Spam Filter Study Project Report

Mihir Patel, Jacob Huber, and Ben Fintel

Introduction

Many websites utilize content provided by users. This kind of content can be constructive for the topic of the website, however, sometimes non-relevant content can be submitted as well. This non-relevant content can be referred to as “spam”. An example of spam could include content which contains attributes that are not allowed for the given website. This is the kind of spam that will be primarily researched in this paper. The research is based on a specific problem that we are running into. We are creating a website that allows users to post reviews of specific things. This website needs a method of filtering out content that violates our review posting guidelines. We want to filter out this type of content because we want to provide a platform that contains as much information as possible that pertains to our user base. Having spam in our site wouldn't be useful to the user since spam doesn’t pertain to the information that users would visit our site for. In fact, the addition of spam on our site would muddy the useful information thus making it harder for the user to determine which information is useful to them. To do so, we can use a spam filtering package. We found that there are many different packages available to solve this task. To find the best package for our use case, we studied multiple different packages and recorded the results based on specific metrics. We hope others will be able to utilize our results to find the best Node.js based spam filter for their project.

Motivation

We want to determine which of the spam filter packages that we have tested fit into what type of work that they say that they do. Which are the fastest, best at filtering, or the easiest to implement. Some packages may work in different types of uses depending on what specific filter it’s going for. However, we want a spam filter that works best for a website where users can leave reviews, but what we find works best for us might not be what is needed for different developers. The spam filters will be running against the same tests that we have created and will be measured against speed, correctness, and usability. In the end, we want to be able to state, with confidence, which filter packages are best for different types of everyday filtering that different applications may need. In short, we want to answer the following questions:

- Which spam filter is best overall?
- Which spam filter is the best for long text?
- Which spam filter is the best for short text?
- Which spam filter runs the quickest on average overall?
- Which spam filter runs the quickest on average for short text?
- Which spam filter runs the quickest on average for long text?
- Which spam filters were easiest to implement?

Background

For this study, we are implementing the spam filters utilizing the Node.js programming language. We are pulling each spam filter from NPM. These spam filters packages come with various usage rates and capabilities. Table 1 shows the different spam filters we utilized with the link to their NPM package repository.

Table 1. The spam filters utilized for the study.

Name	**Link**
spam-filter	https://www.npmjs.com/package/spam-filter
spam-check	https://www.npmjs.com/package/spam-check
spam-detection	https://www.npmjs.com/package/spam-detection
bad-words	https://www.npmjs.com/package/bad-words
leo-profanity	https://www.npmjs.com/package/leo-profanity
retext-profanities	https://npm.io/package/retext-profanities
swearjar	https://npm.io/package/swearjar
censor-sensor	https://npm.io/package/censor-sensor
badwords-filter	https://www.npmjs.com/package/badwords-filter
noswearing	https://npm.io/package/noswearing
profanease	https://npm.io/package/profanease

Work Distribution

Mihir:

- Wrote code to add entries into the CSV
- Implemented spam-filter
- Implemented spam-check

- Implemented spam-detection
- Fixed bugs with all spam filters
- Wrote timing code

Jacob:

- Implemented data import
- Implemented bad-words
- Implemented leo-profanity
- Implemented retext-profanities
- Created results charts
- Wrote timing code

Ben:

- Wrote spam test cases
- Implemented swearjar
- Implemented censor-sensor
- Implemented badwords-filter
- Implemented noswearing
- Implemented profanease

Approach

Since the goal of this project is to sift out only the fastest and most efficient spam filters, we need to have a quantitative set of measurements to arrive at a conclusion. The categories we plan to test for each spam filter we use are: best percent spam caught for large text, best percent spam caught for small text, speed of spam caught for large text, speed of spam caught for small text. These results will take account of different test cases so we can cover a broad range of possible use cases. The way we measured this was in a few ways. The first method was the human observations, which was what we could visually see and calculate ourselves from each run of the different spam filters. We were able to see how much each spam filter has missed or how much it caught that wasn't necessary to filter out. The second method of measuring was through computed calculations. We displayed the data for the exact percentage of filtering that each filter caught for our test cases and how fast each filter ran. Using the combination of our human calculations and what the computer is able to calculate, we were able to effectively measure which filter performed the best in each given situation. We created 20 test cases total where the test cases were mixes of bad words and unwanted characters. 17 of the test cases had short text while 3 of the text cases had long text. 15 of the test cases were identified to be spam by human judgement while 5 of them were identified to be valid. Table 2 shows some of the example test cases we used. We have redacted the test cases with profanity for this report. We also noted down the difficulty of implementing each spam filter so developers were aware of the effort required. Other readers will be able to utilize our findings to select the best spam filter for their project.

Table 2. Example test cases with both spam and valid texts.

Test Cases	Spam or Valid

[illegible]	Spam
THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS THIS TEACHER SUCKS	Spam
This dorm is cool	Valid

Results

We wrote our code to run each test case through the spam filters and added the results to a CSV. The CSV contained the time each spam filter took on each test and whether it was able to accurately categorize the test case. From performing analysis on our data, we found the most accurate spam filter overall was retext-profanities. As shown in Figure 1, it was able to correctly identify 13 test cases correctly. spam-filter, spam-check, and swearjar were the worse performing spam filters where it only correctly identified 8 test cases. If a developer is interested in getting the most accurate output overall, retext-profanities would be the best choice.

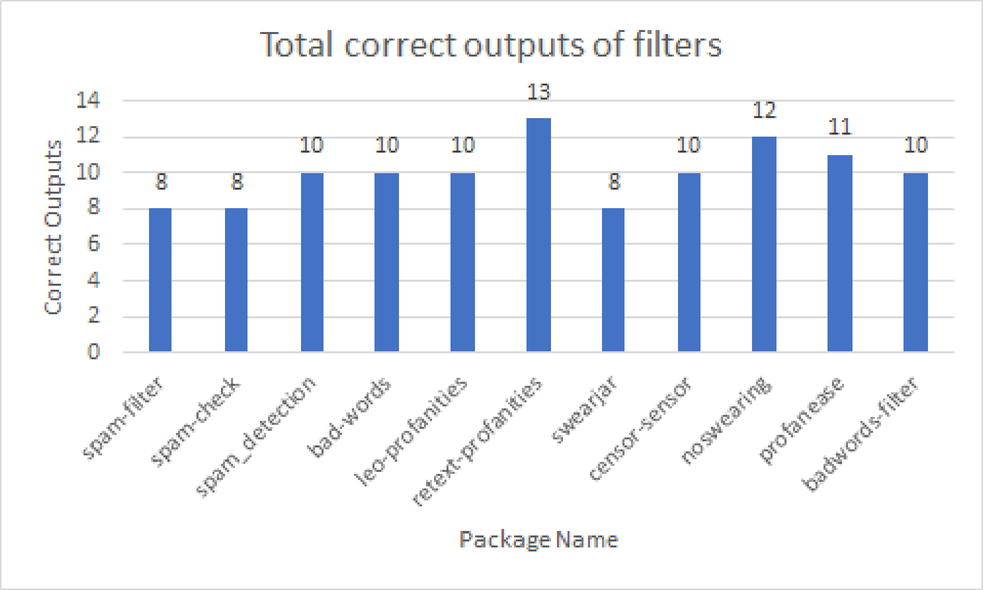

Figure 1. Total correct outputs of filters.

When we separate what type of test case it was this allows us to compare against short and long test cases. Doing this allows us to see which filter works better depending on the type of test case. When looking at short test cases, the disparity between the packages becomes more apparent. Swearjar is the worst, with retext-profanities finding the most correct outputs. When looking at long test cases, a lot find all of them and all the others just miss one.

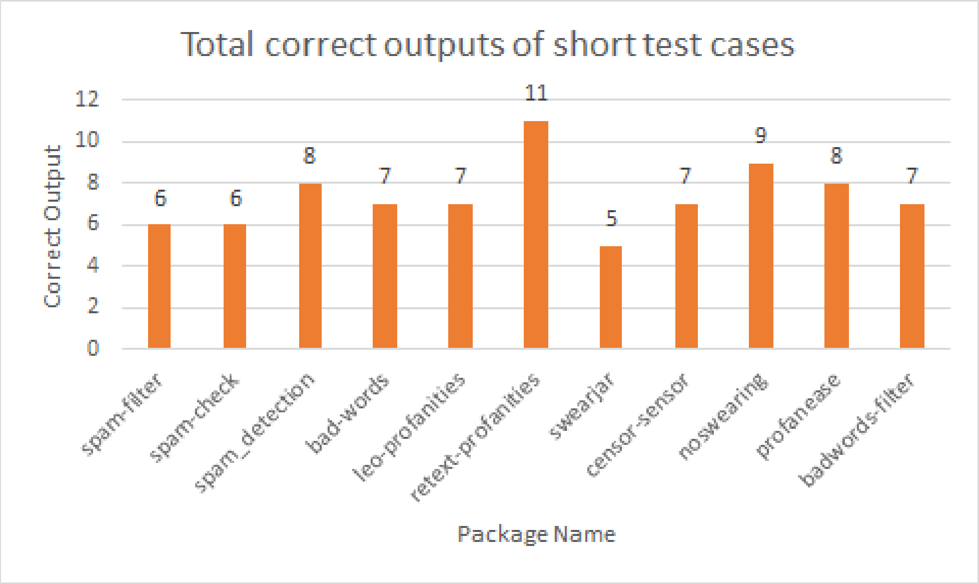

Figure 2. Total correct outputs of short test cases.

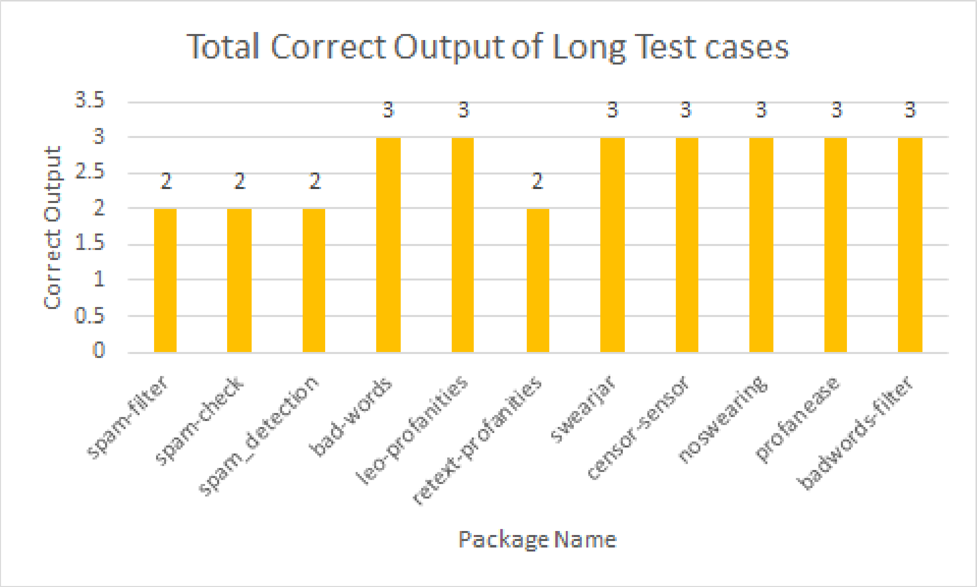

Figure 3. Total correct outputs of long test cases.

The spam filters all ran at diverse times as shown in Figure 4. Badwords-filter, censor-senor, swearjar, leo-profanities, spam-check, and spam-filter all ran at equally fast times. Any of them would be a good choice if speed is desired for a developer's use case—as long the accuracy goal is met. The slowest spam filter was profanease which was on average running at about

0.04 seconds. This spam filter would not be recommended for use cases where execution time is essential.

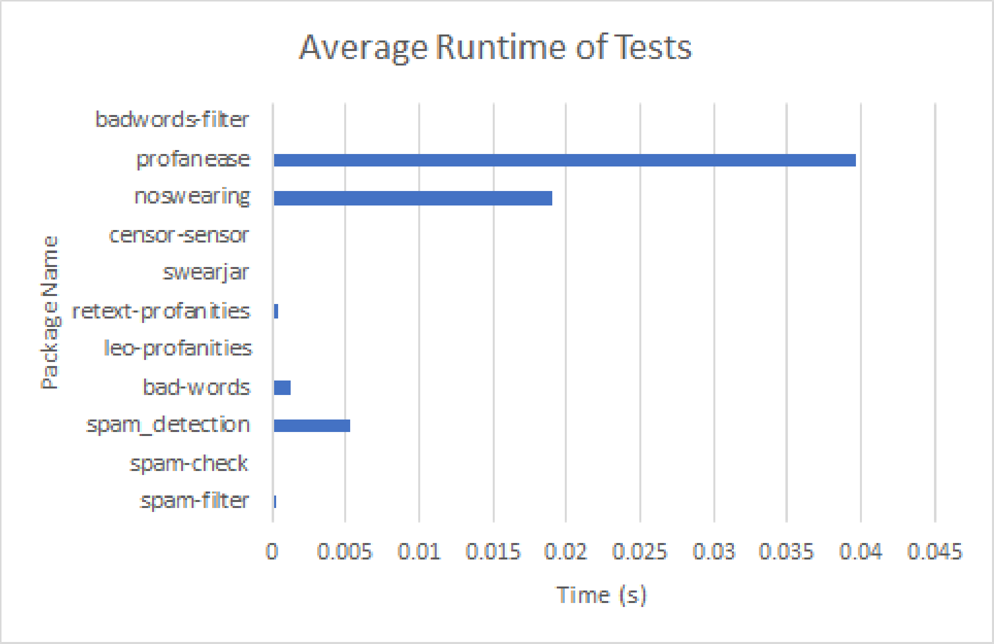

Figure 4. Average runtime of tests.

When we separate what type of test case it was we can compare against short and long test cases to see which work better depending on the type of test case. When looking at the orange chart, it looks a lot like the total average chart. Profanese is the worst, with many being very close in runtime that it was insignificant. When looking at long test cases, we can see that profanese works a lot better and isn't the worst one in the pack but rather noswearing is. The ones that are still quick are still quick in the long one too that it is insignificant to compare them.

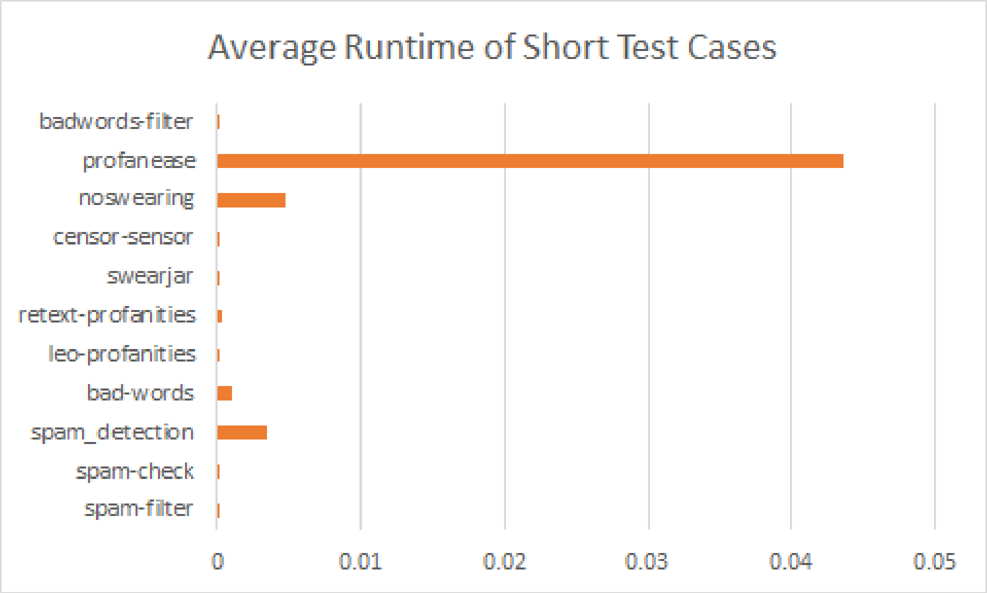

Figure 5. Average runtime of short test cases.

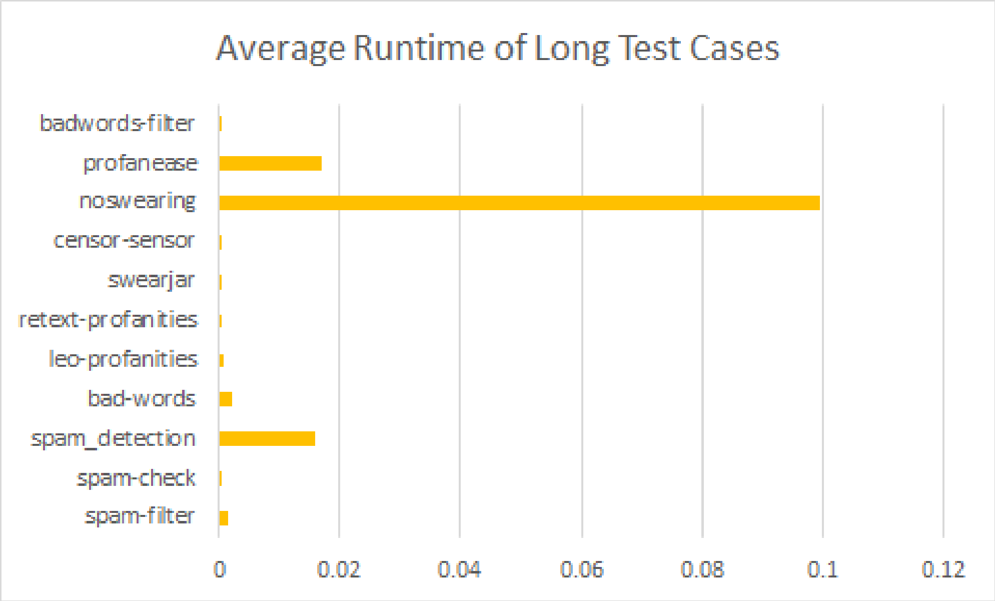

Figure 6. Average runtime of long test cases.

A lot of these spam filter packages were not that difficult to implement and get a result output. Their documentation was easy to read and had meaningful examples with example of how the output will look like. The ones that were difficult just required a little more time reading documentation except retext-profanities because it’s output was in a simple format and had to be parsed through with a regex

Table 3. Difficulty levels for each spam filter based on how hard they were to implement.

Name	**Difficulty**
spam-filter	Easy to implement.
spam-check	Easy to implement.
spam-detection	Easy to implement.
bad-words	Easy to implement.
leo-profanity	Easy to implement.
retext-profanities	Hard. Had to implement a regex to filter out what we needed.
swearjar	Easy to implement
censor-sensor	Easy to implement
badwords-filter	Easy to implement
noswearing	Slightly tricky to implement. Had to check for the length of the return due to it not returning a boolean statement
profanease	Easy to implement

Limitations

With our schedule being full time students with also focusing on our capstone project, we were quite limited on time. This limitation of time for us prevented us from developing an automated testing system for our filters. If we had more time, we would have liked to develop an automated system using machine learning that would generate test cases with the correct outcome already determined. This would allow us to further test each filter to get a better understanding of how intricate each one was. With our 20 tests we were able to get a general idea of how effective each filter was, but having more tests and having the automated system would give us enough data to precisely evaluate each filter to their fullest. Another aspect we would dive into further if we had more time would be the types of filters we test as well as the amount of filters we want to test. Having a time shortage for the project really limited how many filters we had time to research, determine if it was a good use for this study, and then implement into our system. Allowing more time for us to expand what filters we tested may have allowed us enough time to find a filter that scored against our test with flying colors. However, with the time we had allotted for this project, we were only able to implement 11 different filters with the average correctness between the filters being about 50% with the best one being 65%. Increasing the sample size of filters test could have resulted in a better average overall for our filter testing as well as shine some light on really good filters we didn't have time to implement or come across in our research.

Discussion

In the future, we could improve on several different aspects of this project. The first aspect is to automate a system that would generate test cases through machine learning. For this project, we generated the test cases ourselves to allow us to know which test cases should fail and which should pass through the filters. This gave us a fairly big limitation on test cases we were able to run through our filters. Having an automated system to intelligently generate test cases that the system would know if they pass or fail would allow us to have a significantly bigger sample size than what we had during this first implementation of our study. Another aspect we could further improve on in the future is the amount of filters we test and having more functions that combine different filters. For our study we tested 11 different filters that ranged from profanity to straight spam. Expanding the types of filters we test as well as largely increasing the amount of tested filters would be a good step in the right direction to further understand which filter/combination of filters works the best for any given situation.

We learned quite a bit from this project. Whether it was how to implement a system that could test multiple spam filters in a single program against multiple test cases, or it being how to combine two different filters to allow a single filter to cover two different types of spam. However, we think the biggest lesson we learned throughout this study is that the premade filters that are available through packages are adequate. We were not able to find a filter that was able to pass/fail all of our test cases correctly. On average our filters were correctly determining the spam by 50% with our lowest accuracy being 40% of our test cases. This result was something that was surprising to us. We thought there would be at least one or two filters that would be able to get at least 90% of our test cases correctly. However, with that not being the case for all of our filters, it’s safe to say that from the data we collected during this study that it is more efficient to develop the filter you need yourself if you are truly worried about the amount of spam that could be generated in a project. However, the 50% may be fine for a project that has less of an emphasis on catching spam. In the end, the type of filter used, if any are used at all, is completely determined by the type of project being developed. While our research in this study was a good starting point for anyone wanting to implement a spam filter, every project will need to do research to see what best fits them. Whether that be a filter package that can be imported, or a completely custom filter made by the project developers.