Machine Learning NanoDegree

Error Detection Models
Leveraging Machine Learning for QA Automation

Author: Quentin Thomas

Capstone Project

Date: August 27, 2017

overview

This report is to share the performance and insight learned from a new approach to Quality Assurance which makes use of Machine Learning. It encompases all the findings after rigoruous experimentation with real world datasets gathered from the web. Hopefully, with what we've learned we can take the field of QA to new heights with the help of some of the most sophisticated machine learning algorithms and further experimentation and research. This report will thoroughly explain our findings of the answer to the question of whether or not Machine learning can be used to improve software Quality. It turns out that not only can Machine Learning be used in the field of software Quality Assurance, but it just might be possible to completely take over the QA field as a whole due to the level of accuracy that can be achieved with today's Machine Learning algorithms. We will show how we compiled text data from the internet in order to train a learning hour for the task of error detection which is a very human intensive task in Quality Assurance today.

I. Definition

Quality Assurance has become an important field in the technology industry in recent years. The idea that software products could be delivered to users around the world in a quick fashion without consideration of the level of quality diminishes the more connected our software and services become.

A cambridge study conducted in 2013 found that software bugs cost the economy a whopping $312 billion dollars a year[PRweb]. This obviously leads to frustration to end users as well as the engineers that build the products.

As a result QA teams around the world make a tremendous effort to reduce the effects of the bugs in question via several testing techniques and tools in order to increase the level of quality of specific web or mobile applications. This however, only shifts the responsibility from one team to the other, and might create an extreme amount of work for the test team in order to deliver. In order to deal with the increased work loads QA Teams have turned to Quality Assurance Automation.

*In software testing, **test automation** is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes.[1]*

- *Wikipedia*

In this report we want to focus on the area of the definition that says ...

“to control the execution of tests and the comparison of actual outcomes with predicted outcomes.”

This part of the definition is closely related to what machine learning does, and This report will show how effective machine learning can be in enhancing the process of test automation via a real world problem encountered by many qa teams the world over, but first it is necessary to understand why Machine Learning is needed to advance this field further.

Project Overview

In order to ensure that a software application is released in its best state test teams have to perform a series of testing rounds before they can deem a project is ready for production. The amount of time this process takes varies from team to team, but what does not very is a need for a stage called sanity checking, or smoke testing.[SmokeTesting].

Smoke testing is the process of spot checking a web or mobile app as well as possible to verify whether or not an application is in truly in a testable state. There was a time when this process was done manually by clicking every single landing page, and verifying that there were no visible errors on the pages. Now, we have tools such as Selenium, and Runscope that do a pretty good job doing this. However, there are limitations to be considered here. Both Selenium & Runscope, have the ability to pull up a landing page in the browser and show the user whether or not that page loaded successfully via the pages HTTP response. Generally, the technique has been to use these tools to verify whether or not the most trafficked pages were returning a 200 success response codes during a smoke test run. The problem comes in when you have pages that are highly dynamic and have certain containers and divs stacked deep within a landing page. Good examples of this are news sites, and ecommerce web applications. These kind of sites have the tendency to load a landing page and return a 200 response code, but still have errors buried deep down in one of these containers. These errors would slide right under the radar of any general smoke test that is only checking response codes, and then the burden would be left up to the tester to find these corner cases. Leaving this job to the tester however is not a viable solution because testing multiple landing page is very time consuming and resource intensive [LandingPage].

This report will demonstrate how Machine Learning can be used to solve this problem of smoke testing pages based on analyzing the text in order to detect errors within the text. We will show that a model can be successfully trained to recognize even the most crafty error messages, that would otherwise have to be defensively programmed into an automated test script by a test engineer. We also show how Machine Learning model can generalize its' understanding of what true error messages really are thus possibly relieving the testers of having to locate them on their own.

Problem Statement

In this project we seek to solve the problem of web or mobile error detection in text. Many times finding the small areas of an application that has error related messages on web or mobile pages involves an intense amount of exploratory testing by the human tester. We seek to reduce the need for that amount of human effort by training a learning algorithm to learn the difference between regular text and error messages.

This is not a trivial matter as we will show, due to the clever ways we disguise error messages as actual messages in many cases. We will show several of these instances and how we trained an algorithm to be able to distinguish between a real error message, and a message that simply was normal text that we see everyday on the web.

We will solve this problem by searching the web for hundreds of examples of normal text that may include the following, sentences, phrases, titles, and quotes. In addition, we will also search the web for error messages of varying kinds. These error messages can range from blatant messages like ***Unable to Find Resource*** to ***Uh oh... something is wrong.*** What is important here is not the amount of data that is manually collected, but the variations and the nuances of the specific datasets. For example, deconstructing an error message and having a machine make sense of it is as close to a Natural Language Processing problem as one could get. Error messages will not always contain blatant 404, or 503 response codes in the text, so we can't depend on these only as a clue for which messages might be an error.

Once the dataset was manually selected and gathered from the web it was important to clean the data as well as reduce the dimensionality of the data itself so as to help the algorithm converge. We will go into detail as to what this process entailed as we continue forward in this report.

Finally, to fully solve the problem we would leverage two algorithms the Naive Bayes Classifier, and the Random Forest Classifier to see which one would be better suited for the job of error detection classifications.

Metrics

In order to successfully understand the performance of the two algorithms we've chosen we leveraged the confusion matrix to get an idea of how well our classifier was detecting these subtle error messages. The structure of our confusion matrix is depicted below where x^i is an instance of a training set in the collected data where *i* can be represented by a 1 for error messages being in the text of the example and 0 for the error message not being present in the example...

	Predicted Error in text (Yes)	Predicted Error in text (No)
Actual Error in Text (No)	x^1	x^0
Actual Error in Text (Yes)	x^0	x^1

In these case each cell will contain the total number of false positives, true negatives, and false negatives, we gather during the testing of the classifiers accuracy. We share the results of the confusion matrix and compare them.

The accuracy of our algorithm can be determined by taking the amount of misclassified messages and dividing them by the total amount of the test set and multiplying that result by 100. This will give us the percentage of the accuracy of the model. The accuracy will be computed by essentially taking the average amount of correct responses from the algorithm. Accuracy is a good metric to consider for any classification problem, especially ours. For a QA team to trust the results of this algorithm, it would need to be correct more times than it is incorrect. Otherwise, it would not be practical to take this manual task away from human testers.

II. Analysis

Data Exploration

As mentioned before our data was carefully crafted by surfing the web for different types of error messages and web text. Our goal was to make sure the error messages and the text were as close, or similar to each other as possible. As a result we took over 500 examples of all these different types of messages and used it as the training and validation set for the accuracy of our classifier. We also managed to throw in text that was clearly regular messages all the way to blatant error messages that had the actual term **error** in it. However, we didn't want the algorithm to use context clues like a bad response code, or strong language that indicated there was an obvious problem. To negate this bias, we enlisted some of the most crafty error messages we could find on the web to serve as a good test.

In order to be successful, the model would need to be able to generalize well. It could only generalize if we gave it data that would be extremely difficult to generalize with. The thinking was if we could get it to generalize with these difficult hard to decipher error messages, we would have no problem getting it to tackle the blatant error messages like.. **404 Page Not Found.**

Our training data consisted of a tab separated value file with two columns. One column was represented by the message, and the other was represented by the flag of whether or not the corresponding message was an error. Keep in mind that the *tsv* file type was a chosen format for the data file so as to negate the fact that we might run into many text examples with quotes and apostrophes in them. We didn't want our algorithm to be hindered by this so we used the tsv as our go to file type for showing the algorithm the data examples.

Here are a few examples of how we achieved this diverse data set, first we will view some blatant text examples that we had in our dataset keep in mind our flag logic can be expressed as …

$$\forall x \in X \{1 \; if \; x \;\; \in \;\; E, \; 0 \; otherwise\}$$

Where *E* is represented by a vector of learned error messages after training.

Message	IsError?
The site's worth a visit just for the brilliant artwork	0
Bold typography makes this page work well	0
It's brilliantly executed and nicely interactive.	0

As we can see in the chart above we have 3 examples from our dataset that are messages, but they clearly have nothing to do with the existence of this project as a whole. Examples like this our needed as it gives the algorithm more room to exercise its degree of belief when it sees something that might not be as obvious. We will view another set of examples in the training data that might be a little more tricky, but still are not really error messages. The data set below illustrates this example…

Message	IsError?
He needs to learn the error of his ways	0
Websites have long played with fun 404 pages–that's the error page you get	0
This page is so buggy	0

As we see above these messages are a bit tricky. In fact if there was a program written to detect 404 errors in text the 2nd example would show up in the result even though it is not a real error message. This is why traditional automation techniques would not be able to work on a problem like this. Real world text can be quite tricky, and we made sure we gathered data that looked as tricky as possible so the classifier would have a good chance at increasing its confidence.

We will now take a look at what we considered to be standard errors we would like the algorithm to learn. These are blatant errors that our classifier should not miss, as they are obvious to even the most non technical person that something is wrong…

Message	IsError?
The page cannot be found	1
Error 522 Connection timed out	1
503 Response from the server	1

Our classifier would benefit from having these types of messages in the the training set as it serves as a good example of what an actual error message looks like on the web with no subtleties.

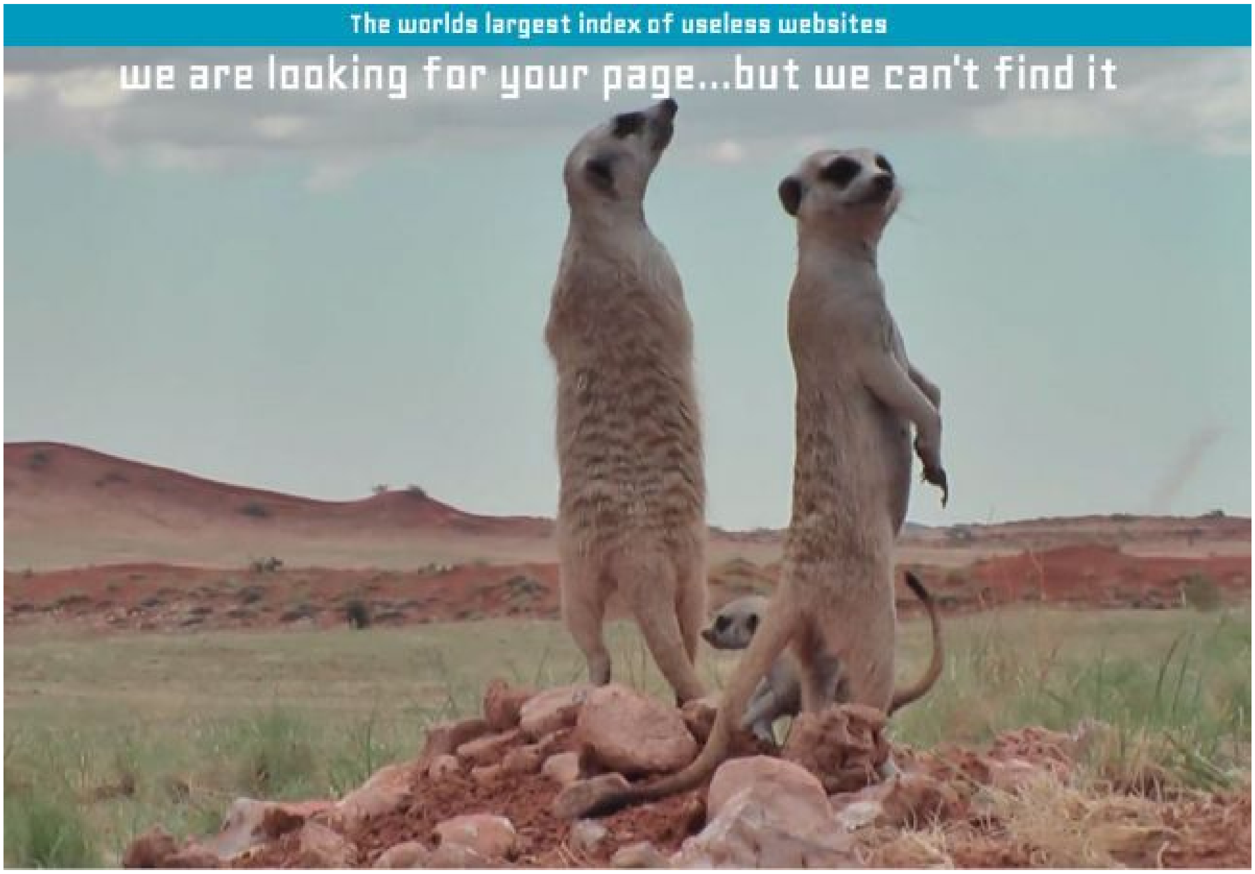
Visualization

Before moving on it's worth it to take at some very tricky error messages that have become commonplace on the web in web design [404 Designs]. We decided to add these error messages into the training set as these would really serve as a difficult test for the accuracy of the algorithm. Here are some images of a few of these error messages that are pretty clever and might be really hard for an algorithm to find the correlations…

Source: http://kualo.com

Clearly the error message has very little context clues in the text ***oh no! Space Invaders destroyed this page! Take revenge on them!***

Another example….



Source: https://www.theuselesswebindex.com

The text here is really subtle. There are no references to 404, or error at all in the text, but our algorithm would still need to know that a page like this is an error.

Making the algorithm have this much diversity in the data set will help it achieve good results.

Algorithms & Techniques

In order to do this successfully we needed an algorithm that had the ability to understand text, and split the results into two categories for binary classification. After doing some research it became clear that the behavior we were targeting was that of a spam filter.

Spam filters have the ability to know whether or not an incoming email belonged in a spam folder, or the inbox of the user just best on several context clues within the text of the email. These spam filters leverage the Naive Bayes classifier along with NLP techniques [SpamFilters]. We decided to leverage this algorithm as our benchmarks and see if we could obtain an accuracy of at least 60%, as any error detector would need to be at least correct half the amount of the time.

The Naive Bayes algorithm which can be expressed in our case if you let *M* represent our message text and you let *E* represent our error we get the following...

$$P(E|M) = \frac{P(M|E) \times P(E)}{P(M)}$$

This will say that our system will determine whether or not a message is an error if the likelihood that that message it seen previously was closer to what the current probability for a message being an error is. In other words, the more evidence the algorithm has in determining whether or not there is an error in the message, the more likely it will begin to conclude that the message it is seeing currently is an error. This was our approach to training the Naive Bayes algorithm.

As a fallback, we tried leveraging a second model which is the Random Forest classifier, which is usually the alternative to Naive Bayes. The Random Forest classifier is significant as it operates as a team of decision trees that vote on the decision of whether or not our message has an error or not.This is quite useful when it comes to classifying text in our case because as we noted before the text we are classifying is very tricky. The data is not that easily separated, so a team of decision trees working to get this done is a much better approach than just have one decision tree trying to find out the binary separation for the messages. However, we enlisted grid searching in order to find the optimal number of tree estimators in order to properly classify these messages.

In addition to the chosen classifiers we also needed to leverage the bag of words technique in order to normalize, and reduce the dimensionality of the data. This was needed because we are dealing with hundreds of examples to train our model. The combination of all these techniques led to us finding our optimal solution for error detection.

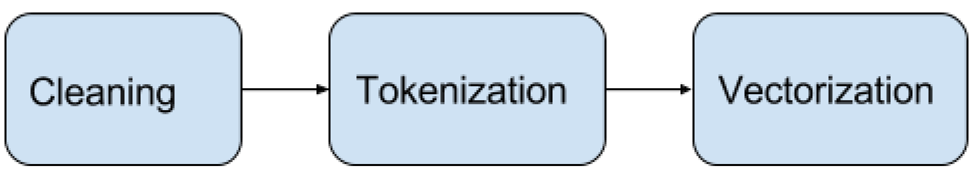
Benchmark

Our benchmark was simple, We would first start off training our Naive Bayes classifier and saving the confusion matrix. This would be our baseline. If any model surpassed the initial one we would then implement our full solution in the new model. Our target accuracy for the project was at least 60% as we believe this would be a viable starting point for the accuracy of any testing tool.

III. Methodology

Data PreProcessing

The data preprocessing phase is perhaps our most important. In order to deal with any type of Natural language processing some initial phases needed to be implemented. We took our data through the following phases....



<u>Cleaning:</u>

In order for us to make sure we dealt with relevant words and numbers and not strange characters we implemented a cleaning process that gave us only letters and numbers by rejecting everything that did not fit into that category. The cleaning process can be expressed as a set of numbers and letters...

$$x \ni \{\neg a - z, A - Z, [0 - 9]\}$$

This meant that everything that was left after this operation was to be just numbers and letters.

<u>Tokenization:</u>

The next step was to tokenize the data by stemming out all the variations of the same word. We did this so as to not have the algorithm consider the same variations of the same word during training. Each word would have to have its own meaning, and not be repeated anywhere else in the definition set. This comprised our tokenization step

Vectorization:

Finally, our vectorization step was our final stage. This step reduced our data down to a multidimensional vector so that it could be feed to the algorithm. The vector consisted of taking 400 training examples that were vectorized into a 500x500 matrix. This matrix was then used for training our algorithm, while the remaining 100 examples that were never seen before were reserved for the test set.

Implementation

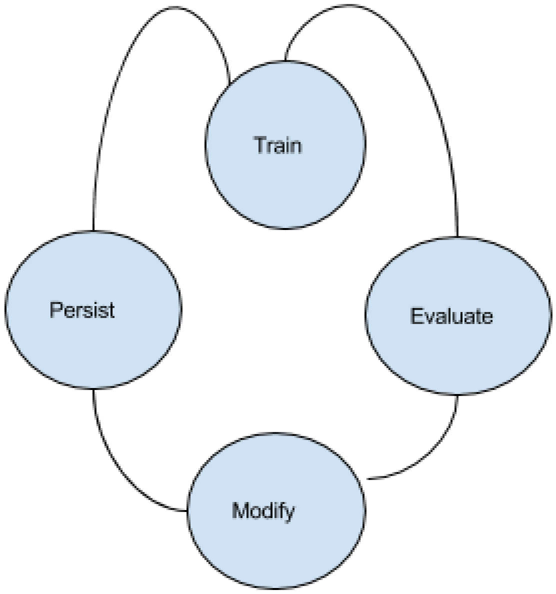
Our Implementation consisted of several functions that took our data through the 3 stages of our preprocessing phase, and finally, two general functions for actually running the Naive Bayes Algorithm and The Random Forest classification algorithm. It was during this phase we noticed that we didn't need that much data to train either of these algorithms as both of them were known to converge with a small amount of examples. See our reference to study An empirical study of the Naive Bayes Classifier [Study].

Refinement

The beginning of our refinement process included separating our 500 examples into two categories the training set, and the test set. This consisted of a 20% split of our original examples. This left us with an allocation like below…

Training Set	Test Set
400 Examples	100 examples

The diagram below shows our full refinement process for training our algorithm end to end..



Training:

In this stage we trained our classifiers on 400 examples of messages.

Evaluate:

In this stage of the refinement process we leveraged our grid search algorithm to find the most optimal parameters for our Random Forest Classifier. We performed a grid search with a ten fold cross validation exploring the following hyper parameters

Random Forest Classifier

Hyper Parameter	Explored values
n_estimators	[10, 20, 30]
max_features	[2, 100, 300, 500]
max_depth	[5,10,15]

Our Grid search suggested that the best hyper parameters for this stage was the following..

Hyper Parameters	Optimal Values

n_estimators	20
max_features	500
max_depth	15

For the Naive bayes classifier the refinement stage included saving the testing whether or not the model performed higher than 60% if so the model would be considered for persistence, if not we would discard this approach.

Modify:

This process consisted of taking the suggested hyper parameters from our grid search and training a new Random Forest classifier using the new hyper parameters. For our Naive Bayes there was not much to do here but discard the model if it did not get an accuracy of at least 60%.

Persistance:

In this stage we saved the models that performed the best. If the accuracy was not achieved by either of the models, then ideally this cycle would repeat back to the training step.

IV. Results

We were surprised of how fast our algorithm converged to acceptable accuracy. We will now show the best performances we received from both models.

Model Evaluation and Validation

As mentioned earlier we trained our algorithm on 400 examples, and held out 100 for validation purposes. In order to get a good idea of the performance of our classifier we leveraged the confusion Matrix.

Below are the confusion matrixes on the validation set for both model. Keep in mind this was on data neither classifier had seen before...

Naive Bayes Classifier Confusion Matrix

	Predicted Error in text (Yes)	Predicted Error in text (No)
Actual Error in Text (No)	39	15
Actual Error in Text (Yes)	7	39

This means that the Naive Bayes classifier had an accuracy of 78% on the data that it never seen which was much higher than our target which was 60%.

Random Forest Classifier Confusion Matrix

	Predicted Error in text (Yes)	Predicted Error in text (No)
Actual Error in Text (No)	49	5
Actual Error in Text (Yes)	8	38

The Random Forest classifier showed a much stronger accuracy on the validation set with an accuracy score of 87% which was really impressive.

Justification

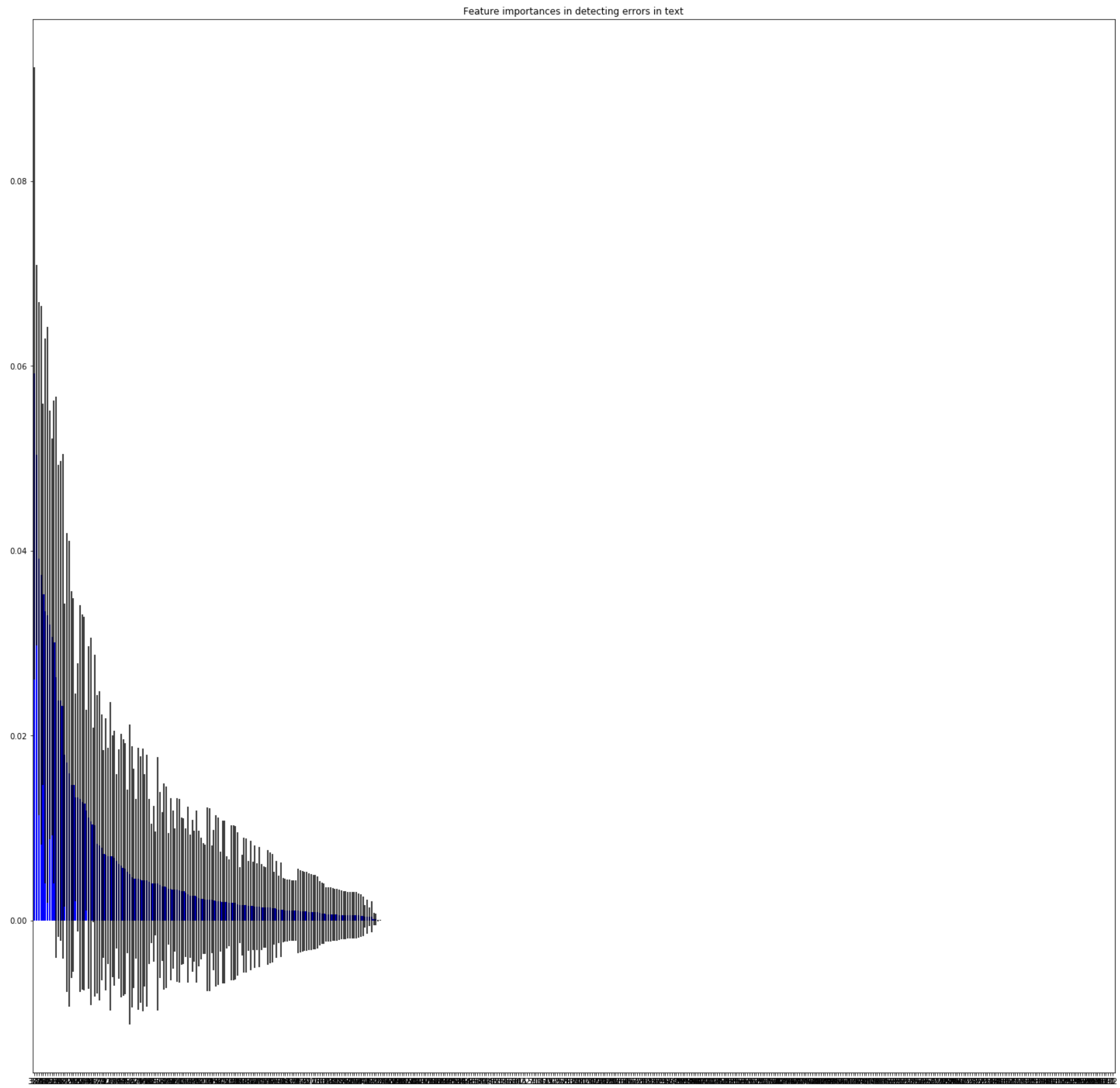
The accuracy of both models excelling above our target accuracy of 60% servers as justification that this is a good approach to error detection. Treating error detection as a Natural Language problem is very reasonable as the results show. The good thing is we didn't need to focus so much on finding tons of data, but rather our focus was on making sure that data was sensible to the problem at hand. Our approach to using subtle tricky language in our data set is what led to the great performances of the models during testing. This suggest that it is vitally important how the data is chosen for an algorithm.

V. Conclusion

We ranked all 500 features in the training data in accordance to their importance to the Random forest classifier being able to find the difference between error and none error. We found that the top 3 rankings showed the following numbers…

Rank	Feature Number	Importance probability
1	382	0.59
2	340	0.50
3	273	0.39

We can see in the chart below the standard deviation for all the importances of the features detected in the dataset…



The chart above gives us a good idea how all the features we collected rank in importance to the algorithm. We can see that it has detected quite a few features that are clearly visible for the algorithm to determine whether or not something that it is seeing is an error. This further suggests that there are clues in the data around us that can be used by algorithms to build really sound QA systems and tools.

We set out to prove whether or not it is possible to leverage machine learning in the field of quality assurance. It turns out that error detection is a big part of quality assurance, and the fact that a machine learning algorithm can be trained specifically to achieve this is a very promising sign for further research into this space.

There are a few improvements to consider, the first is that we only trained our system to deal with english related text. The web is international and is made up of many dialects and language symbols that are completely different from American style english. A good system would need to be able to detect errors on sites in other languages as well, so there is a need to come up with a more generalized approach for testing in different languages.

The next thing to consider is having a system that can visually see the text. For example, Imagine a system that would have the ability to recognize status codes like 523, 404 etc.. This would make web test automation much more sophisticated as we will leverage text as well as visual queues. This ability is already possible in the field of Machine Learning due to the breakthroughs in Convolutional Neural Nets. However, there is much work to be done in gathering the data sets that could be used for training.

Quality Assurance will only get more difficult the more our systems become interlinked and we will need better and better tools to help us with this. We believe that Machine learning is a good place to start based on the results we’ve seen here.

Some improvements we seek to make in the future is to design a real time sequence model that is responsible for searching land pages autonomously in order to find issues within an web domain. This would be the ideal next step as we have already proven that it is possible to detect even the most subtle error messages. We would like to see Machine learning merging with current testing techniques in order to boost the quality of all software around the world.

References:

Spam Detection and Natural Language Processing [Paper]

Software bugs cost the Economy $312 Billion per year [Study]

Smoke Testing a Necessary Evil [Study]

An empirical study of the Naive Bayes Classifier [Study]