such as irrelevant features from the data, this may lead to more correct predictions that are reflected in positive macro-F1 deltas. It is therefore

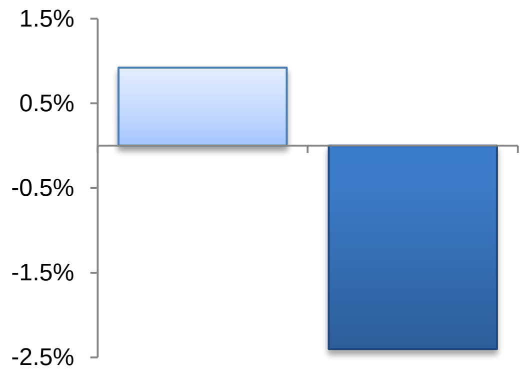


Figure 4: Mean macro-F1 deltas for RSW-NB w/UF (left) and WRSW-NB w/UF (right).

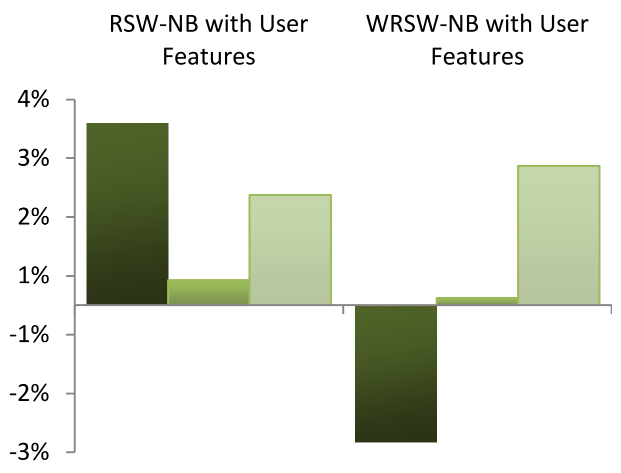


Figure 5: macro-F1 deltas for RSW-NB w/UF (left) and WRSW-NB w/UF (right) with the effects of different class imbalance adjustment methods shown in the different columns (Over/Undersampling is darkest, SMOTE is midrange, Extreme SMOTE is lightest).

Algorithm	Biggest macro-F1 decrease for an individual	Biggest macro-F1 increase for an individual	Average macro-F1 delta over all users	No. of macro-F1 decreases	No. of macro-F1 increases
RSW-NB w/UF	-12%	13%	0.9%	28	38
RSW-NB w/UF + over/undersampling	-12%	29%	3.0%	31	42
RSW-NB w/UF + SMOTE	-22%	29%	0.4%	39	34
RSW-NB w/UF + Extreme SMOTE	-14%	27%	1.9%	35	38
WRSW-NB w/UF	-25%	20%	-2.4%	42	31
WRSW-NB w/UF + over/undersampling	-25%	20%	-2.3%	44	30
WRSW-NB w/UF + SMOTE	-17%	30%	0.1%	39	34
WRSW-NB w/UF + Extreme Smote	-16%	27%	2.4%	33	40

Table 1: Summary of changes to the macro-F1 delta for individual users for the various classifiers.

important to consider how individual features that participants added contribute to the deltas. We conducted an offline, retrospective analysis which involved, for each user, training a classifier by adding individual user-defined features one-by-one and then calculating the resulting macro-F1 delta from the test set. Since our classifiers incorporate user-defined features in different ways, the impact of these features is classifier-dependent.

For each feature we can determine whether it decreased or improved macro-F1 i.e. whether it was *predictive* or *non-predictive*. Table 2 shows that end users are capable of coming up with predictive features and that on average, there are slightly more predictive features defined by users than non-predictive ones. A single predictive feature can contribute substantially to accuracy (17.8% for RSW-NB w/UF, 26.7% for WRSW-NB w/UF); a non-predictive feature can cancel out this positive effect.

To illustrate the effects that only adding predictive features could have on performance, we greedily added a single user-defined feature to the WRSW-NB classifier for each user until no more improvement in the macro-F1 score was possible. In doing so, we can approximate the “best” classifier formed out of the set of user-defined features. From our data, this “best” classifier allows us to obtain a mean macro-F1 delta across all users of 19% for RSW-NB w/UF (Wilcoxon: V=2591, N=73, p<0.001) and 11% for WRSW-NB w/UF (Wilcoxon: V=2389, N=73, p<0.001).

Determining predictive or non-predictive features based on their contribution to the macro-F1 score in an end-user situation is infeasible. First, it is time-consuming, as it requires adding user-defined features one at a time and then retraining the classifier each time. Most importantly, the macro-F1 score can only be calculated if all of the labels for the test data are available. Clearly, this requirement is unrealistic as it requires the coding task to be already completed and can only be conducted with hindsight.

Identifying non-predictive user-defined features

To aid the end user in identifying non-predictive features during feature engineering, the system could focus on traits of these features. In this section, we define two characteristics of features that have a negative effect on predictions. We investigate the usefulness of these characteristics by filtering out features based on these characteristics and measuring the impact on accuracy.

We inspected the non-predictive user-defined features in our data and found that they had two main characteristics. To aid with the definitions of these characteristics, recall that when a user creates a feature, she indicates that the user-defined feature F is important for class C.

Characteristic 1: Poor test data agreement

In many cases in the test data, we noticed that the user-defined feature F is not present in segments assigned to class C.

We compute the *test data agreement* as:

$$\frac{\#\ of\ test\ segments\ with\ feature\ F\ and\ class\ label\ C}{\#\ of\ test\ segments\ with\ feature\ F}$$

If this value falls below a threshold then the feature is considered to have poor test data agreement.

	RSW-NB w/UF	WRSW-NB w/UF
Average no. of predictive user-defined features	13.16	10.83
Highest macro-F1 increase after adding user-defined feature	+17.8%	+26.7%
Average no. of non-predictive user-defined features	10.64	9.61
Highest macro-F1 decrease after adding user-defined feature	-14.0%	-29.5%

Table 2: Number of predictive / non-predictive user features for the RSW-NB w/UF and WRSW-NB w/UF classifiers over all users and their effects on the macro-F1 score. On average, users added a total of 17.28 features.

Characteristic 2: Under-representation of a user-defined feature in its assigned class in test data

A user-defined feature *F* is considered to be under-represented in the test set if it is absent from a large percentage of test data segments with class label *C*. For example, if a user defines the feature "I don't know" to be important for the class "Info Lost", it is considered to be under-represented if it is absent in most of the segments classified under "Info Lost" in the test data.

We compute *under-representation* as:

$$\frac{\#\ of\ test\ segments\ with\ feature\ F\ and\ class\ label\ C}{\#\ of\ test\ segments\ with\ class\ label\ C}$$

If this value falls below a threshold, then the feature is under-represented.

These two characteristics indicate that non-predictive features are those in which the assumption that the presence of the user-defined feature is important for the specified class no longer holds in the test data. As a result, the absence of a user-defined feature biases the prediction away from the user-specified code for that feature. This change can be due to a variety of reasons. For instance, two transcripts may contain different words and phrases, resulting in a different set of salient features for the code set. Another possibility is that an individual's interpretation of a code may be inconsistent during the entire coding process, resulting in a changing set of features being salient for a certain code over the entire transcript.

We evaluated the usefulness of these characteristics by applying them to the non-predictive features in our data, and removing the ones that fell below the thresholds. (We use a threshold of 0.5 for test data agreement and a threshold of 0.1 for under-representation.) When combined, these two characteristics are able to filter out many of the non-predictive features. If we rank the top 100 worst user-defined features added by all the users in our study, 7% fall under low test set agreement only, 9% fall under under-representation only, 79% fall under both characteristics, and 6% fall under neither. Thus, 94% of the 100 worst user-defined features can be filtered out if either characteristic 1 or 2 is met. However, while this filtering is successful at removing non-predictive user-defined features, it is somewhat indiscriminate as it removes predictive user-defined features as well. Of the top 100 best user-defined features added by all users in our study, 64% are removed by both of these criteria.

To assess the effects of removing user-defined features based on these characteristics on the macro-F1 deltas, we removed user-defined features that matched both characteristic 1 and 2 from the RSW-NB w/UF and WRSW-NB w/UF classifiers. This resulted in 52 out of 74 participants having positive macro-F1 deltas for WRSW-NB w/UF Filtered, as compared to only 31 before filtering them out. Using this approach also increased the largest macro-F1 delta to 32.2%, instead of 20% before filtering. Consequently, the mean macro-F1 delta over all users increased to 5% compared to the Baseline, which has no user features, (Wilcoxon: V=1934, N=73, $p<0.001$) and it also represents a significant improvement over the WRSW-NB w/UF without filtering (Wilcoxon: V=2276, N=73, $p<0.001$). This approach also works for RSW-NB w/UF, which does not weight the user-defined features. RSW-NB w/UF Filtered had an average increase in macro-F1 of 5%, showing a significant improvement over the Baseline (Wilcoxon: V=1048, N=73, $p=0.0062$).

Using this approach to improve the machine learning algorithm has several advantages. It is simple and can therefore be performed quickly in real time without retraining the classifier. Furthermore, although test set agreement and under-representation can only be exactly computed in hindsight, these two criteria are indicators of discrepancies between the classifier's predicted class labels and the user's class labels. We can approximately detect these discrepancies using the parts of the test transcript that have already been coded by the user rather than waiting for the *entire* test transcript to be coded. In addition, we can compute other surrogates for test data agreement and under-representation that do not require hindsight. These indicators can be incorporated into the design of user interfaces that could help the end user avoid creating non-predictive features.

IMPLICATIONS FOR DESIGN

In this work, we have taken the position that end users can have a hand in feature engineering. To support this possibility, just as one would not expect a machine learning expert to add features during feature engineering without first understanding how relevant the features are, it is unreasonable to expect end users to add features without understanding their impact. Our prototype's user interface aims to help users understanding the impact of features they think should be considered by the machine.

However, our results show that allowing the user to add features to a machine learning algorithm is not a silver bullet. Although some of our participants' features were beneficial, many were detrimental to accurate classification. While addressing class imbalance alleviated this problem to an extent, it was improving the *quality* of features that brought about the most beneficial change.

We were able to identify two quality characteristics that had a detrimental impact on predictions: 1) poor test agreement and 2) underrepresentation. Intelligent user interfaces can act upon both of these characteristics to improve the predictive accuracy of a learned program.

Regarding the combination of both characteristics, a user interface could easily check whether a new user-defined feature appears anywhere in the test data. In our data, if the feature was completely absent from the test data, then the feature fell into both characteristics 1 and 2, and was very likely to hurt performance.

A more robust approach to approximately detect both criteria would be for the system to track discrepancies between the current state of the classifier and the classes being assigned by the user. As the user provides feedback on segments, the system could check the user-defined features against the already-coded section of the transcript to see if the features met either criterion. If the number of these discrepancies passed a certain threshold, the system could alert the user of inconsistencies between the user's perception of the feature's importance and what actually existed in the test data.

With either of these additions the user would be better equipped to provide genuinely useful features to the machine, increasing the machine's performance and the user's trust in the system.

CONCLUSION

Intelligent systems that customize themselves to an end user's behavior are different from other machine learning situations: only one person knows the ground truth as to what is appropriate behavior, and that person is the end user. Thus, end users themselves must have the final say in feature engineering.

Therefore, in this paper we investigated ways to enable end users to improve machine learning algorithms via feature engineering *after* the machine learning system has been deployed. Our results showed that adding user-defined features resulted in a minor improvement to the learning algorithm's performance. However, this performance increased when the issues of class imbalance and low quality user features were addressed.

Class imbalance corrections, such as in SMOTE and Extreme SMOTE, were particularly important given our assumptions about user intent. Specifically, we assumed that the presence of a user-defined feature was important for the user-specified class. We modified the learning algorithm to focus more attention on the user-defined features, but this resulted in a decrease in performance. We discovered that a solution to this problem was to also employ sophisticated methods for dealing with class imbalance.

Our results also showed that end users were indeed capable of creating predictive features, but if our algorithm used these features indiscriminately, they degraded the learning algorithm because of the mix of non-predictive features with predictive ones. Nevertheless, end users created more predictive than non-predictive features, which could be leveraged to good effect. One approach is to identify and filter out non-predictive features. The majority of non-predictive features can be identified by two characteristics: test set agreement, and under-representation. Identifying these two characteristics may seem to require hindsight in the form of actual test set labels. However, we propose two approximations to these characteristics: 1) monitoring test set occurrence to quickly remove detrimental features, and 2) keeping track of the number of disagreements between the user and the current classifier.

Our results provide specific steps for improving the design of intelligent user interfaces so that they can be feature-engineered by end users after deployment. We believe such improvements are key to end users' trust, usage, and realization of the full potential of user interfaces that support users by learning from their behavior.

ACKNOWLEDGMENTS

We thank the participants of our study, along with Kevin McIntosh and Forrest Bice for their assistance. This work was supported by NSF IIS-0803487 and by the EUSES Consortium via NSF CCR-0325273.

REFERENCES

1. Blake, C., Merz, C. UCI Repository of Machine Learning Databases. http://www.ics.uci.edu/~mlearn/~MLRepository.htm. Department of Information and Computer Sciences, University of California, Irvine. 1998.
2. Blythe, J. Task learning by instruction in Tailor. *Proc. IUI*, ACM, (2005), 191-198.
3. Chawla, N.V., Bowyer K.W., Hall, L.O., Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research* 16 (2002), 321-357.
4. Chawla, N.V. C4.5 and imbalanced data sets: Investigating the effect of sampling method, probabilistic estimate, and decision tree structure. *Proc. ICML* (2003).
5. Chawla, N.V., Lazarevic, A., Hall, L.O., Bowyer, K.W. SMOTEBoost: Improving prediction of the minority class in boosting. *Proc. Principles of Knowledge Discovery in Databases* (2003), 107-119.
6. Chklovski, T., Ratnakar, V. and Gill, Y. User interfaces with semi-formal representations: A study of designing argumentation structures. *Proc. IUI*, ACM, (2005), 130-136.
7. Cohn D. A., Ghahramani, Z., and Jordan, M. I. Active learning with statistical models. *J. Artificial Intelligence Research* 4, (1996), 129-145.
8. Culotta, A. Kristjansson, T. McCallum, A. and Viola, P. Corrective feedback and persistent learning for information extraction, *Artificial Intelligence* 170, (2006), 1101-1122.
9. Cypher, A. (ed.) *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, 1993.
10. Dietterich, T. Machine learning for sequential data: A review. In T. Caelli (Ed.) *Structural, Syntactic, and Statistical Pattern Recognition; Lecture Notes in Computer Science 2396* (2002), 15-30.
11. Dönmez, P., Rosé, C., Stegmann, K., Weinberger, A. and Fischer, F. Supporting CSCL with automatic corpus

analysis technology. *Proc. CSCL*, ACM (2005), 125-134.

12. Fails, J. A. and Olsen, D. R. Interactive machine learning. *Proc. IUI*, ACM, (2003), 39-45.
13. Fogarty, J., Tan, D., Kapoor, A., and Winder, S. CueFlik: Interactive concept learning in image search. *Proc CHI*, ACM, (2008), 29-38.
14. Hamming, R.W. Error detecting and error correcting codes. *Bell System Technical Journal* 29.2 (1950), 147-160.
15. Huang, Y. and Mitchell, T. M. Text clustering with extended user feedback. *Proc. SIGIR*, (2006), 413-420.
16. Kulesza, T., Wong, W.-K., Stumpf, S., Perona, S., White, S., Burnett, M., Oberst, I. and Ko, A. Fixing the program my computer learned: Barriers for end users, challenges for the machine. *Proc. IUI*, ACM (2009) 187-196.
17. Kulesza, T., Stumpf, S., Riche, Y., Burnett, M., Wong, W-K., Oberst, I., Moore, T., McIntosh, K., and Bice, F. End-user debugging of machine-learned programs: Toward principles for baring the logic (Technical Report). Oregon State University, School of EECS (2009). http://hdl.handle.net/1957/12706
18. Lafferty, J., McCallum, A., Pereira, F. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proc. ICML,* Morgan Kaufmann, (2001) 282-289.
19. Lieberman, H., (ed.) *Your Wish is My Command: Programming By Example*. 2001.
20. Liu, B. Li, X. Lee, W. and Yu, P. Text classification by labeling words. *Proc. AAAI* (2004).
21. McCarthy, K., Reilly, J., McGinty, L. and Smyth, B. Experiments in dynamic critiquing. *Proc. IUI*, ACM (2005), 175-182.
22. Rabiner, L. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE* 77(2) (1989), 257-286.
23. Raghavan, H. and Allan, J. An interactive algorithm for asking and incorporating feature feedback into support vector machines. *Proc SIGIR*, ACM, (2007), 79-86.
24. Roth, D. and Small, K. Interactive feature space construction using semantic information. *Proc. CoNLL*, (2009), 66-74.
25. Shilman, M., Tan, D., and Simard, P. CueTIP: A mixed-initiative interface for correcting handwriting errors. *Proc UIST*, ACM, (2006), 323-332.
26. Stumpf, S., Rajaram, V., Li, L., Burnett, M., Dietterich, T., Sullivan, E., Drummond, R. and Her-locker, J. Toward harnessing user feedback for machine learning. *Proc. IUI*, ACM (2007), 82-91.
27. Stumpf, S., Sullivan, E., Fitzhenry, E., Oberst, I., Wong, W.-K. and Burnett, M. Integrating rich user feedback into intelligent user interfaces. *Proc. IUI*, ACM (2008), 50-59.
28. Stumpf, S. Rajaram, V., Li, L., Wong, W.-K., Burnett, M., Dietterich, T., Sullivan, E. and Herlocker J. Interacting meaningfully with machine learning systems: Three experiments. *Int. J. Human-Computer Studies*, 67,8, (2009), 639-662.
29. Tague, J. The pragmatics of information retrieval experimentation. *Information Retrieval Experiments*, (1981) 59-102.
30. Ware, M., Frank, E., Holmes, G., Hall, M., Witten, I. H. Interactive machine learning: Letting users build classifiers. *Int. J. Human-Computer Studies*, 55, (2001), 281-292.