

proposal

August 26, 2017

1 Machine Learning Engineer Nanodegree

1.1 Capstone Proposal

Quentin Thomas

August 18, 2017

1.1.1 Error Message Detection

In the domain of software Quality Assurance there is a need to use human labor to Smoke Test different websites or mobile apps. The Smoke testing process consists of spot checking whether or not a web app, or mobile app are in a testable state before the quality team can begin their real process of integration testing.

This initial step of smoke testing is extremely vital to the testing process, as it keeps teams from starting a round of full regression testing on a web or mobile site before it is ready to be used. See [Smoke Testing a Necessary Evil; mindlance inc.](#). In addition to this, the pressure is on the QA members to limit the number of software bugs in products because such bugs costs the software industry \$312 billion dollars a year! [Cambridge University Research](#)

This process however, has become very cumbersome in recent years, as the complexity and size of many web and mobile site pages have grown considerably since 2010. See [Everts; Radware inc.](#). It is a necessary job for QA engineers to make sure each one of these pages are loading correctly after deployments. This is a simple task if the site or app in question has a small number of pages to consider. However, based on the study above, we can safely conclude that small web pages are a thing of the past.

One way that teams have tackled this is to utilize a tool like [Runscope](#) to monitor when a specific api call is returning a 200 response. This is a fine solution for general website verifications, but it does not solve the case when a page returns a 200 response code but shows error messages within the html being returned to the user by way of the web or mobile app. In my career as a QA automation Engineer I have seen this happen on various websites and mobile apps plenty of times, and currently the solution is to leverage tools like Selenium to try to check whether or not

certain areas of a particular page is loading, but this is very brittle and difficult to do effectively. Hidden error messages on landing pages create a big problem for test teams that must be solved.

1.1.2 Problem Statement

As a quality professional there is nothing worse than navigating a web or mobile landing page and finding glaring error messages. The obvious solution is to spot check all the pages and look for all the cases in which the pages load successfully, but have error messages somewhere on them. This is a good step in the right direction, but it is extremely time consuming, especially when SEO professionals encourage businesses to have more than 40 landing pages to increase leads 12x over. See [Browning; SEO](#).

If a business is encouraged to increase the number of landing pages on their website, then we can expect the number of testable pages that test teams would need to verify would increase as well. It is clear an automated approach is needed to solve this problem. However, the automated approach would need to be slightly more sophisticated than checking for a specific `div` and `web containers` with selenium. The solution would have to read through all the text on the page and discover the error messages on the page on its own. The solution would then report back to the tester whether or not a certain page has a problem. Is there a possibility this problem can be solved in a automated fashion via Machine Learning?

1.1.3 Datasets and Inputs

In order to leverage Machine Learning to tackle this problem we will need to leverage a good classification algorithm, along with really good data examples of random internet text as well as random error messages. Assuming we are going for a binary classification problem in regards to the pages texts under test, we will say that 0 will represent no error messages found while 1 will represent an error message has been found somewhere on the page. This notion can be expressed as

$$\forall \tau \in T \begin{cases} 1 & \text{if } \tau \in E \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

In the equation above we let τ represent the text on the page T , and we represent E as a specific error message that might be present on the page.

The data would also need to be several examples for the algorithm to help distinguish between real error messages, or just normal text. An example of what this could look like is below..

PAGE	Page Text	Has Error?
Landing Page 1	Welcome to our home page...	0
Landing Page 2	Unable to find resource url for...	1
Landing Page 3	Want to make money fast?.....	0
Landing Page n

It is clear from the chart above we are dealing with a NLP related problem. Language on web

pages can be very diverse, and the algorithm would need to have good datasets that can help the algorithm determine the difference between seeing text that says **He should learn the error of his ways** and **error thrown loading window pane** on a specific page. On the other end of the spectrum, the data set also needs to be able to find error messages that might not even contain the term `Error` in it at all. For example, **Resource not found** is clearly an error to a human tester, but this is just simply text to a machine. These are the type of examples our data set would need to include in order to train for this task. In short, the algorithm will take as input the page text, and return as output a 1 or 0 as a response which map to error found or no error found.

1.1.4 Solution Statement

The solution to this problem is needed for two reasons. These reasons will be discussed here.

- First, a human tester needs to free up themselves from mundane smoke testing tasks. QA members spend a large amount of time and energy smoke testing landing pages. Landing pages are also important to test on mobile apps as well. For example, deep linking a user from specific site into a mobile app, must work if a business is to obtain their users attention. If landing pages aren't loading correctly it looks really bad on the software team in question. The solution here is simple; the goal is to rid QA of the manual labor of verifying this, while at the same time creating a more sophisticated way to detect errors rather than just simply depending on teams to do so.
- Second, As mentioned before, current automation techniques require a automation engineer to use selenium to look for error messages. This is not practical because the burden is placed on the engineer to make sure he or she hard codes the error that selenium is to search for that might or might not be on the page during the time the test engineer creates the test script. This is a tedious and brittle approach. Even the best QA automation engineers would come up short looking for error messages this way because there are many flavors of error messages out there, and everytime a new error message is discovered, the code would need to be updated to reflect the new error message that needs to be detected. This iterative approach to automation is extremely time consuming and should not be the engineers responsibility.

Therefore, we will develop a Machine Learning approach to tackle this problem. We will train a classifier with various examples of web text and error message text. These examples will be fed into the algorithm for training. The algorithm will train until it reaches a reasonable predictive capability that can be used on any landing page a QA desires. This way, the algorithm can evaluate hundreds of pages and find the ones that has errors and separate them from the pages that don't. This will allow the test team members to locate these pages quickly, and take the necessary action.

1.1.5 Benchmark Model

We already have algorithms that are quite good at sentiment analysis, or detecting whether or not an email contains spam or not, as well as categorizing news stories by their specific content. These algorithms have been quite successful. The algorithm that governs the behavior of spam filters would serve as an excellent benchmark for our project here. Spam classifiers leverage

the Naive Bayes algorithm in order to determine whether or not a user is getting an email they shouldn't. These algorithms fit our problem as we would only need to tweak it to fit our purposes. Some of the benefits of this algorithm are the fact that it doesn't require that much training data before it converges to good performance. See [Giyayani;Desai](#)

There are already several cases of Naive Bayes classifiers being the go to algorithm for analyzing text on the web. Our problem fits right in with this use case. Our system would need to have an accuracy greater than 60% in order to be considered usable for any QA member. The good news is this benchmark is not impossible achievement for this algorithm as there are cases where it has achieved accuracies greater than 90% on certain problems. See [Rich](#). As a result of its success, we will use this algorithm and train it to solve our problem. However, the way we will evaluate its usefulness will be much different than a spam detection system.

1.1.6 Evaluation Metric

There are two evaluation metrics that we can use to determine the usefulness of this tool. We will explore these two metrics and how they relate to our error detection system.

LogLoss The log loss function which can be expressed in our terms as

$$- \log P(a|t) = -(a * \log(t) + (1 - a) \log(1 - t)) \quad (2)$$

where a is actual classification and t represents the models predicted classification for the text on the page. This will be a good way for us to determine whether or not our algorithm is converging.

There is however another metric we will need to leverage which could be considered much more important.

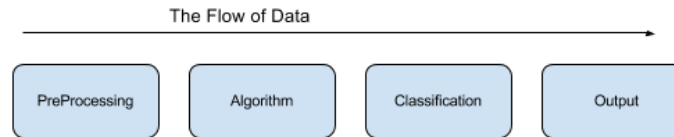
Confusion Matrix The confusion matrix will give us a good idea of how we are doing with our classifications. We want to leverage this metric so that we can lower the number of mis-classified pages. The layout for our confusion matrix can be viewed as...

	Predicted Error on Page (Yes)	Predicted Error on Page (No)
Actual Error on Page (No)	t1	t0
Actual Error on Page (Yes)	t0	t1

We will seek to lower the number of false positive and false negatives as a large number of these will damage the credibility of the tool. Throughout the training and test cycle we will consult our confusion matrix before we consider the algorithm useful for QA purposes.

1.2 ### System Design

Even though the problem before us has many layers to it, and as a result the actual design of the system will need many parts in order to make this tool functional. For example, The system will need to pass the data through several stages before the outcome of the algorithm can be known to



Flow of data

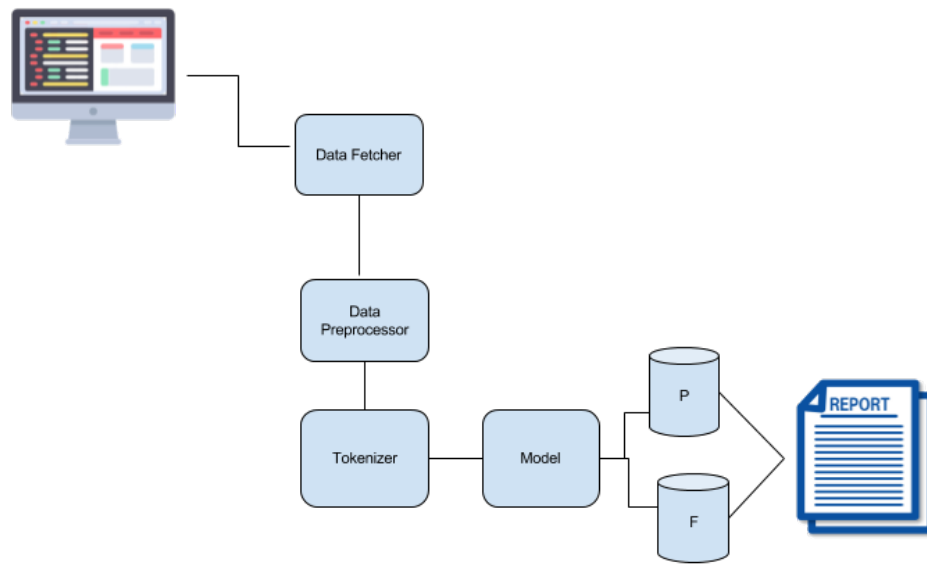
the tester. In this section we will explore these stages in depth. First, we will take a look at these stages from a high level which will help us grasp the task of this system.

Pre Processing The Preprocessing stage is where we will make sure all the unstructured data that comes from the web is parsed and tokenized so that it can go from text data to numerical data. Once this is complete we will then be able to feed the result of the preprocessing phase to our algorithm.

Algorithm The algorithm as we mentioned before will leverage the Naive Bayes Classifier and NLP techniques in order to gain insight into which page has something we deem to be error messages. There is a possibility that we might not get the accuracy we are pursuing which is above 60%. In the case where this accuracy cant be acheived we will consider other algorithms to try to reach an accuracy of 60%.

Classification After the algorithm does its work the classification stage can begin. In this phase we will place the page in question into its appropriate place. We will have two categories, landing pages that have no errors, and landing pages that have errors.

Output Finally, the output stage will be responsible for presenting the results of all the stages to the user in a digestible manner. This stage is where the tester can use this information to take the actions necessary.



System Diagram

Now that we know the stages the data will go through, we will now consider the system architecture. As a whole we want the system to be extremely accessible. We want the user to only have to input the url of the landing page, and the algorithm handle the rest. Below is a full chart of the system architecture...

The data fetcher will be responsible for fetching the data from the specific landing page given. Once the landing page data is fetched then the data preprocessing can begin. Once the data preprocessor is done with its work the tokenizer will make sure all the data is ready to be fed to the model. After the model makes its prediction it will separate the landing page in its appropriate bucket. These buckets can then be curated to generate a report for the QA team member so that it can be read and hopefully shared.

1.2.1 Training The Model

The heart of this system of course is the Machine Learning model. In order to ensure its flexibility, and accuracy we will need to train it with hundreds of examples of regular text from random websites, as well as specific error messages that it might come across. We will split all of the data into 3 categories, the categories are listed below

- Training
- Testing
- Validation

We will leverage the training data to give the model a chance to learn the differences between error messages and regular web jargon. After training we will use a small portion of the data

to run it through several rounds of testing. This stage will allow us to discover whether or not some changes being made to the algorithm to reach an accuracy of at least 60% as anything less is unusable for any QA member. Finally, we will use the validation set to test how well our model performs in the real world. We will point it towards websites that have error messages, and sites that don't. If this is successful we will hopefully have a useful error detection tool for landing pages that can be used for smoke testing and quite possibly a bigger AI system.

1.2.2 References

1. Smoke Testing a Necessary Evil; MindLanceInc. [presentation](#)
2. Cambridge University Study States Software Bugs Cost Economy \$312 Billion Per Year [Article](#)
3. The average web page has almost doubled in size since 2010; Tammy Everts [Article](#)
4. How many pages should a website have? [Article](#)
5. Spam detection and NLP [Paper](#)
6. An empirical study of the Naive Bayes Classifier [Paper](#)