Using Machine Learning to Predict Bug Outcomes from Automation

Wayne Roseberry

wayner@microsoft.com

Abstract

Sure, automation is great because it can give you a whole bunch of information really fast. And yeah, it is a big pain because it gives you a whole bunch of information really fast. Sometimes the amount is too much to process, but you know you need to look because there might just be an important failure in the pile that has to be fixed.

What if it was possible to predict your decision before you read it? How could that save time? How could that help your team focus on important work first rather than dig through a huge pile of data.

This paper will talk about experiments the author has executed in the Microsoft Office team using machine learning to predict outcomes for failures found by massive automation suites. It will talk about the methodology, the problems involved with evaluating such results, the potential discovered in the experiments and some of the inherent challenges, difficulties and risks in trusting such decisions to a machine.

Biography

Wayne Roseberry is a Principal Software Engineer at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, and all versions of SharePoint. Currently, Wayne works in the Office Engineering team focusing most of his time on test automation and automated testing systems. Previous to testing, Wayne also worked in Microsoft Product Support Services, assisting customers of Microsoft Office.

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne writes, illustrates and self-publishes children's literature.

1 Introduction

Large scale automation efforts offer a lot of value by running lots of tests, freeing people and engineers to perform other tasks. But the costs of triaging and processing those failures can exceed the capacities of teams to keep up with their incoming failures, assign to personnel to fix, make a decision on how to process them.

This paper describes an experiment run by the author exploring the potential of using machine learning algorithms to aid in the triage of bugs. It asks how well a computer would do predicting the decision that a human would make encountering the same failure, how difficult would it be to construct the models and tools to perform this task. It also asks what we can do with the results, whether we should trust a machine to make decisions about quality for us. Will the machine accelerate the worst part of our own decision making? Or perhaps we can use the information to improve the decisions we do make, executing them faster or more effectively.

This paper is targeted at readers and an audience with an intermediate to expert level familiar with software testing and test automation, but that may be new to machine learning concepts and methodologies. Some introductory material sufficient to explain the methodology and approach and results will be covered, although it intentionally avoids deeper issues of machine model tuning, increasing accuracy and comparison between different models and approaches.

The results show some real potential in using machine learning to aid the failure processing and triage process. The technique described in this document did not prove to be too expensive or difficult, and most certainly creating models based on existing test automation failure triage histories is well within the capabilities of available technology. The experiment never left the laboratory (that laboratory literally being the PC under the desk in the author’s office), but certainly deployment to a production system is feasible. As will be described later in the document, there are certainly limits at this point to applying the results blindly, so applications are probably best selected that work in conjunction with human decision making rather than replacing it. This results in this document raise questions more than answer them, which is this author’s intent.

2 Motivating Problem – Triaging with Large Volumes of Automation Failures

2.1 Test Automation in the Microsoft Office team

2.1.1 Office runs a lot of automation, with a lot of failures.

The Office automation system is used actively by approximately 2600 software engineers[1], all working on an integrated product with large numbers of dependencies. Engineers run automated tests that deploy to a lab of several thousand machines which install and configure the product before launching what is typically an end to end integrated test. These tests are launched either to validate changes before submitting to the source repository, or to look for bugs that may have accumulated in large blocks of changes. Individual product teams run constant looping builds of their own on team branches off the main source fork while daily builds of the entire main line Office branch also initiate several automation suite runs.

[1] Based on a query 7/19/2015 which identified 2619 distinct users launching automation jobs on this system over the prior 90 days. On a daily basis, the system will be used by as much as 700 users per day. Engineers in this case are all software developers, as the Office organization does not have a distinct job title for testers.

On a daily basis, the Office automation system typically runs over 2 million tests per day. The number of failures observed by the system on a daily basis tends to be about 130,000 – 150,000, which are distributed to product teams to triage for resolution (fix, not fix, etc.)

2.1.2 Failures are automatically classified as new or unique

In response to a large number of failures, many of which were symptomatically the same problem, Office introduced a failure bucketing mechanism that attempts to determine if a failure was seen previously (Robinson). The technology affords a great deal of time savings on the part of engineers and product teams as they no longer need to determine if a specific instance of a failure is new or not.

This document does not detail the failure matching mechanism beyond the following points: 1) it matches based on symptoms of the failure as root cause is unknown at the time failure, 2) the matching is based on a textual similarity metric inside the failure message emitted by the testing 3) the matching considers details of the call stack collected at time of failure.

Once a failure is identifies as new, it is entered automatically into the Office bug tracking database.

The auto-bucketing system and the eventual resolution of the bug entered are important inputs into the machine learning experiment that follows.

2.1.3 Triaging failures is expensive and error prone

Once bugs are in the bug database, product teams resolve them with one of the following states, "FIXED", "WON'T FIX", "BY DESIGN", "POSTPONED", "DUPLICATE", "NOT REPRO" and "EXTERNAL." The general disposition of this paper is that any decision other than "FIXED" represents energy and cost spent toward a desirable outcome, while all other costs are inefficiencies of the system. We would have preferred spending the cost on something driving to a fix.

Product teams spend a considerable amount of time and energy making decisions about bugs. As a bug is created by a test, it is often the case that the output of the failure message is not something immediately obvious to a person reading it without having to read the test code itself. And even inside the test code, the actual failure may be hard to discern. For example, a test might have been "Insert text, open Format dialog, applying formatting, dismiss, check if formatting matches expectation" and the failure might have been "The Format dialog never displayed" Any number of underlying causes, from legitimate product failures to issues in the test harness code, could cause such a problem. Investigation is difficult and expensive.

Even the seemingly simple act of eliminating duplicate failures (current rate of duplication in the system is ~40%) takes time. An anecdotal survey by this author of Office engineers reported an average of 20 minutes per bug to establish a credible case for a failure being a duplicate.

2.2 Questions of interest

2.2.1 How accurately could a computer model predict the final resolution of a failure?

The first question was whether or not a trained model could save time on bug triages. Would it be able to predict, accurately, the same resolution that humans would? Test automation failures messages tend to repeat the same text patterns, so I thought it was possible it might. But it was also far from certain, as failure messages are purely symptoms, and often underlying issues only seen after exploring the code in-depth drive the final decision.

But there is certainly an opportunity to save some sort of cost or energy if a prediction was possible. Even just hinting to a triage team what the model predicts could guide them in focusing their attention in specific directions more efficiently than without. Of course, it was also just as likely to waste time by pointing the triage team in the wrong direction.

2.2.2 How well could a computer model explain why it chose a given resolution?

Triage teams and engineers need justification for their decisions. Maybe a given failure represents a false assumption in test code, and does not justify a product fix. Maybe a given failure only occurs on conditions the test creates and deserves a lower priority. Maybe a given failure, while rare under test conditions, represents a severe failure state that would cause a customer great harm.

Likewise, we would want some sort of explanation from a computer model. This can be a difficult requirement, as many machine learning models get complex very quickly and determining why the model offered a specific solution or answer is sometimes unwieldy if not impossible. Further, the reasons for the decision are based on nothing more than the inputs to the system. If all a model looks at is occurrence rate of certain words in a failure message, then there is no direct relationship to other factors that might drive a team to make a decision, such as likelihood of occurrence in market, or severity of failure, or likelihood that the failure is a test artifact only. At best there is correlation of certain inputs with certain outcomes.

2.2.3 Would automatic prediction present any dangers or challenges we need to worry about?

Numerical analysis and machine learning are very good at allowing people to believe things that are not true. The reason why ML has to date been mostly within the realm of data science is because there is a need for disciplined statistical analysis to avoid drawing erroneous, and perhaps dangerous, conclusions. Consider:

Blindly trust an inaccurate model to resolve bugs: Imagine a model that achieves an 80% accuracy rate on predicting a bug would be something other than “FIXED”. This would mean that for every 100 bugs punted, 20 of them are bugs the team would have fixed. An 80% accuracy rate is actually a rather impressive feat, but for something like whether or not to fix a given bug, that 20% miss is likely much higher than a team would tolerate.

Reinforce bad habits: In the model described in this document, the training data was based on historical resolution trends. The model was being taught to predict team behavior, not necessarily best behavior. Suppose the team was making bad decisions by punting bugs that were important to the customer. The model could actually make such a phenomenon worse by deferring decision until a later point.

3 Classification via Machine Learning, Background

At this point, we will discuss a bit about machine learning and how it works for problems such as the one described here.

3.1 This problem is a supervised classification problem

A supervised classification problem means two things in machine learning terms:

1. “Supervised” means the model is trained by showing it example data with “correct” answers. In this case, the example data comes from test failure message text, the sequence of methods executing in the source code at the time of failure (call stack), and the answers to the data comes in the form of bug resolutions as chosen by product teams and engineers.

2. “Classification” means the goal of the model is to determine which category a given piece of example data fits into. In this experiment, the classifications were the actual bug resolutions assigned by product teams to bugs. The possible values are: “FIXED”, “POSTPONED”, “WON’T FIX”, “BY DESIGN”, “DUPLICATE” and “EXTERNAL”. A resolution, in this case, is the final decision about what to do with a given bug. In the case of “FIXED,” the resolution is assigned after an engineer has written, tested and submitted the code change. In all other cases, the resolution is typically assigned at the moment of triage.

3.2 Multi-class logistic regression partially explained

In this discussion, the word “term” refers to strings extracted from the failure message and call stack text in the training data.

It is almost sufficient to say multi-class logistic regression is *“…a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes*” (Wikipedia, “Multinomial Logistic Regression”) and leave it at that. There are many different types of machine learning models that are capable of multiple classification, and they all have their various strengths and weakness. They almost wind up being treated like a black box, and the engineer/scientist training the model will typically try different types of classification models with their data and just pick the one, or combination of models, that consistently yields the highest accuracy. However, there are a few details worth mentioning because it is relevant later to the discussion:

1. Multi-class logistic regression is a very simple and fast classification technique

2. The heart of most logistic regression problems is a math formula that looks like the one below. In this case, “X_N” represents the frequency of some term in the training data and “Θ_N” represents a coefficient in the model that is multiplied by that frequency:
   $\Theta_0 X_0 + \Theta_1 X_1 + \ldots \Theta_{N-1} X_{N-1} + \Theta_N X_N$

So, the variable X_0 might be for the frequency of the term “window” and the variable X_1 might be for the frequency of the term “crash” and so on. The model is trained by looking at thousands of examples of data, and iterating until it finds a set of values for “Θ_N” that most optimally produce the same answer given for the data. A more thorough and very accessible explanation of the exact algorithm behind logistic regression is available from Coursera (Ng, Andrew).

The reason this is interesting is because it makes the reasons behind the decision the model makes very apparent. If a given term, such as “menu” is highly correlated with the resolution “FIXED” then it will have a “Θ_N” value associated with it that is very high. Likewise, if a given term, such as “timeout” is negatively correlated with the resolution “FIXED” then it will have a “Θ_N” value that is very low. We will refer back to this point later.

3.3 Understanding Classification Results,

3.3.1 Recall versus Precision

There are several metrics used to judge the accuracy of classification problems, but the two simplest and most commonly used are precision and recall.

Precision is the percentage of times that a prediction selected the correct answer. So, if a trained model were tested and said that 100 items were resolved “FIXED” and in fact only 99 of them were resolved “FIXED” then the model is said to have a 99% precision rate on the category “FIXED”.

Recall is the percentage of total items in a given category that were correctly identified by the trained model. For example, if in the training data there were 200 items whose correct classification was “FIXED” and the trained model only classified 100 of them as “FIXED” then that model is said to have a 50% recall rate on the category “FIXED”.

3.3.2 Dangers of over-fitting

Over-fitting is a classic mistake in machine learning. “*In statistics and machine learning, overfitting occurs when a statistical model describes random error or noise instead of the underlying relationship.*” (Wikipedia, “Overfitting”). It typically happens when there are so many variables in a given model that it is capable of extremely high precision and recall rates on its training data, but on nothing else. It can easily happen when the training examples do not generalize well or are too small relative to the large real-world examples. It also happens when the test data used to measure the model accuracy is really just a repeat of the data used to train the model instead of a suitable representative sample of the real world.

Over-fitting is relevant to the discussion partly because it is always relevant to machine learning. It is also relevant because as the results will show later, the experiment yielded surprisingly high accuracy. It became important to examine whether or not the sin of over-fitting had been committed.

4 Methodology

4.1 Toolset

For the experiment, I used a set of tools and library of machine learning APIs created and used at Microsoft. The tools are not available externally, but the code in the libraries are the same code that is available via Microsoft Azure Machine Learning (Microsoft, "Multiclass Logistic Regression"). The rest of this document is not going to focus a great deal on specifics of the toolset.

4.2 Automation result collection and data preparation

Most of the effort in this experiment involved preparing the data for consumption. This is common in a lot of machine learning examples, as machine learning models are typically generalized learning systems, and the source data is usually specialized to whatever problem it relates to. The data needs to be converted from its specific, original form, to a generalized form for the model.

In this case, the data needed to be converted from something that looked similar to the following:

> *"Assert 'hy5k' occurred in WINWORDD.EXE. Message: This dll has failed to load…"*

Into the following format:

> *Assert   WINWORDD.EXE   failed_to_load*

This required the following steps

1. Running a query from the automation system database to get instances of test failure, the failure message text and the final bug resolution. This resulted in hundreds of thousands of rows of data.

2. Breaking the failure message text into single terms to serve as training "features" (the ML term used to indicate variables used to train a model)

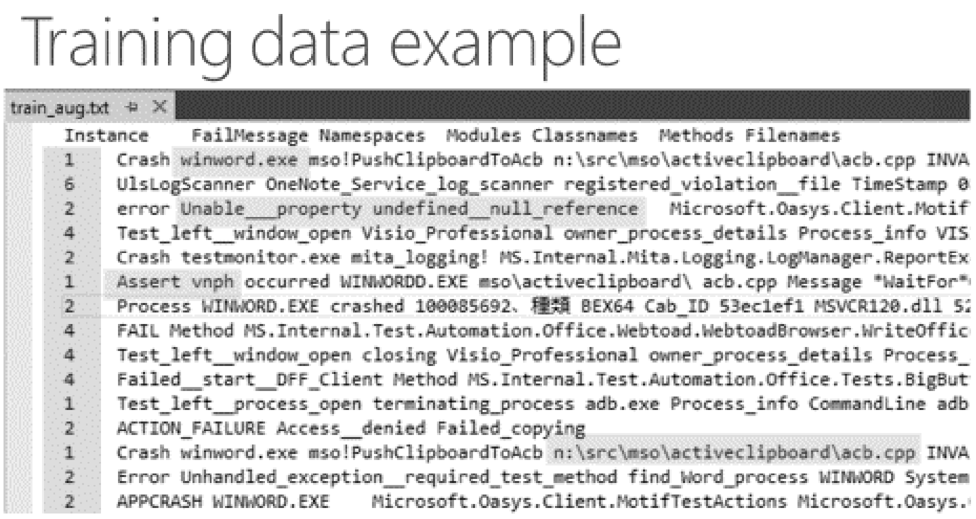
    a. Split on common separators (comma, semicolon, etc.)

    b. Extract everything that looks like a filename or assembly name into a term (e.g. "WINWORD.EXE" or "Microsoft.Internal.Office.Automation.UI.dll")

    c. Identify special terms that are interesting, e.g. "Assert" or "Crash"

    d. Remove anything that looks like noise (short words, dates, numbers, GUIDS, random strings used as filler test data)

    e. Concatenate remaining multiple words together into single terms ("Application Foo closed the Window" -> "Application_Foo_Closed_Window"). This was to reduce the number of variables training in the model, mostly for speed reasons.


Training data example

```
Instance    FailMessage Namespaces  Modules Classnames  Methods Filenames
1     Crash winword.exe mso!PushClipboardToAcb n:\src\mso\activeclipboard\acb.cpp INVA
6     UlsLogScanner OneNote_Service_log_scanner registered_violation__file TimeStamp 0
2     error Unable___property undefined__null_reference   Microsoft.Oasys.Client.Motif
4     Test_left__window_open Visio_Professional owner_process_details Process_info VIS
2     Crash testmonitor.exe mita_logging! MS.Internal.Mita.Logging.LogManager.ReportEx
1     Assert vnph occurred WINWORDD.EXE mso\activeclipboard\ acb.cpp Message "WaitFor"
2     Process WINWORD.EXE crashed 100085692、種類 BEX64 Cab_ID 53ec1ef1 MSVCR120.dll 5
4     FAIL Method MS.Internal.Test.Automation.Office.Webtoad.WebtoadBrowser.WriteOffic
4     Test_left__window_open closing Visio_Professional owner_process_details Process_
4     Failed__start__DFF_Client Method MS.Internal.Test.Automation.Office.Tests.BigBut
1     Test_left__process_open terminating_process adb.exe Process_info CommandLine adb
2     ACTION_FAILURE Access__denied Failed_copying
1     Crash winword.exe mso!PushClipboardToAcb n:\src\mso\activeclipboard\acb.cpp INVA
2     Error Unhandled_exception__required_test_method find_Word_process WINWORD System
2     APPCRASH WINWORD.EXE    Microsoft.Oasys.Client.MotifTestActions Microsoft.Oasys.
```

4.3 Result collection and comparison

The training data was collected from 90 days of automation data (selected because the automation system purges data older than 90 days), extracting only those results with resolved bugs. The bug resolutions were the real, actual decisions of product teams. 20% of the data was removed from the training set and used for model testing.

Product teams take a while to resolve automation bugs, on average 15 days (there is a division wide goal pushing to this). This means that toward the end of the training data sample there are fewer bugs in a resolved state to train the model. I was curious if this would affect the accuracy of the model with brand new failures. To measure this, an additional 15 days of data after the training data were evaluated and predictions recorded and compared against actual resolutions.

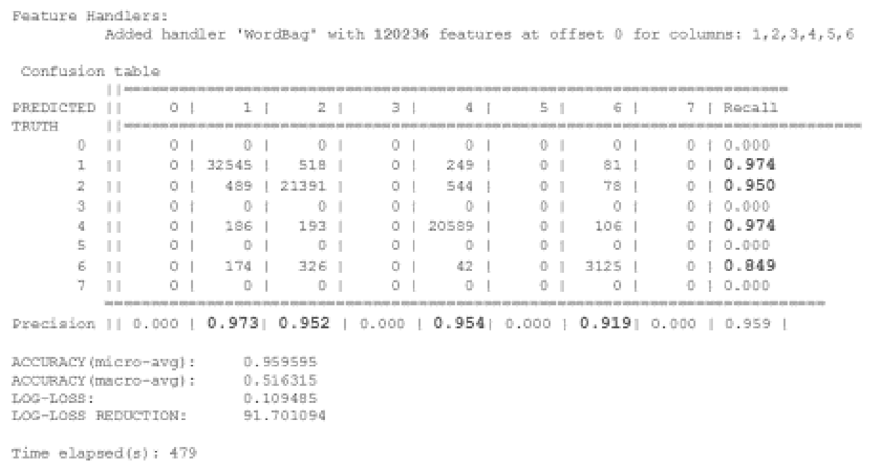
5 Results

5.1 Initial accuracy numbers

The initial results were surprising. Anecdotal accounts from other efforts by teams within the company attempting similar problems (automatically assigning bugs to appropriate owners based on the text on human created bug reports) were that achieving accuracy as high as 70% required considerable effort. The results of this experiment were quite different.

For the categories, “FIXED”, “DUPLICATE”, “WON’T FIX” and “NOT REPRO”, the model made a correct prediction (precision) 97%, 95%, 95% and 91% of the time, respectively. On the same categories, the model correctly identified 97%, 95%, 97%, 84% of the data in the category (recall). The other categories (“BY DESIGN”, “POSTPONED”, “EXTERNAL”) were so rarely used by the product teams that the model generated no predictions.

A diagram of the results is shown below. The results are shown in the form of a confusion matrix, which is a common format used in predictive analytics to compare recall against accuracy for a given population of data. The table below can be read to determine recall for a given classification by looking at the final column in the table and going to the row number for that classification. For example, “FIXED” is classification 1, and it has a recall value of 97.4%. Precision can be looked up by going to the final row of the table and looking at the number for the column of that classification. For example, “FIXED” (1) has a precision value of 97.3%. Prediction labels are shown on the top row of the table and the actual value are shown on the leftmost column of the table. Individual cells indicate how many predictions for a given classification hit how many items of a given, real classification. Where the column and row labels match, the model predicted correctly (e.g. 1:1 had 32454 hits for “FIXED”). Where the column and row labels do not match the model predicted incorrectly (e.g. 4:1 had 186 hits where the model predicted “FIXED” but the actual resolution was “WON’T FIX”).



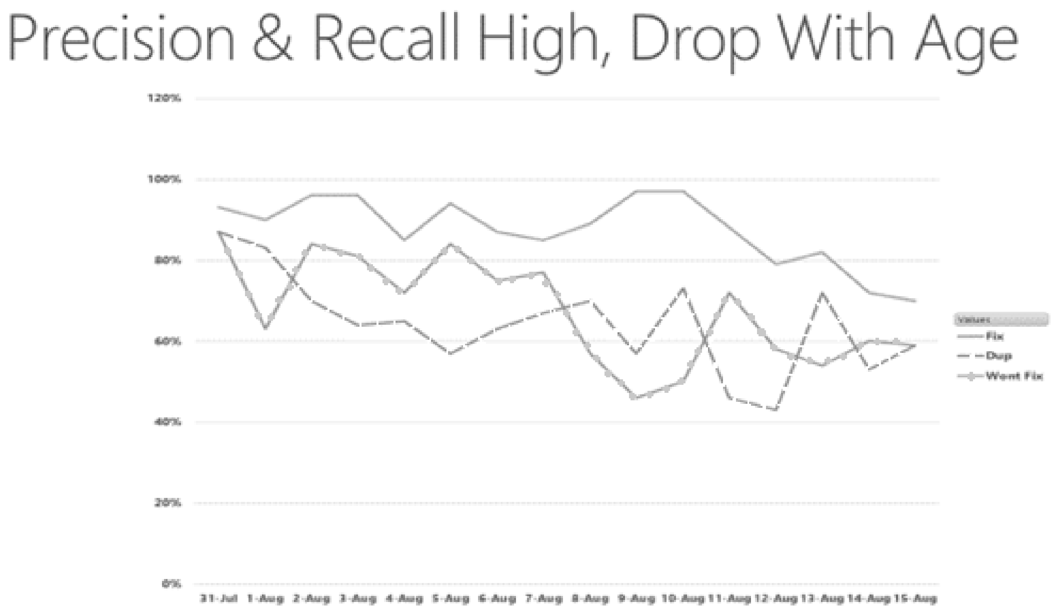
```
Feature Handlers:
         Added handler 'WordBag' with 120236 features at offset 0 for columns: 1,2,3,4,5,6

 Confusion table
          ||================================================================================
PREDICTED ||      0 |      1 |      2 |      3 |      4 |      5 |      6 |      7 | Recall
TRUTH     ||=========================================================================================
        0 ||      0 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000
        1 ||      0 |  32545 |    518 |      0 |    249 |      0 |     81 |      0 | 0.974
        2 ||      0 |    489 |  21391 |      0 |    544 |      0 |     78 |      0 | 0.950
        3 ||      0 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000
        4 ||      0 |    186 |    193 |      0 |  20589 |      0 |    106 |      0 | 0.974
        5 ||      0 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000
        6 ||      0 |    174 |    326 |      0 |     42 |      0 |   3125 |      0 | 0.849
        7 ||      0 |      0 |      0 |      0 |      0 |      0 |      0 |      0 | 0.000
          ===================================================================================
Precision || 0.000 | 0.973| 0.952 | 0.000 | 0.954| 0.000 | 0.919| 0.000 | 0.959 |

ACCURACY(micro-avg):      0.959595
ACCURACY(macro-avg):      0.516315
LOG-LOSS:                 0.109485
LOG-LOSS REDUCTION:       91.701094

Time elapsed(s): 479
```

1=Fixed
2=Duplicate
3=By Design
4=Won’t Fix
5=Postponed
6=Not Repro
7= External

5.2 Accuracy changes over time

As stated previously, our bug resolution data tends to take 15 days or more to resolve, so I wanted to see how accurately a model built up to 15 days prior to a failure occurrence could predict the resolution of the bug from the failure. Would the accuracy hold up, or would it degrade with age? The results were clearly the latter.

Close to the training window, overall results showed high average of the measures for precision and recall, at approximately 90%, depending on which resolution was predicted for. The results dropped as the model aged relative to the failure occurrence. The daily results varied a greatly, but the overall slope was strong, show the predictions dropped in overall accuracy by > 20% points by the 15 day mark at the far end of the graph. This suggests that the accuracy of the model correlated to how close in time the model was trained relative to the failure itself. The following chart shows the drop rate over time from the last day of training data on the far left, and the last bug predicted against on the far right.

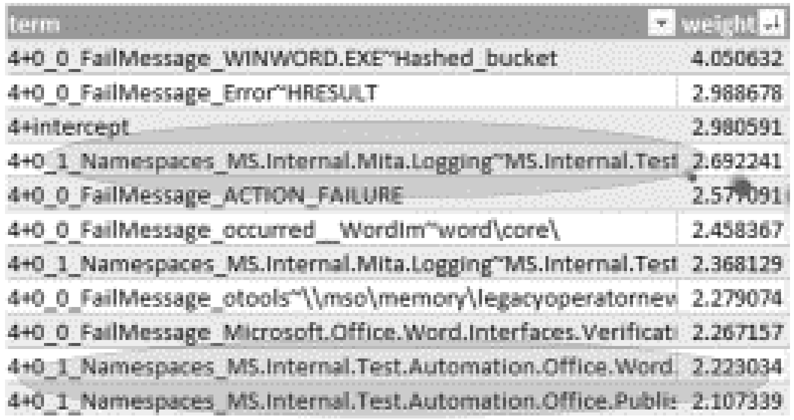


This drop in accuracy makes sense intuitively, as the new failures seen by the automation system, coming largely from changes to the product code base or to the test automation system, would not be reflected in the data in a much older set of training data. As you continue testing, you're exercising code that the training model hasn't seen yet and didn't train on. This had implications for the value of the model for dealing with brand new failures. The model accuracy was as much as 20% lower for any failure that was completely new than for a failure that was very similar to ones seen prior.

5.3 Term weightings and explaining results

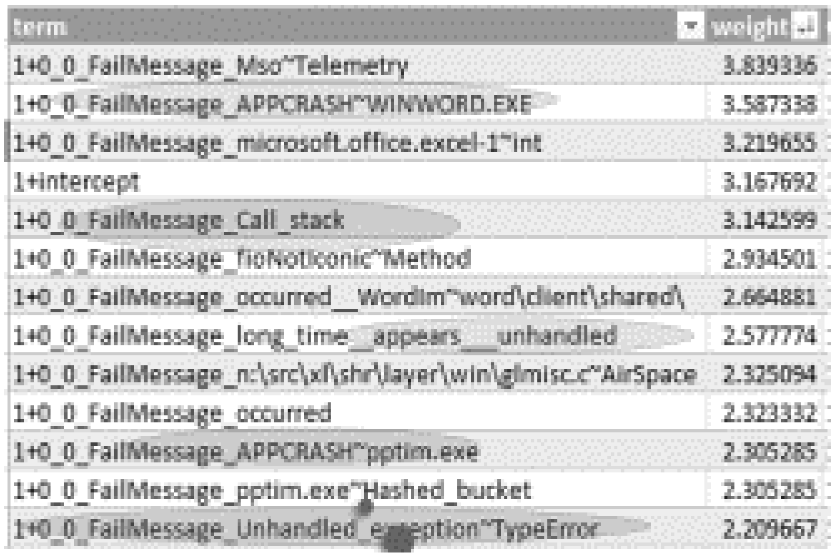
As mentioned prior, the simple underlying structure of the way logistic regression based models are built makes it somewhat easy to evaluate how much a given feature, or in this case term, contributed to a given outcome. Every single term in the model is assigned a numerical weight that will be multiplied by whatever frequency it is given inside the instance data. If the number is large, the term is considered very important to the outcome of the bug resolution. If the weight is very small, or negative, it actually pulls away from the outcome of the bug resolution.

There was a strong correlation to certain terms appearing in a failure message and the tendency to fix or not fix a bug. Terms that teams related to test infrastructure components throwing a failure tended to receive a “WON’T FIX”, particularly components associated with simulating end users controlling and manipulating UI state. The image below shows a list of terms from the experiment that were heavily weighted toward the “WON’T FIX” resolution. Two of the very interesting sub-strings in the terms that came up a lot toward the top of this data set were “MS.Internal.Test.Automation” and “MS.Internal.Mita.” Both of these are test tool and automation frameworks internal to Microsoft. It is impossible to tell from a numerical analysis of this sort the true cause of whatever failure was thrown. It may be that the presence of these strings in the failure message content push a team strongly not to fix a bug, although correlation does not always indicate causation. It is most likely the case that teams are interpreting the failure as purely a test system artifact, but that is not a point that can be made conclusively without further detailed investigation.



term	weight
4+0_0_FailMessage_WINWORD.EXE~Hashed_bucket	4.050632
4+0_0_FailMessage_Error~HRESULT	2.988678
4+intercept	2.980591
4+0_1_Namespaces_MS.Internal.Mita.Logging~MS.Internal.Test	2.692241
4+0_0_FailMessage_ACTION_FAILURE	2.577091
4+0_0_FailMessage_occurred__WordIm~word\core\	2.458367
4+0_1_Namespaces_MS.Internal.Mita.Logging~MS.Internal.Test	2.368129
4+0_0_FailMessage_otools~\\mso\memory\legacyoperatornew	2.279074
4+0_0_FailMessage_Microsoft.Office.Word.Interfaces.Verificat	2.267157
4+0_1_Namespaces_MS.Internal.Test.Automation.Office.Word	2.223034
4+0_1_Namespaces_MS.Internal.Test.Automation.Office.Publis	2.107339

Meanwhile, terms that correlated with application thrown errors, such as crashes and faults, or with better information such as call stacks inside product code, tended to receive a “FIXED” resolution. The image below shows a list of terms that were strongly correlated with a “FIXED” resolution. Not the presence of the phrase “APPCRASH” in the data. These are errors that could only have happened within the product, and by no means could have been invoked by test code. Also interesting is the phrase, “Call_stack”. In this case a product engineer would be able to determine exactly what line of code threw the failure, making diagnosing and fixing the failure much easier. Another interesting phrase is “Mso~Telemetry” occurring at the top of this list. This refers to in-product telemetry markers, which suggests that telemetry reported data inside a failure is very strongly correlated with the resulting bug receiving a “FIXED” resolution.



term	weight
1+0_0_FailMessage_Mso~Telemetry	3.839336
1+0_0_FailMessage_APPCRASH~WINWORD.EXE	3.587338
1+0_0_FailMessage_microsoft.office.excel-1~int	3.219655
1+intercept	3.167692
1+0_0_FailMessage_Call_stack	3.142599
1+0_0_FailMessage_fioNotIconic~Method	2.934501
1+0_0_FailMessage_occurred__WordIm~word\client\shared\	2.664881
1+0_0_FailMessage_long_time__appears___unhandled	2.577774
1+0_0_FailMessage_n:\src\xl\shr\layer\win\glmisc.c~AirSpace	2.325094
1+0_0_FailMessage_occurred	2.323332
1+0_0_FailMessage_APPCRASH~pptim.exe	2.305285
1+0_0_FailMessage_pptim.exe~Hashed_bucket	2.305285
1+0_0_FailMessage_Unhandled_exception~TypeError	2.209667

Such an analysis is not possible with all classes of machine learning, which makes use of simple predictive models as logistic regression particularly advantageous. There are probably several ideas to take away from this analysis, although they may be specific to practices and culture within the Office division at Microsoft. Some of those takeaways could be:

1. Failure messages that contain test harness and tool only data will tend to result in bugs that are resolved “WON’T FIX”
2. Failure messages with application specific failure content will fare better in terms of bugs resolved “FIXED”
3. Failure messages that aid in the investigation, such as providing stack trace information, fare better in terms of bugs resolved “FIXED”

6 Conclusions and Questions

6.1 Repetitive nature of the text input made the results unusually accurate

Over-fitting was an immediate concern because the numbers were so accurate. Particularly with logistic regression, it is common that with lots of variables (in this case, terms in the failure message) that the model contains a combination of coefficients capable of predicting exactly the result from the training data and no other. This concern was countered by the testing data. The 20% sample data used to test the model was never part of the original training data, and the results showed the same for the testing data as for the training data.

The falling trend also suggests an explanation for the very high accuracy rate of the model to begin with. The training data is coming from test automation, which produces nearly (but not exactly) identical results every time it is executed. If a given test fails with the message “The window failed to close after clicking OK” then it will say that same message every time the test fails in that way. The text of the failure may change somewhat, for example some tests display error messages with data that is derived randomly at test time, or might display call stack data which changes every build, or might record time based information in the text. But key words and phrases in the message will still be identical and in the same order every time. This makes the failure message highly predictable, and thus easy to train a model on. But those identical phrases hold less predictive value when they are new and the system has had fewer chances to see them. The words and terms are less strongly associated with a given outcome than other words, so the model does not emphasize them. This association gets weaker and weaker as the model ages.

The highly predictable nature of the text is contrasted against human generated text, where the string describing a problem might be “I cannot open the window” or “The app will not start” or “The app won’t run” all to describe the same behavior. By contrast, the repetitive nature of automation failures is much easier to classify. It is after model ages, and the value of the repetitive text diminishes by virtue of fewer instances in the sample that the predictive capabilities start entering ranges typically seen for matching human generated text.

6.2 The actual work was not ridiculously difficult

Build the experimental system took approximately two weeks. This was mostly time learning the APIs and tools, as well as time building the data preparation tools. A couple of days were given to experimenting with different model types to pick the model and settings for input. A couple of days were given to build a tool to iteratively call the model and generate the output.

Once built, the actual model took less than an hour to collect and prepare 90 days of training data, and approximately five minutes to re-train the model. The nature of logistic regression is that actual prediction is near instantaneous.

The point here being that the basic work was easily within reach of a typical software engineer with a modest amount of training in machine learning.

Another point glossed over here is the amount of work spent tuning the model. For sake of the experiment, I avoided tuning the model completely. The goal was just to see what a naïve approach could generate and how accurate the results would be. In most business applications, achieving the highest results possible is an important goal, and this frequently consumes a lot of time and demands more than a simple exposure to machine learning to accomplish. Model training can involve anything from different data selection and preparation (usually a big factor on the results) to changing behavioral parameters on the model builder (requires sophisticated understanding of how model training works) or selection and maybe combination of different models.

The summary on all of this is that the starting costs of this problem were low and rather simple. The eventual costs could remain low, or drive high and complex, depending on what demand one wanted to place on the results and the accuracy required.

6.3 The results probably should not be trusted to let a computer automatically triage failures

Whether or not one should let the trained model in this experiment automatically make triage decisions about bug resolutions probably depends on one’s tolerance for risk. But mathematically, the recall numbers on the “FIXED” category are pretty straightforward. At 97%, 3 out of every 100 bugs that would have normally been fixed by the product team would be rejected automatically by the system.

The number of bugs this represents on a regular basis to the Office team varies a great deal, but during the time the experiment was running, the weekly average bugs created by the automation system was 426, and the fix rate was (a somewhat low) 14%. This works out to 426 * .14 * .03 = 1.7 bugs a week that would have been fixed by the product team being automatically punted by the trained model.

One way of working with this might be to let the system communicate its recommendation and see if that at least makes product team triage decisions more efficient. Having a human see the decision that the model predicted might offer some protection against cases where the computer gets the prediction wrong while still leading the human to that decision more quickly.

But even that may not be a good idea. For reasons discussed next.

6.4 The results may reinforce bad decisions

There is a joke that says “To err is human. To really mess things up takes a computer.”

The training data is based on historical decisions. There is nothing in the data that says the decisions were good decisions. Unfortunately, there is little, if any, record correlating automation failures, bugs resolved other than fixes and events that later changed that decision. Failures encountered later are sometimes traced back to the first time the failure was seen in automation, but even when done data is not available in the system that records this in a consistent and trackable fashion. It is not uncommon that a failure resolved “WON’T FIX” or “POSTPONED’ occurs later during pre-release usage or even in the hands of customers, but in going through the process of reporting the failure and eventually fixing it nobody identifies the original automation bug that found the failure.

This means that if the team made bad decisions when resolving automation failure bugs that the training data will see those decisions, predict a similar decision for similar failures, and reinforce the bad decision in its prediction. A product team that errs on the side of blindly trusting the guidance of a computer trained based on their own bad behaviors may only get faster and more efficient and making the same mistake again.

Some ideas to consider which might mitigate against such problems:

1. track more closely errors resolved not “FIXED” which result in problems experienced by customers later
2. use other data, such as failure frequency rates, to temper the resolution, or to re-activate bugs that might have been previously resolved not to fix
3. train the model based on customer telemetry data to discover things like high frequency code paths, or features and product behaviors that have a high incidence rate in support traffic

6.5 The results can probably be trusted to guide teams toward further investigation

It is possible the output of the model could be useful for engineers needing to investigate failures. For example, if a model predicts that a result is very likely to be a duplicate of another bug, then an engineer is probably very likely to start first looking for other failures with similar text in the message rather than beginning an investigation into the reasons behind the failure.

If an engineer has decided that a failure might merit fixing, then the text weighting in the model could help as well. Knowing that specific words, or filenames, or maybe portions of a call stack caused the model to predict “FIXED” could offer a strong hint as to where the underlying cause of the problem is.

But even more interesting could be the value in helping teams target improvements in their test engineering. Any team with a suite producing failures that predict heavy for resolutions other than “FIXED” ought to make some sort of change in test automation or in the product to cause the suite to yield something more valuable. Either a very stable run of the product, or failures that are taken seriously and drive to fixes. So just the raw prediction hint is a clue that maybe there is something amiss. Maybe the team has a decision making bias toward “NOT REPRO” and needs to invest in better tooling. Maybe the wording in the test failure is using words that are too vague. Maybe a given test library needs to be made more robust. Maybe the product itself needs to provide more testability access so that failures are more obviously a product problem instead of appearing to be a test artifact.

7 Summary

The experiment overall was a success, and demonstrated the following things:

1. A trained model to predict bug resolution based on automation failure messages can achieve very high accuracy ratings

2. The trained model needs to be kept current to sustain the high accuracy ratings
3. Using simple models, the training data suggests potentially useful information regarding how certain words in the message drive different predictions
4. Despite the surprisingly high accuracy, it is doubtful that such a trained model could be trusted just yet to replace the human in the decision making process

Of course the experiment also raises many new questions, all available for exploration at another time. Most certainly, it seems human job security is still safe and intact, because if anything it appears that trained predictive models may prove to make people better at doing the job they already do.

References

Microsoft, “Multiclass Logistic Regression” https://msdn.microsoft.com/en-us/library/azure/dn905853.aspx (accessed June, 2015)

Ng, Andrew, “Machine Learning”, Stanford University online course at Coursera, https://www.coursera.org/learn/machine-learning/ (accessed September, 2014)

Robinson, M. P. Test failure bucketing, U.S. Patent 8,782,609 July 15, 2014

Wikipedia, “Multinomial logistic regression”, https://en.wikipedia.org/wiki/Multinomial_logistic_regression (accessed June, 2015)

Wikipedia, “Overfitting”, https://en.wikipedia.org/wiki/Overfitting (accessed June, 2015)