Report On The Unsafe Minifier
By Jacob Friesen

Introduction And Application

As websites increase in dynamic capabilities they increasingly need more dynamic scripting components which are almost always done with JavaScript (JS). The problem is the JS files need to be loaded then used, meaning the amount of source code affects how fast dynamic scripting can be applied to the client interface. One page applications like Gmail make this problem more acute as they require lots of dynamic scripting.

The solution is to remove as many characters from the source as possible while still keeping its original functionality (minification). Note that there is no explicit decompression step, the source is runnable in minified form. There are many minification algorithms to do this, but they can only do what I call *safe minifications*; Minifications that remove and alter some characters but do not change program flow e.g. function call order. My algorithm aims to do *unsafe minifications*, that is to alter the actual program flow but still keep the program functioning in the intended way. Since the domain of unsafe modifications is so large, the minification potential is enormous, especially when paired with safe minification.

Algorithm

There is a few unusual aspects to how I approached the problem, so they are worth discussing. More details can be found in my codes READMEs and program files (decently commented). Firstly, due to time constraints I only implemented one type of unsafe minification, merging two function calls and their corresponding bodies. There is many different ways for this to happen based on what functions have returns, which calls have variables assigned to and etc., but here is a simple example:

Original (120 bytes)	***Unsafe Minify (110)***	***Unsafe+Safe Minify (69)***

`function function1(x, y){`
`  x + y;`
`}`
`function function2(w, z){`
`  return w * z;`
`}`
`function1(1,2);`
`var x = function2(3, 4);`
	`function function2(x, y, w, z) {`
	`  {`
	`  x + y;`
	`  }`
	`  return w * z;`
	`}`
	`var x = function2(1, 2, 3, 4);`
		`function function2(a,b,c,d){a+b;return c*d}var x=function2(1,2,3,4);`

`Note: Just doing a safe minify results in 100 bytes, also there is more than one strategy I applied to minifications e.g. when there returns in both functions.`

Of course not every function can be minified so the challenge is to correctly identify which functions can and can't be minified. This is much harder than it sounds, there is seemingly an infinite number of ways two functions could disqualify themselves. For example, the first function call could call the second function in its body. So I have no practical way of determining all the cases in which the above minification strategy will fail. This is where the AI portion of my algorithm works, it determines from statistics on the 2 function calls and their bodies if they can be minified.

How the AI Works

Mainly due to problems with my data the AI strategies employed are also somewhat unique. The main problem is I have many potential cases for when the merging fails but a small amount (40) for when they succeed. So I can only use 80 cases total.

1. Data

My data is generated from analyzing a set of source files (in data/raw_data). Specifically, each files source is analyzed by putting the code into an Abstract Syntax Tree (AST) which is a representation of the code structure (usually used by JS interpreters). Using this method, most of the functions declarations and calls are then found. Then the calls that are close together are merged along with their function declarations, statistics are recorded on these functions.

Finally, the validity of the merges needs to be verified, but I could find no automated way of doing this. The number of ways of testing JavaScript is very large because a lot of libraries implement custom testing instead of using a testing library. This means I would probably need to develop a unique verification system for style of testing in my initial set of source files. As such I was forced to look at each of the 119 merges and see if they were valid, which limited the total amount of data that I could use.

There may be a possible solution to this using clustering, but it would require a very thorough analysis on what attributes separate functions that can and can't be merged. Currently, I am able to get by with a comparatively basic one.

2. Machine Learning

Due to the small amount of data, overfitting was a big deal. Firstly, for simplicity I chose to rely on a standard fully connection, backpropagating three layer network. Using parameters:

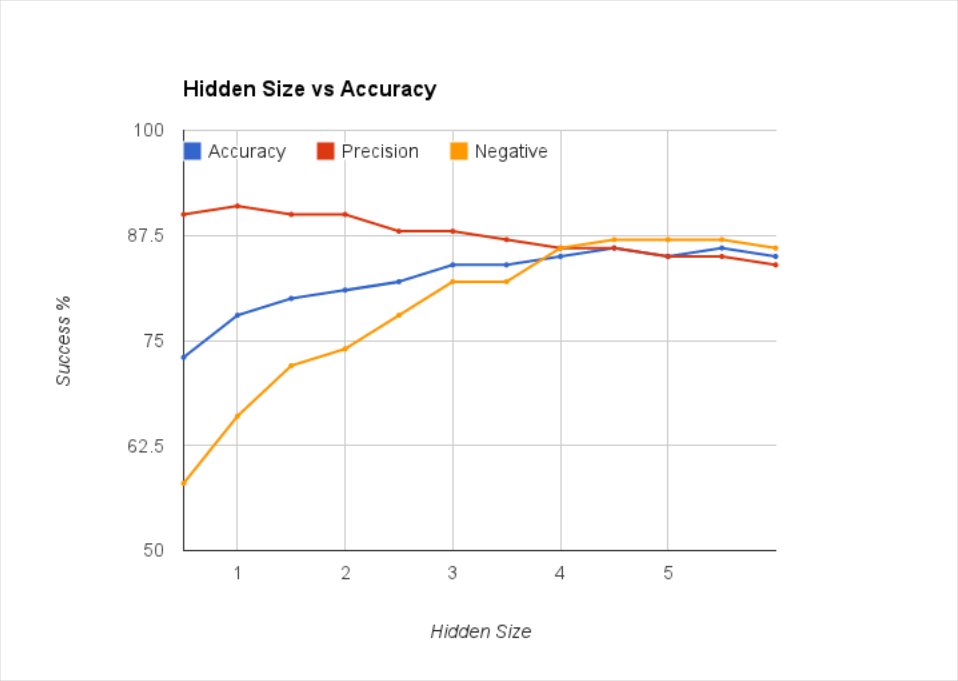
```
hidden layer: 2x input layer size
desired error: 0.1
output threshold: 0.5 (less than this is negative)
backpropagation limit: 10
```

I was able to achieve a 82/89/75% accuracy/precision/negative success rate over 1000 creations. Note negative = TN/(TN + FN). As backpropagation limits got higher accuracy slowly decreased, hence why I have such a low number. This makes sense because the network is so small. The traits I used were:

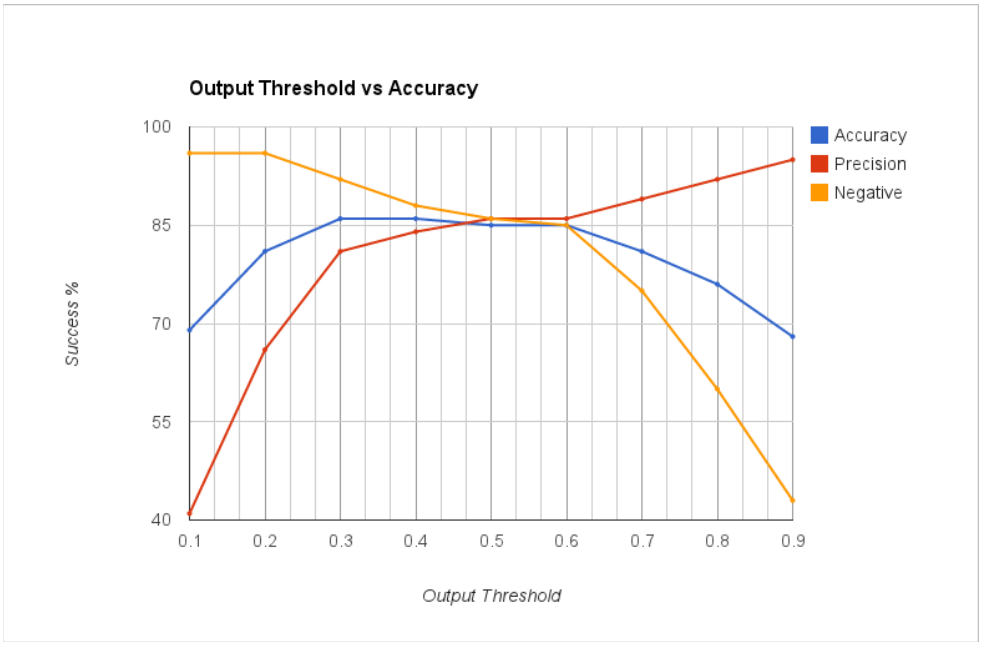
- First and last characters of both merging calls (4 total)
- Argument number of both merging calls (2)
- Parameter number of both merging functions (2)
- Line of both merging calls (2)

I could not use many more because I have such a small amount of data points. Also, I made an attempt to ensure function size would not have a direct effect. If I had more data I would find

some way to read in all of a functions contents, and numerize those.
Interestingly, with this small data set a very large hidden layer helped with accuracy. I adjusted it to 4 (average over 1000 creations):



Finally, the penalty of guessing wrong on a negative means an incorrect minification will happen. So a user will have to detect something has went wrong and either retrain and reminify or manually repair the error. Incorrectly guessing a positive (precision) means no minification, but it also needs no user corrections. As such, it makes the most sense to optimize negative success rates over positive. The easiest way to do is by adjusting the error threshold. Using the previously set parameters I got the following, so I chose 0.3:



3. Merging

Once the system is trained a user can send a file to be minified. First the file is minified via my algorithms and saved. Then that file and the original are both minified by the traditional minifier and saved. This way you can see the total result of minifying the file with and without my algorithms.

From before individual networks still varied from 75 - 96% accuracy, and even more with the individual precision/negative success rates. So I decided to separately train five networks and use the consensus of the results to decide if I should minify at this step. There is about a 12% gap in accuracy/precision and negative with this.

Conclusion

Although the training evaluation section displays good results, minification on non trivial files outside training seems to end up with only a 67% correctness. When minifying a file there is about a 4% decrease in size on average over just using the safe minifier, although the average varies heavily based on the coding style of the source file. Optimally, it will be about a 30% reduction.

Considering the small amount of data I had to work with these results seem promising, but not good enough for a real application. I think most of the problems could be solved with adding a lot more data (like 20x more), but that requires solving the automaticity problem with valid examples. Finally, on a more constrained type of minification even with this small amount of data I think a much higher accuracy could be achieved.