Lifetracker

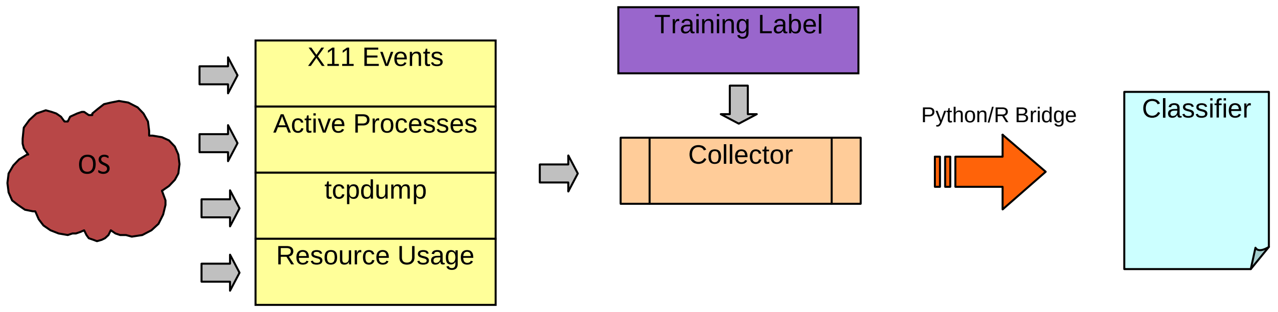
Carlin Eng, Tim Su, Vala Dormiani
CS 229, Professor Andrew Ng
Autumn 2009

Introduction

In the Internet age, worker productivity is plagued by the myriad of distractions available to anyone with a computer and an Internet connection. For those amongst us obsessed with maintaining reasonable levels of productivity, it might be useful to have a program running in your computer's background to monitor system usage and prod you whenever it detects that you might be slacking. For our project, the Lifetracker application, we attempted to create a framework to collect data on computer usage behavior and classify that behavior into one of several categories.

Data Collection Framework

Lifetracker relies on a modular data collection framework written in Python to collect data about user activity in a Linux based operating system. The following diagram explains the architecture of this system:



Data modules collect information from the operating system. The Collector component aggregates all data, solicits a training label from the user (if provided), and feeds it to our machine learning algorithms written in R through an interface library. Each data module subclasses the Recorder class, built atop Python's threading library:

```
class Recorder(threading.Thread):
        """Generic class for collecting and returning data. To create a
        Recorder, create a class that subclasses Recorder and override the
        methods below"""
        def __init__(self):
        threading.Thread.__init__(self)
        self.finished = threading.Event()
        self.daemon = True
        def collectAndClear(self):
        """Method to return collected data and clear internal state. This
        method is called by the master recording thread to collect data from
        individual recorders.
        Data should be returned as a dictionary of feature name / value
        pairs. A feature value can be either a boolean, a single number, or
        a list of strings."""
        raise Exception('unimplemented method')
        def run(self):
        """Start collecting data"""
        raise Exception('unimplemented method')
```

```
def stop(self):
"""Main thread wants to stop collection"""
raise Exception('unimplemented method')
```

Example Data

Following are examples of data collected by the framework, attached to illustrate the types of data collected and give the reader an intuition about how different features are related to user activity:

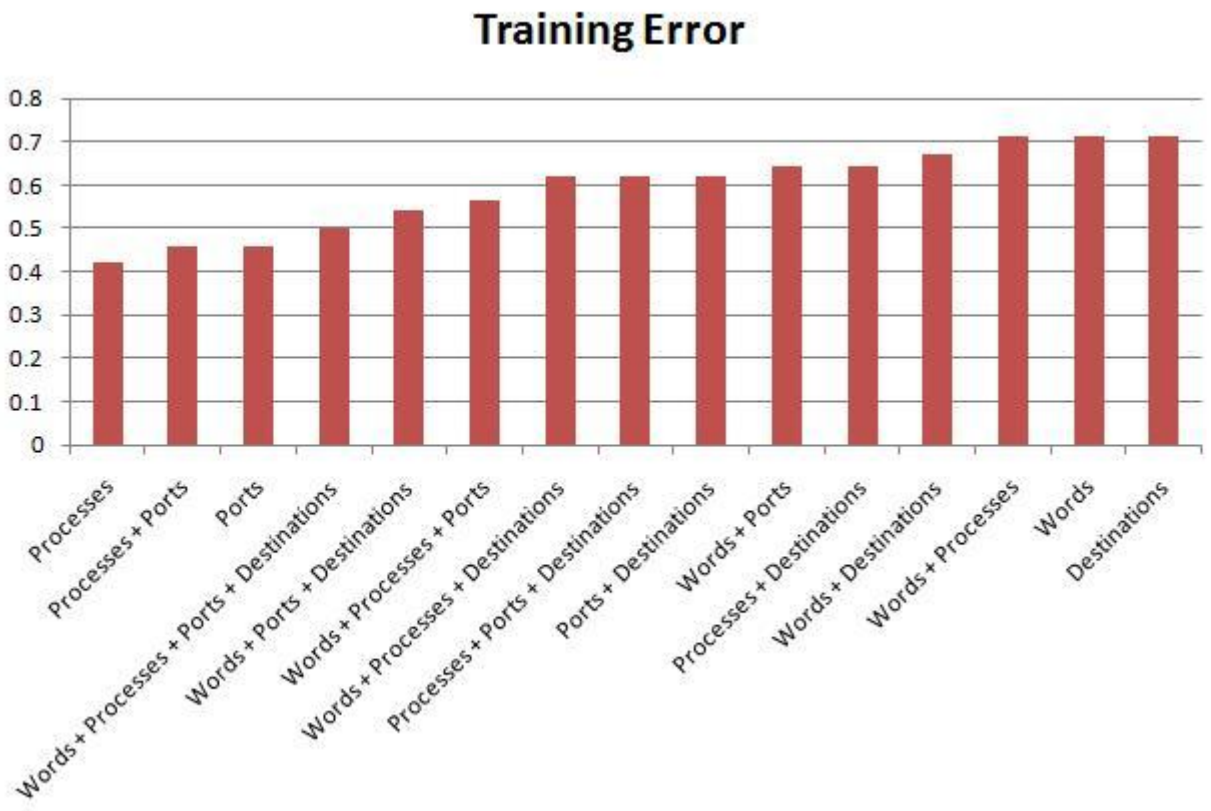
Label	programming	e-mail	internet	gaming
Words Typed	ac, cl, xb, cd, coll, exit, sleep, xs, provided, no, py, start, kill, xwcls, ls, answer, import, run, stop	and, right, just, house, is, hard, am, idea, at, yes, next, really, are, little, thank, things	delic, digg, popcorn, mmmm, com	rroo, zzz, zzx, zzzz, zz
Packet Ports	www: 98, imaps: 48, domain: 10	domain: 16, imaps: 31, www: 353	https: 104, www: 2203, domain: 126	https: 35, www: 7, imaps: 28, domain: 3
Packet Destinations	stanford.edu, mozilla.com, google.com	google.com, stanford.edu, moody.edu	google.com, facebook.com, akamaitechnologies.com	google.com, 60.201.44.122, 74.125.95.109
Mouse Travel Pixels	7935	21361	25735	2796
Mouse Clicks	2	36	32	0
Keystrokes	935	586	107	1555
Transmitted Bytes	18 kB	45 kB	5.7 kB	14 kB
Received Bytes	111.0 kB	442 kB	3.9 MB	67 kB
Disk Write Bytes	21.0 MB	86.2 MB	10.8 MB	3.2 MB
Disk Read Bytes	159 MB	39.4 MB	3.2 MB	9.5 MB
User CPU %	8.55%	8.01%	36.60%	15.27%
Kernel CPU %	5.08%	2.39%	4.85%	1.66%

Naïve Bayes Classifiers

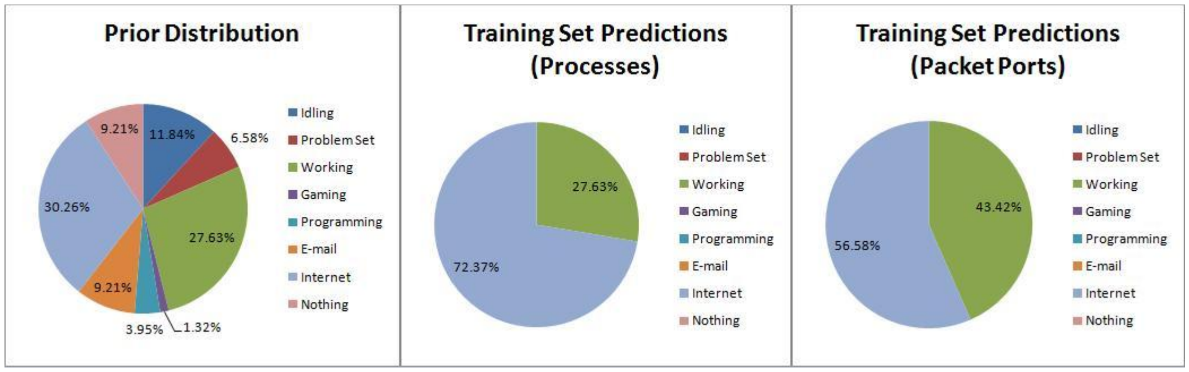
We implemented several Naïve Bayes classifiers with the goal of using specific events as predictors for user behavior. Our data collection scripts output Python dictionaries with histograms of four types of events: words typed, outgoing TCP packet destinations, outgoing TCP packet port numbers, and processes running. The idea is simple: certain events are more likely to occur when a user is performing certain tasks, so these events should have some power in helping us predict current activity. For example, someone in the act of programming in Python is more likely to type the word “import” than someone watching online videos. Similarly, an open web browser is more indicative of Internet activity than a closed web browser.

Our implementation reads the dictionary outputs of the data collection scripts and generates a prior distribution over all potential activities based on their relative frequencies. Then, using a multinomial event model, we attempted to calculate the likelihood of each of the activities based on the dictionary contents.

The events come in four different flavors, so a Naïve Bayes classifier can be constructed using every combination of one or more of these dictionaries. Our team tried implementing classifiers using all 15 possible combinations. The results were underwhelming. The best training error we achieved was 40%, with the majority of classifiers guessing incorrectly over 50% of the time:



The worst classifiers, using only dictionaries of words typed, or packet destinations, had astounding training errors of over 70%, correctly classifying only 30% of the time, which is only slightly better than the performance we could expect from someone randomly guessing (1/8 = 12.5%). These astronomical training errors indicate our model suffers from an extremely high bias, and fails to capture the story correctly. Looking at the distribution of guesses from training set predictions reveals part of the reason why the errors are so high:

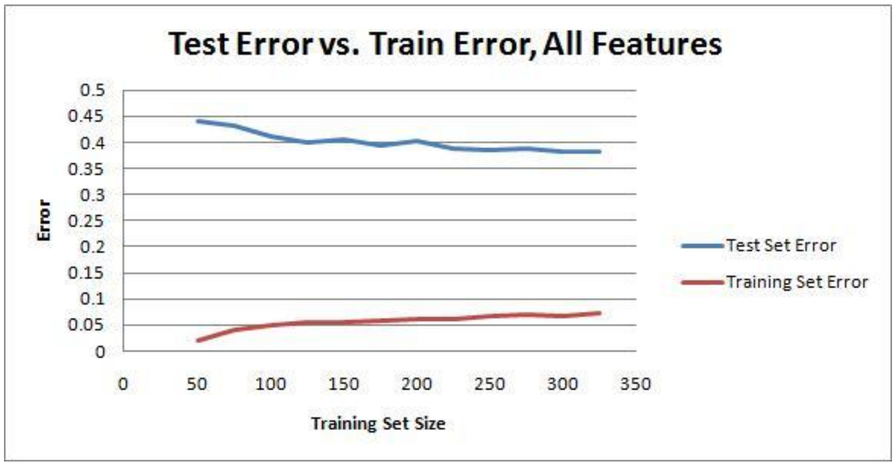


These charts show the distribution of guesses on our training set for the processes and the ports classifiers, as well as our distribution of prior beliefs. Using these naïve Bayes classifiers, every single one of our points is either assigned “Internet”, “Working” or “Nothing”. Looking at the prior distribution, we see that “Internet” and “Working” are by far the most likely, accounting for 30.26% and 27.63% of all observations respectively. With so little data, we were not able to consistently observe specific events associated with specific behavior, so when calculating the likelihood for each of the categories, these large prior values dominated the likelihood observed from the actual events. The events contributed almost nothing to the likelihood, resulting in the majority of our observations picking up the label of “Internet” or “Working”. These results suggest that the Naïve Bayes approach to solving this problem performs poorly when given small samples of data.

Logistic Regression

Quantitative data such as number of keystrokes, mouse movement, bytes read/written to disk, network activity and CPU usage can all serve as predictors for user behavior. Users who are idle, playing games, coding, or watching videos are all likely to use varying degrees of system resources. We used a multinomial logistic regression to classify user behavior into one of eight categories based on the following features collected over five minute intervals: total mouse distance moved, total number of mouse clicks, total number of keystrokes, RAM used, bytes transferred, bytes received, bytes written to disk, bytes read from disk, last one/five minutes of system load, kernel CPU%, user CPU%, and IO Wait CPU%.

We implemented our multinomial logit model in R using the VGAM package developed by Yee and Wild (1996). Each of our feature vectors contains 13 predictors, but it is very likely that several of the features are either correlated with each other, or simply do not act as good predictors for user behavior. In fact, plotting the test error and training error vs. training set size shows that when our model includes all 13 predictors, our test error remains very high, despite very low training error.



This is a classical indication that our model suffers from high variance and we are overfitting. Choosing an appropriate model is an essential task, and there are plenty of methods available for model selection, several of which we explored.

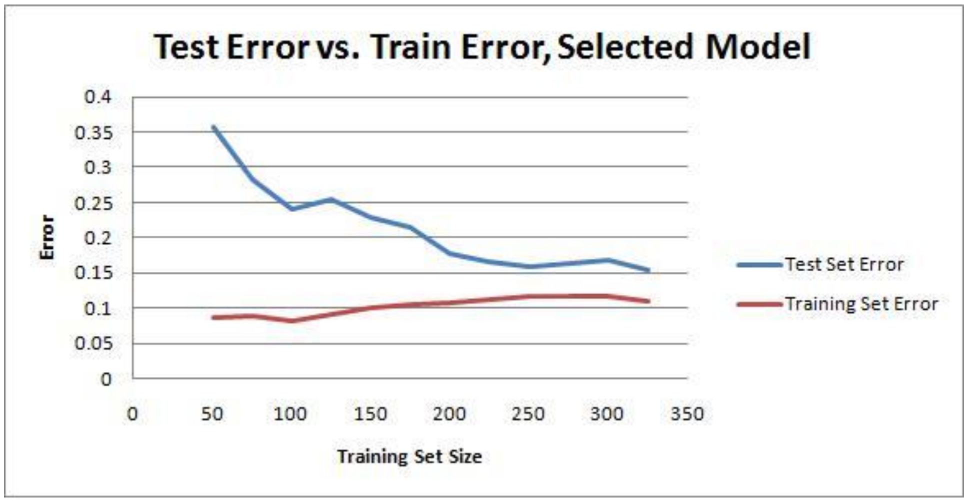
Model Selection

Exhaustive Search:
An exhaustive model search fits all possible models and compares their performances using a technique such as leave one out cross-validation, selecting the model with the best results. Each of our observations consists of 13 features, so an exhaustive search would require fitting 2^{13} = 8192 models and cross-validating each one. This is an extremely expensive operation, and as our training set size increases, as it inevitably will for our application, the procedure becomes computationally infeasible.

Stepwise Methods:

Akaike's Information Criterion (AIC) calculates the log-likelihood of the estimated model and penalize the model for each additional parameter: AIC = -2 * loglik + 2k, where k is the number of model parameters. A model with lower AIC is superior. We utilized a two-way stepwise search, starting with all 13 parameters. At each step, we consider either dropping or adding a parameter, making our decision based on which model has the lowest AIC. When we reach a point where a better model cannot be attained by either dropping or adding a parameter, our model selection is complete. Results can be further tested using cross validation.

Our final selected model contained the following predictors: mouse distance moved, number of keystrokes, bytes read from disk, bytes written to disk, used memory and bytes received. The following chart plots our errors versus training set size, showing the new model is a significant improvement over the previous approach:



Conclusions

Our attempts to classify behavior using a Naïve Bayes algorithm performed poorly. The set of total dictionary items such as words typed was simply too large for our small sample size to effectively associate events to user behavior. Perhaps if we extended our sample size to include many more data points across many more users, our algorithm's effectiveness would improve. Our multinomial logistic regression model performed reasonably well, but initially suffered from overfitting. Using a stepwise model selection process, we were able to reduce the variance of our model by a significant margin by including only the most important features in our model. In the future, we could also utilize regularization techniques such as L1 or L2 penalties, combined with a prior distribution over our parameters. The logistic regression model, with its ease of implementation, and satisfactory results, will most likely be the algorithmic method of choice moving forward. Also, our current data collection framework works only in a Linux environment. In the future, we hope to create a platform agnostic version, suitable for use with more common operating systems such as Windows.

References

Ng, Andrew. CS 229 Autumn 2009 Lecture notes. Available at: <http://www.stanford.edu/class/cs229/materials.html>.

Sakamoto, M. Ishiguro, G. Kitagawa. "Akaike Information Criterion Statistics." *Journal of the Royal Statistical Society. Series D (The Statistician)*, Vol. 37, No. 4/5 (1988), pp. 477-478.