proposal

August 27, 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 Neccesary 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 neccessary 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 possiblility 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

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

In the equation above we let `t` 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 destinguish between real error messages, or just normal text. In the table below are a few examples of the text the algorithm will be trained on from the `error_detection_training.tsv` file...

Message	IsError?
oops... There seems to be a problem loading the page	1
He needs to learn the error of his ways	0
Want to make money fast?	0
404 Error, Looks like no one else is here	1
The traffic tonight in the area is going to be significantly busy	0
We seem to have a high level of web traffic tonight sorry for the delay	1

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. There are also several examples of very creative error messages on the web that sound like normal sentences. In fact, there is a whole genere of web design that encourages creativity with error messages. See BestErrors.

Our algorithm will need to be flexible enough to detect subtle error messages that are not as straight forward. As a result a training dataset of 1000 examples of messages and text gathered from random internet sites will be compiled for the training. The training set will also include very creative, error messages that might look like normal sentences in order to allow the algorithm to find the correlations. All training samples will be labeled by in the `IsError?` column by either a 1 or 0. The idea is to have enough of these random messages on the web that are so diverse it will give us a chance at a higher accuracy.

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.

In order to compose the training data we will manually gather different examples from around the web of messages of different types. Some of the messages will be errors, some will be random text, others will be very tricky error messages that might be disguised as normal text. It is our assumption that this diverse collection of messages will creat a good decision boundry for the algorithm to utilize when the tool is being used in the real world. In addition, We will make sure that there are several examples of common error codes which we expect, our system to recognize as errors like 400's - 500's. The training file will also need to be in a tab separated format as a comma separated format would make the preprocessing very tricky, as we expect a large amount of the text online to obtain commas. To avoid this complexity the data will be imported as a `.tsv` file.

1.1.4 Solution Statement

As mentioned earlier we will be faced with a Natural language processing problem. In order to come up with a viable solution the following steps will be attempted.

1. We will implement the Bag of words algorithm in order to group together like terms that can be useful for helping us decipher whether or not we are encountering an error, or just plane text

2. We will need to reduce sparsity because of the curse of dimentionality rule. We expect that there will be a large amount of words that will be collected by our algorithm, but all of these can not be fed into the algorithm. As a result we will specify a maximum amount of words to consider while training our model with the training data. We will also remove stopwords as they are not to relevant for what we are trying to acheive.

3. It also might be neccessary to perform stemming in order to reduce the amount of repeat workds in the dataset. We are only interested in the root words and not past or present

tenses. This will keep our algorithm focused on the root of the words in the error messages that really matter.

4. Finally, we will split the data into a training and test set so that we can get an idea of how well our algorithm is working.

Once all these stages are complete we will be able to leverage the NaiveBayes Classifier in order to predict whether or not the message our system finds on the web is an error. Generaly, Random Forest and NaiveBayes have been used for binary classification of text, so it would not be unreasonable to leverage one of these two algorithms as the classifier.

The solution to this problem is needed for two reasons. These reasonse 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 practicle 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 responsiblity.

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 neccessary action.

1.1.5 Benchmark Model

We already have algorithms that are quite good and sentiment analysis, or detecting whether or not an email contains spam or not, as well as catagorizing 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 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 greator than `60%` in order to be considered usable for any QA member. The good news is this benchmark is not impossible acheivement for this algorithm as there are cases where it has acheived accuracies greator 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

$$-logP(a|t) = -(a * log(t) + (1-a)log(1-t)) \tag{2}$$

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.

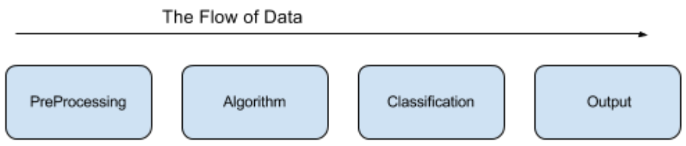
Confusion Matrix The confusion matrix will give us a good idea of how we are doing with our classifications. We want to leverage this metric so that we can lower the number of mis-classified pages. The layout for our confusion matrix can be viewed as...

	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 flase 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 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.



Flow of data

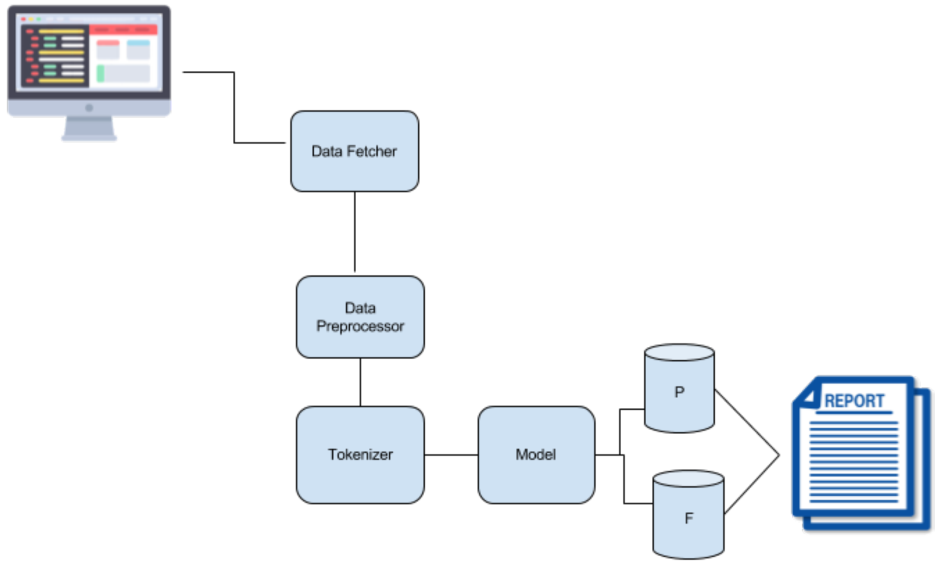
Pre Processing The Preprocessing stage is where we will make sure all the unstructured data that comes from the web is parsed and tokenized so that it can go from text data to numerical data. Once this is complete we will then be able to feed the result of the preprocessing phase to our algorithm.

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 possiblity 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 neccessary.

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



System Diagram

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 catagories, the catagories 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 trianing 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.

We expect to go through several stages of refinement in order to verify we have the most optimal classifier. In the case we use random forest we might consider leveraging Grid Search in order to find the best hyper parameters for the algorithm. However, Naive Bayes will be the first attempt, and leveraging the bag of words algorithm and text cleaning approaches should be sufficient enough to reach the accuracy we are after.

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 successuful we will hopefully have 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 Neccesary 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 has? Article
5. Spam detection and NLP Paper
6. An emprical study of the Naive Bayes Classifier Paper