Analyzing JavaScript in Modern Webpages Using Machine Learning

Vladyslav Cherevkov

Capstone

Advisors: Yasir Zaki, Moumena Chaqfeh

Abstract

Javascript is the most popular web-based programming language and is also unfortunately a major reason why the performance of modern web pages has been deteriorating. My main objective is to create the most accurate and robust machine learning classifier for third party JS scripts that will be deployable on a client system that does not have many available computational as well as disk resources. Previous research suggests that tree-based machine learning algorithms perform better in third party JS script classification. Therefore, I focus on these algorithms and, in particular, on performance differences between top bagging and boosting ensemble classifiers. The abundance of JS scripts used for tracking, advertising, beautification, and unnecessarily heavy content in the form of images or videos, put a heavy burden on users that lack either computational or internet data resources. By accurately identifying such scripts, the classifier can later on be implemented in a web-simplification focused piece software or a browser plug-in and provide users the ability to disable various JS scripts based on the label provided by the classifier. I trained and compared the performance of three tree-based classifiers: Decision Tree, Random Forest, and XG Boost. Random Forest turned out to be the best performer with a macro-average 83.97 F1 score on a normal feature set and a slightly lower F1 score of 83.32 on the reduced feature set.

1 Introduction

Even though the Internet is being increasingly accessed more with mobile devices such as smartphones and tablets, most of the development for the web is done nowadays with the powerful PC in mind. With the exception of the main players such as Google, Facebook, and Amazon, most websites prioritize looks over functionality. Such websites are filled with unnecessarily heavy structure: cluttered with advertising, tracking, and beautifying JS scripts instead of putting efficiency and proper resource allocation as the primary goal.

Meanwhile, web pages have undeniably become the dominant tool through which users satisfy all their information, entertainment, and social needs online. Therefore, the end-to-end Web page load time has a direct on the user experience as well as on stream of revenue for these websites. This claim is supported by recent research conducted by Google in 2017, which showed that more than 50 percent of users will leave a mobile page if the load time exceeds 3 seconds. [1]

Unfortunately, even with the increasing rise of technology with and the latest hardware both on the client-side and the user side, page load times for mobile connections are still far from desirable. The same research done by Google conducted that the average load time of a web page on mobile in 2017 was 22 seconds. [1]

Loading a website rarely poses an issue when a user tries to access such a website from a powerful computer at home with a strong Internet connection. Great bandwidth allows the client to retrieve the HTML, JavaScript, Videos, Images, and other web elements without a problem. The rise of PC and rapid improvements in internet speeds, gave Web Developers the green light to maximize the visual appeal of a page through using unnecessary graphics and other beautification scripts.

Javascript is the most popular web-based programming language and a predominant force that is defining the direction in which current webspace is going. In fact, it is used to build 95.2% of the web at the moment. [2] The problem, however, is that Java Script with its recursive calls and heavy structure is often paired up with implementation by inexperienced web developers. These two factors combined result in a slow poorly-optimized service to serve mobile users, especially those with weak and slow internet access which is the case for many developing communities.

Moreover, the abundance of tracking and advertising JS scripts that web services use to monetize their services and gather the maximum amount of data on their users results in further clutter and speed reduction. However, these scripts usually offer little to no value to the average user who is purely interested in the content of a web page they are visiting. Therefore, clients' mobile data and computational resources that are required to load such scripts are oftentimes wasted and become especially difficult of a burden on those who visit such websites from cheap mobile phones with little computational power and poor internet access.

2 Specific Aims:

To help solve this problem of excessive use of computational and mobile resources for these users, my contribution is to develop the best tree-based Machine Learning classifier that will be able to accurately and reliably identify JS scripts. This classifier can then be included in any kind of application or web-extension that benefits from knowing which category any specific script belongs to.

The main deployment and implementation goal is for the classifier to be compatible with the Firefox plugin that my teammate Manesha Ramesh worked on and a web-simplification software that my other teammate Jacinta Hu developed. Therefore, it is extremely important that the classifier in its trained form is not just accurate and robust, but also requires a minimum amount of computational resources as well as takes up little disk and ram space. These factors are extremely important for it to be easily deployable on any kind of client application.

Background:

In terms of the general idea of building a web simplification system, there has been intensive research done by researchers from renowned universities worldwide such as Stanford, MIT, UC Berkeley, and industry leaders such as Google and Facebook. [3] We have already covered close to ten state-of-the-art papers on these issues and identified that while significant progress to combat this problem has been made, there is still a lot to do. Most leading researches we covered focused on commercial applications to the technology such as Google's Flywheel initiative [3] that was built into mobile Google Chrome to improve its performance and Shandian, another proxy-based solution that has been commercialized and turned into a company.[4] Other covered research focused on the latest browser optimizations (Brave's SpeedReader) [5], web page load performance (WProf)[6], DOM Tree structure and increasing page load time (Polaris).[7] Finally, several solutions for local edge cache implementation were discussed based on Professor Yasir Zaki's xCashe. [8]

In terms of classifying JS scripts using machine learning techniques, most of the previous research focused on the detection of malicious JS scripts as well as identifying specifically advertising and tracking scripts. These can be seen in [9], [10], and [11]. The first research team that decided to focus on classifying JS scripts based on all the available JS categories was the JS Model research team [12]. They ran several classification algorithms and third-party JS scripts. Among them are Support Vector Machine, K-nearest neighbors, Linear Support Vector Machine, and Random Forest. The team found that Random Forest was the best performer out of all those algorithms. [12]

3 Goals And Potential Impact

The implications of this research are vast since an accurate and robust JS script classifier can have a huge impact on which content is loaded by the end-user and greatly reduce the number of computational and internet resources that are needed to load a desired piece of content on the web.

Removing these scripts will lead to huge potential benefits to users in developing communities where access to the Internet and availability of mobile data plays a role that is far more important than simply checking ones Instagram or Facebook. Instead, it may determine someone's daily or even weekly income as is in the case, for instance, for many farming communities in Ghana.

4 Methodology

4.1. Data Set

For the dataset to be used for training Machine Learning classifiers, I used a dataset that was generated by the team that wrote the JS Model paper [12]

This dataset consists of features that were extracted from third party JS scripts retrieved from 20 thousand popular web pages. Afterward, the JS Model team used a Web Almanac repository that contains a set of domain names accompanied by their categories to label the dataset. I chose this particular dataset because at the time of writing of this paper, it is the most comprehensive dataset that includes a complete list of 12 general JS categories: Advertising, Analytics, Social, Video, Utilities, Hosting, Marketing, Customer Success, Content, CDN, Tag Management, Other. [12]

The explanation of each category is provided below, as per the JS Model paper:

1. Advertising - refers to domains associated with advertising efforts such as Google Ads service.
2. Analytics - encompasses JS scripts which provide tools to keep track of the web service's users
3. Social - refers to JS scripts that provide social features in websites.
4. Video - refers to resources used to provide various video services such as streaming
5. Utilities - covers developer tools, such as API clients and fraud detecting
6. Hosting - refers to JS scripts used in web hosting, for instance those utilized by WordPress and Squarespace.
7. Marketing - refers to JS scripts that are related to marketing tools.
8. Customer Success - relates to customer support tools that provide contact solutions.
9. Content - refers to publishing-specific affiliate tracking.
10. CDN: which is a mixture of publicly hosted open source libraries (e.g. jQuery) served over different public CDNs and private CDN usage.
11. Tag Management - refers to JS scripts that are likely to load JS elements.
12. Others - refers to JS scripts that do not fall in any of these categories mentioned above [12]

The JS Model team performed Class assignment to the JS scripts they scraped by comparing the domain name of the script's source URL to the URLs that are contained in the Web Almanac dataset. By matching those URLs, the team appended one of the 12 categories to each script, therefore, providing a class. [12]

Once I gained access to this data set, I did some exploratory analysis using python and specifically the Pandas and Matplotlib libraries for data manipulation and visualization. I found out that the length of the data set (number of rows) was 117,239 entries, which is more than sufficient to run various machine learning algorithms but by no means goes in the realm of big data that would require significant computational power (dedicated server with a powerful CPU, GPU, and lots of RAM).

However, what could have made it potentially inconveniently difficult to work with this data set was its width. The initial data set that I acquired from the JS Model team had 2656 columns, out of which 2653 could be used as features for the classification. The other three were almanac labels, Script Index, and URL. URL column is the domain URL of the given JS script, Scripts Index is just a unique identifier of each script within this data set, and almanac labels is the actual label column that will be used for training and testing of each algorithm.

After the initial basic exploratory analysis, I moved on to Feature Selection. This is a highly important part of the training process because it will define the amount of time it takes to train each algorithm as well as the size of the Machine Learning classifier when it will be exported and deployed. Therefore, the goal is to have as few features as possible while maintaining high accuracy measured by f1 score.

4.2. Data Preprocessing

The next crucial step was to analyze all the available features from the correlation angle. If a given dataset has features that have a high positive or negative correlation coefficient (commonly referred to as r), it may have potential negative effects on the accuracy and performance of the classifier. One such negative effect is that the model can be affected by Multicollinearity. Multicollinearity is something that occurs in multiple regression models when one feature can be linearly predicted by another (or a set of others). That has a high potential to bias the predictor or provide misleading information. Some ML algorithms are especially susceptible to this problem, among them Logistic Regression and Linear Regression while others such as decision trees or boosting tree algorithms are not affected. However, even in that case removing correlated features is useful as such features will provide the same predictive information to the algorithm therefore using up computational resources while not improving the accuracy or robustness of its prediction. [13]

Therefore, my next step was to find all the features that have a high positive or negative correlation coefficient and remove them from the feature space. In order to speed up the process, I built a correlation matrix using python's numpy library for efficiency and removed features that have a correlation coefficient higher than 0.8 (absolute value) with any other feature (or sets of features). This made a significant difference because most of the features turned out to be highly correlated. To be exact – 2145 features had their correlation coefficient higher than 0.8 with at least one other feature. After this procedure the feature space was reduced to 508 columns, which could be used to run multiple classification algorithms. I also needed to do some cat feature encoding in order for the algorithms to function properly.

4.2. ML Classification

The next steps consist of finding the most accurate and robust tree-based classification algorithm by running several and comparing the results. Afterward, once the best algorithm is selected, I will perform further reduction of the feature space using Recursive Feature Elimination with Cross Validation in order to optimize the disk and computational resources that it takes while minimizing the decrease in accuracy. This will ensure that the classifier can be implemented in various client systems that do not have the computational resources that most servers possess. Therefore, this will expand its uses and distinguish it from the majority of other classifiers that have been researched with server implementation in mind.

At the most basic level, the problem I was facing was a multiclass classification because I was trying to predict a discrete variable (JS label) that can be any one of the 12 categories (advertisements, analytics, etc...) based on a set of other variables (features). For this type of problem I decided to focus on tree-based machine learning classification algorithms because they perform well when it comes to multiclass classification paired with a large number of discrete and continuous features. When looking at the machine learning algorithms that the JS Model team utilized, it confirms this statement because the best performer was the only tree-based algorithm – Multiclass Random Forest that had 0.8 Macro-Average F1 score [12]. The others, which were K-Nearest Neighbors, Support Vector Machine, and Linear Support Vector Machine performed worse with macro-average F1 scores of 0.6, 0.78, and 0.76 respectively [12]. Thus, my approach will be to first use the most fundamental of all tree-based algorithms – decision tree. Next, I will investigate how some of the improved ensemble learning techniques that the machine learning community developed for it (in particular bagging and boosting) improve the performance.

The main metric for the evaluation of these algorithms was the f1 score which is an extremely important metric that is used by most researchers and data scientists. This score is computed by a formula F1 = 2 * (Precision * Recall) / (Precision + Recall). Within this formula, precision refers to how accurate the classifier is by concentrating on how many values that were marked positive are actually true positives. Precision is a good metric in problems where the cost of a false positive is high, such as email spam detection. On the other hand, recall captures how many of the true positives are actually marked correctly. In other words, recall is used in problems when the cost of a False Negative is significant. Therefore, since F1 score takes both of them into account it is a metric that is generalizable to a greater number of problems and can be used as a key metric in this paper as well. [14] Also, for the sake of brevity every time that I mention F1 score in this paper, it refers to macro-average f1 score unless stated otherwise.

First of all, I ran the most basic of all the tree-based algorithms – decision tree. Its main two advantages is that it is suited for multi-class classification and that is that it is extremely easy to visualize and interpret the results when compared to its counterparts. What can be inferred from the name is that a decision tree algorithm tries to solve the issue, by using a tree representation. Every internal node of the structure represents an attribute (feature) of the data set, each leaf node represents a class label, and finally each node is connected to its parent based on the rule that the algorithm learned during the training process. This algorithm classifies the observations by sorting them down the tree starting at the root and ending at some leaf/terminal node. Each of the decision tree nodes could be looked at as a test case for some specific feature. Similarly, every edge that spans out from that parent node represents the possible answers to that test case. This process is repeated recursively at every node for the sub-tree that will be built under it. The information gain approach is utilized as a criterion when estimating the information contained by each attribute. [15] (Figure 1)

I used Google's sklearn library which is widely used for decision tree implementation in python. Using this same library I performed data split into two datasets – one for training and one for testing. The first dataset that was used for training the model contained 75 percent of all the observations (rows) and the remaining 25 percent was used for testing in order to eliminate the in-sample bias. It was crucial to perform out of sample evaluation on this testing dataset because the classifier needed to be evaluated on the data it has not been exposed to earlier. Therefore, after training the decision tree classifier on the training set I evaluated it on the testing set. The F1 score was high at 83.26 which further reinforces the initial hypothesis that tree-based algorithms have a high predictability potential for this specific multiclass classification problem.

The next iteration of this classifier selection process involved running and examining the performance of the Random Forest algorithm, which is the most robust implementation of the bagging method, and comparing it to the performance of XG boost which is the most popular algorithm when it comes to boosting. Comparing these two will be interesting, because Random Forest was the algorithm that performed best in previous research made by JS Model, while XG Boost made a major name for itself in Keggle competitions oftentimes beating Random Forest in multi-class classification problems. However, we must also keep in mind as well that an F1 score of 83.26 that the decision tree classifier provided is already very high. This is why any marginal improvement in this score should be treated as a success.

Before getting into the details of how the Random Forest algorithm works, it is important to clarify what ensemble algorithms are. Ensemble machine learning technique refers to algorithms that combine predictions from several classifiers (or regressors) in order to create a more accurate result.

The random forest algorithm is one of the most popular ensemble machine learning techniques that utilizes the method mentioned above called bagging or Boosted Aggregation. Random Forest is commonly used for both classification and regression problems and the same variation of the algorithm can be used to solve both issues. In conceptual terms, this machine learning algorithm creates a forest that consists of decision trees. (Figure 2) Higher numbers of these trees usually suggests that the algorithm will provide more accurate results. However, there are usually very marginal improvements after a certain threshold when its accuracy starts approaching an asymptotic value depending on the data set and problem. In most of the other classification machine learning algorithms increasing the accuracy of prediction usually comes with a necessary cost of increasing the bias. Yet, adding more trees to the Random Forest does not result in introducing bias or over-fitting, because of how the algorithm is implemented through the bagging technique. The algorithm creates and runs many decision trees simultaneously with subsets of features selected at random. It chooses a label for the specific row to be the label that was selected by the greatest number of trees. The default number of trees is set to 100; however, I used 220 trees for my Random Forest classifier. I settled on that value because I saw marginal increasing F1 and accuracy scores when going from 100 to 150 and later to 200 trees. However, results did not improve by increasing the number of trees above 200. Therefore, a random forest classifier with 220 trees was able to improve the F1 score to 83.97.

The final algorithm I examined was XG Boost which stands for extreme gradient boosting and is considered the most advanced technique within tree boosting algorithms. The easiest way to understand XG Boost is to look at it from a point of view of incremental improvement to the basic decision tree algorithm. To make the first step into the boosting world from a regular decision tree we must look at the most basic of these algorithms – the boosted tree. It operates under the principle that in order to minimize the prediction error, we must select the next iteration of the model to be better than all the previous ones. Under this principle, if we continuously select the iteration of the model that is better and combine it with the previous ones, this will provide the best result. This is handled by machine learning libraries implicitly by having a cost function that is constantly recomputed to see if the next model is making results better or worse. The next iteration of the boosting tree algorithms takes us another step closer to XG Boost. This step is called gradient boosted trees. They operate the same way as regular boosted trees with the only exception that they use gradient descent as the algorithm for handling their cost function which has shown to provide superior results. Finally, XG boost builds on these two algorithms by including numerous benefits such as parallelized tree building, in-built cross-validation capability, tree pruning depth-first approach, easy cloud integration, and many more that make it a much more robust and best boosting algorithm. [16]

Unlike random forest, there are actually several parameters that can be tweaked in XG boost.

These are max depth, learning rate, objective, num round, and num class. The num class and objective are defined by the kind of problem that the algorithm is used to solve. Since we are facing a multiclass classification with 12 distinct classes, num class should be set to 12 and objective to multi:softmax. I tweaked the other three parameters – max depth, num round and learning rate to find the optimal solution. Max depth refers to the maximum depth of the trees that are being boosted, num round defines the number of boosting iterations, and learning rate defines the extent to which the step size should of the boosting algorithm be shrunk (range is from 0 to 1). This is used to prevent over-fitting, making the boosting process more conservative. The industry standard is to keep learning rate between 0.1 and 0.3.

After tweaking all of these parameters, the combination which provided the maximum F1 score for the XGBoost was: max depth = 32, learning rate = 0.3, objective = multi:softmax, num class = 12, and num round = 112. The F1 score which XGBoost was able to provide given these parameters was 83.85, which is, in fact, lower than that of Random Forest discussed above. Moreover, XGBoost took around 5-6 times longer to be trained when compared to Random Forest. Therefore, even though this algorithm has proven to be very powerful in different scenarios, it is evident that for this specific problem and dataset, Random Forest is a better performer.

4.2. Further Feature Reduction Using Recursive Feature Elimination

After realizing that the classifier was Random Forest with the F1 score of 83.97 trained on all the 508 uncorrelated features, it was time to reduce the feature space further in order for the algorithm to be feasible to deploy on the client side. For this process, I used an advanced feature reduction technique called Recursive Feature Elimination with Cross Validation (RFECV). It is a powerful technique that helps to figure out the most predictive features. RFECV, in a way, is a backward selection of predictors. It starts by building a model using all of the features and computing the importance score for each one of them. The least important feature(s) are then removed. Afterward, RFECV re-builts the model and calculates the importance scores again. Once the optimal set of features is found, it is then used to train the final model. For instance, in Figure 1, where the accuracy of classification is plotted on the vertical axis and the number of features selected on the horizontal, we can see that not all of the 508 uncorrelated features we used provided the same value. In fact, it is clear that only the top 50 accounted for most of the predictability that the model had. (See figure 3)

RFECV needs a machine learning algorithm to be specified as one of its parameters to do the job. This is why we needed to choose the best classifier before running this process. Since I found out that Random Forest with 220 trees performed the best on the 508 features, I also used it for RFECV. One disadvantage of Recursive Feature Elimination is that it is very computationally difficult to run; however, due to the nature of the task, it only needs to run once.

After using this technique I had a list of features sorted according to how predictive they are. This allowed me to take the top 50 (since the graph clearly showed that these are the most important ones) (Figure 3) and reduce the feature space to just those. This provided a major improvement. I was able to reduce the feature space by one order of magnitude while sacrificing less than one percent of accuracy and f1 score. The f1 score for Random Forest algorithm on this new small feature space of only 50 attributes is 83.32.

Results

This paper found that Random Forest was the best tree-based algorithm to accurately classify third party JS scripts into the 12 known categories. Random Forest classifier provided the highest F1 score of 83.97 on the full set of 508 uncorrelated features and only this F1 value slightly decreased to 83.32 when reducing the feature space to 50 most predictive features. Therefore, it satisfies all the goals that were put in front of the desired classifier – the highest possible accuracy as well as being conservative in terms of computational and disk resources it needs to operate.

The decision tree classifier showed promise with an objectively high (yet still the lowest of the three) F1 score of 83.26. However, since random forest and XG boost are two great ensemble learning techniques that incorporate the decision tree into their learning process, both unsurprisingly performed better. XG boost performed slightly worse than the Random Forest receiving the F1 score of 83.86; however, it also took much longer to be trained and used up a greater amount of disk resources.

The resulting final Random Forest classifier was trained on the 50 features and had an F1 score of 83.32. This classifier was, in fact, successfuly deployed as a part of software called JSAnalyze developed by Jacinta Hu as well as a Firefox Plugin developed by Manisha Ramesh. Both of these implementations give users the ability to disable JS scripts based on the label they are classified as.

Even though this paper reinforced the claim which was previously made by the JS Model team – that Random Forest is the best algorithm so far for classifying JS scripts, it did so by looking at a different set of classification algorithms. It focused on the tree-based algorithms and in particular on the performance differences between two best tree-based ensemble classifiers: Random Forest and XG Boost. It showed that XG Boost does not provide superior performance in this particular problem in terms of accuracy as well as computational and disk resources use. Finally, I improved the deploy-ability of the Random Forest classifier through successfully implementing Recursive Feature Elimination with Cross Validation. This resulted in reducing the feature space by an order of magnitude while sacrificing less than one percent accuracy.

Figures

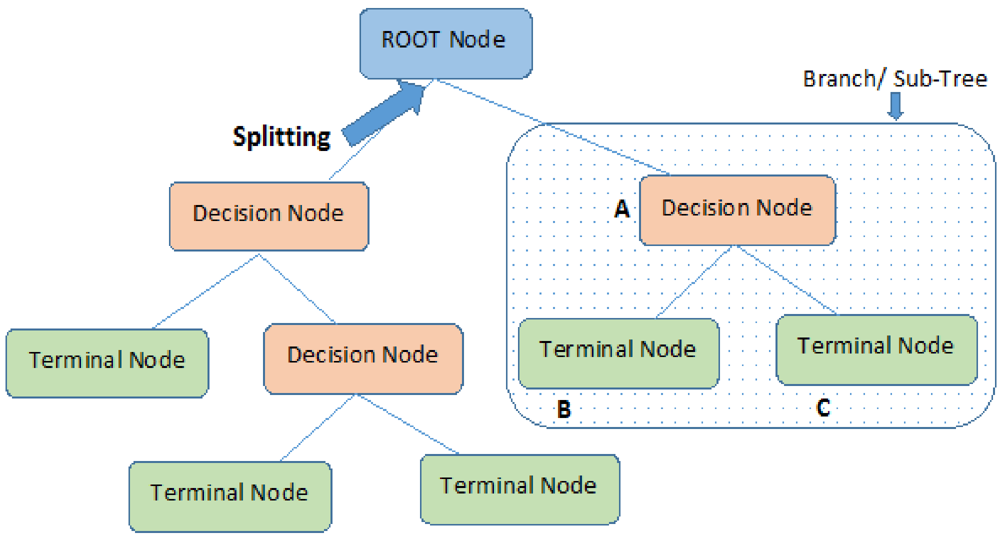

Figure 1: Decision Tree Conceptual Visualisation
[17]

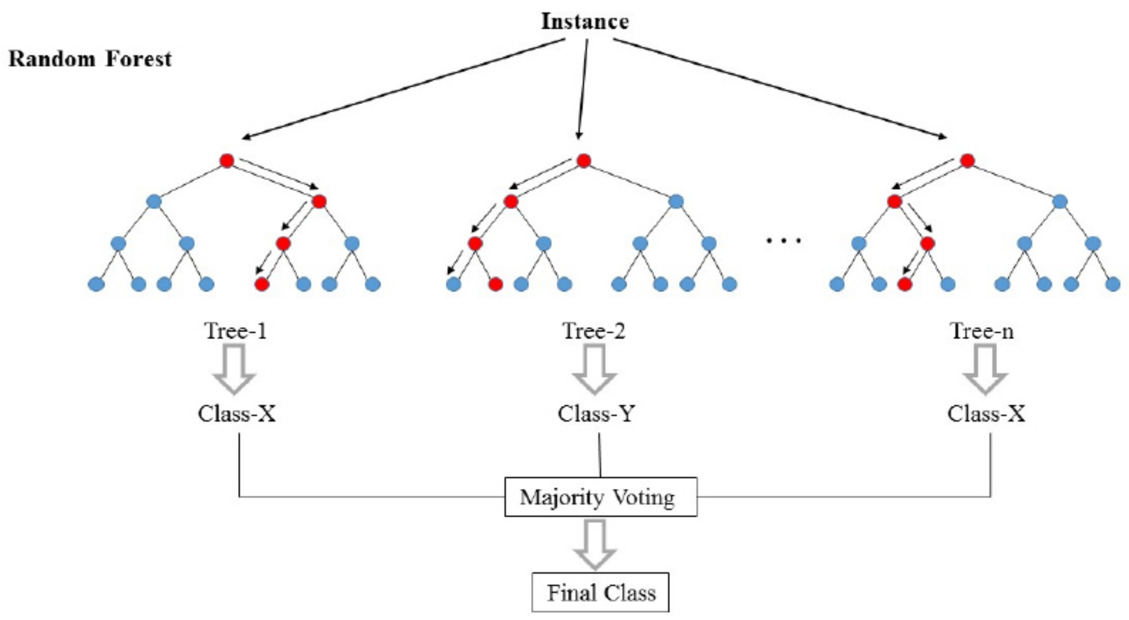

Figure 2: Random Forest Conceptual Visualisation
[18]

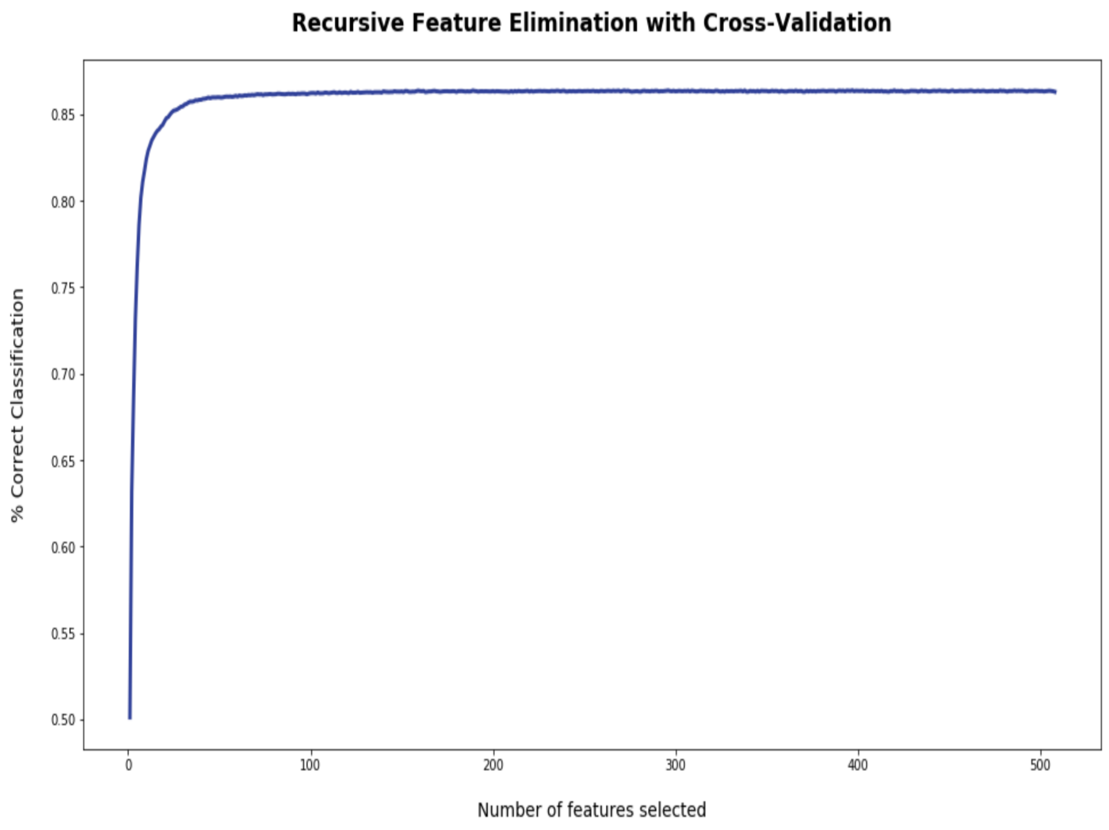

Figure 3: Recursive Feature Elimination

References Cited

[1] "Think with google: Find out how you stack up to new industry benchmarks for mobile page speed." [Online]. Available: https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf

[2] T. DeGroat, "The history of javascript: Everything you need to know," Aug 2019. [Online]. Available: https://www.springboard.com/blog/history-of-javascript/

[3] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin, "Flywheel: Google's data compression proxy for the mobile web," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 367–380.

[4] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with shandian," *USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi16/nsdi16-paper-wang-xiao-sophia.pdf

[5] M. Ghasemisharif, P. Snyder, A. Aucinas, and B. Livshits, "Speedreader: Reader mode made fast and private." [Online]. Available: https://arxiv.org/abs/1811.03661

[6] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with wprof," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 473–485.

[7] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster page loads using fine-grained dependency tracking," *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. [Online]. Available: https://www.usenix.org/node/194917

[8] A. Raza, Y. Zaki, T. Pötsch, J. Chen, and L. Subramanian, "xcache: Rethinking edge caching for developing regions," *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development - ICTD 17*, 2017.

[9] X. He, L. Xu, and C. Cha, "Malicious javascript code detection based on hybrid analysis," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 365–374.

[10] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *Proc. of IEEE Symposium on Security and Privacy*, 2020.

[11] A. J. Kaizer and M. Gupta, "Towards automatic identification of javascript-oriented machine-based tracking," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, 2016, pp. 33–40.

[12] Anonymous, "Jsmodel: Analyzing and categorizing javascript in today's web using machine learning," 2018. [Online]. Available: https://doi.org/10.1145/1122445.1122456

[13] W. Badr, "Why feature correlation matters .... a lot!" Jan 2019. [Online]. Available: https://towardsdatascience.com/why-feature-correlation-matters-a-lot-847e8ba439c4

[14] K. P. Shung, “Accuracy, precision, recall or f1?” Apr 2020. [Online]. Available: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

[15] N. S. Chauhan, “Decision tree algorithm-explained,” Jan 2020. [Online]. Available: https://towardsdatascience.com/decision-tree-algorithm-explained-83beb6e78ef4

[16] J. Brownlee, “A gentle introduction to xgboost for applied machine learning,” Apr 2020. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

[17] Arun Mohan, “Decision tree algorithm with hands on example,” 2019, [Online; accessed May 1, 2019]. [Online]. Available: https://cdn-images-1.medium.com/max/1600/1*MCn6_qX_KYNwHaTSOqqbSA.png

[18] Stavros I Dimitriadis, Dimitris Liparas, “How random is the random forest? random forest algorithm on the service of structural imaging biomarkers for alzheimer’s disease: from alzheimer’s disease neuroimaging initiative (adni) database,” 2018, [Online; accessed April 27, 2019]. [Online]. Available: http://www.nrronline.org/viewimage.asp?img=NeuralRegenRes_2018_13_6_962_233433_f2.jpg