Building a User Interface for a Temporal Machine Learning System

George Yiu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-156
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-156.html

May 27, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Building a User Interface for a Temporal Machine Learning System

by George Yiu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Anthony D. Joseph
Research Advisor

(Date)

* * * * * * *

Professor J. D. Tygar
Second Reader

(Date)

Building a User Interface for a Temporal Machine Learning System

George Yiu

May 27, 2015

Contents

1 Abstract

Machine learning results are often difficult to understand and draw insight from. While building a general machine learning pipeline, we design a user interface to help users of the system follow the machine learning process. This interface allows users to monitor the status of pipeline jobs as well as interactively explore results by inspecting individual samples and visualizing the impact of each feature that contributes to a sample's final classification.

2 Introduction

The world of malware is a complex environment for both average consumers and anti-virus vendors. With anti-virus vendors receiving as many as 300,000 samples daily [5], it is imperative to design systems to quickly detect and prevent infections so that end-users can stop this behavior before irreparable damage is done. A successful system must be able to adapt quickly to the constantly changing malware landscape. Building such a system then requires two important pieces: a core software component that can detect malware effectively and some feedback mechanism for humans to understand where the system is working well and where it needs improvement. The need for the first part is obvious; we need to detect malware. We can think of the second part as serving two different purposes:

1. For developers of the system, this mechanism provides a way to get feedback from the system, which helps them verify things are running smoothly and that the results make sense. Often, a visual interface is ideal for this case.
2. For users of the system (ie. a domain-expert), this mechanism provides an easy way to run jobs over datasets and analyze the results without specific knowledge on how every component of the system was built.

In this paper, we will first briefly go over the design of the entire malware pipeline. Then we will walk through the history of the user interface that was built first to display simple results but then upgraded quickly as different users joined the project and different needs arose. The focus will then be on the high level design of the current web user interface for our pipeline and the architecture built to support it.

3 Pipeline Overview

Machine learning with big data is often difficult because it is hard to find a system that can handle and effectively process large amounts of data. One goal of our project is to design a generic machine learning pipeline that can handle any general binary classification problem with minimal configuration. By designing the pipeline around Apache Spark, an in-memory distributed computing framework [9], we hope to build a fast and scalable system that can handle any kind of dataset. While we focus primarily on a malware classification system throughout this paper, the pipeline is designed to handle any other similar classification problem with minimal extra work. Since the bulk of the pipeline was designed and implemented by Brad Miller and Alex Kantchelian, both also from UC Berkeley, I will emphasize only the parts of the pipeline that I worked on directly while summarizing the rest. An overview of the pipeline is necessary in understanding the design of the web user interface, the focus of my work. Here is a summary of the overall pipeline architecture.

3.1 Data

Our dataset consists of 1.1 million instances of malicious and benign x86 executables at a roughly 8:1 malicious to benign ratio. Every executable is identified by the hash (MD5 or SHA256) over its binary data. Our dataset was obtained from VirusTotal, an online service that scans user-uploaded executables with a suite of tools from major anti-virus vendors [8]. Each hash is associated with one or more scan events corresponding to each time the file is uploaded to VirusTotal. Each scan event provides us with two things: a binary vector consisting of malicious/benign classifications from all vendors with software on VirusTotal and a JSON of static and dynamic analysis of the executable. Since anti-virus vendors change their definitions over time, classification vectors from different scan events for the same hash are often different. On the other hand, static and dynamic analysis typically remains the same, so we only keep one copy of the JSON for each hash. Our dataset contains roughly 5 million scans (binary classification vectors) of 1.1 million unique files (hash, analysis JSON) between January 2012 and June 2014.

3.2 Feature Extraction

In order to apply machine learning algorithms, we must first transform the static and dynamic analysis of each executable into a feature vector. For simplicity, we choose to restrict each entry in our vector to a binary value. This means that the feature vector for a sample contains either a 1 or 0 at each index, depending on the presence of the corresponding feature in the sample.

3.2.1 Feature Categories

VirusTotal provides the analysis in the form of JSON. Some of the information comes from running static analysis tools, while other information is extracted from dynamic execution of the file in a sandbox environment. While we are provided with both static and dynamic features directly by VirusTotal, vendor classification software on VirusTotal often uses only their own static analysis. Thus, while we will use both static and dynamic features in our experiments, we keep the feature sets distinct within the pipeline so we can use just static features for direct comparison to anti-virus vendors on VirusTotal. For more information on the exact tools used by VirusTotal, visit https://www.virustotal.com/en/about/.

Some example categories of static features include binary metadata, digital signatures, packer detections, and static imports. Some dynamic categories include file and network operations, processes created or injected, registry key sets and deletes, Windows API calls, and dynamic imports. Each of these categories is represented by some segment of JSON, which we parse to create features. An example snippet of the JSON structure is as follows:

```
root
 |-- md5: string (nullable = true)
 |-- secondsfirstseen: integer (nullable = true)
 |-- positives: integer (nullable = true)
 |-- vtreport: struct (nullable = true)
 |    |-- ITWurls: array (nullable = true)
 |    |    |-- element: string (containsNull = false)
 |    |-- additionalinfo: struct (nullable = true)
 |    |    |-- asfminer: struct (nullable = true)
 |    |    |    |-- error: string (nullable = true)
 |    |    |    |-- result: string (nullable = true)
 |    |    |-- autostart: array (nullable = true)
 |    |    |    |-- element: struct (containsNull = false)
 |    |    |    |    |-- entry: string (nullable = true)
 |    |    |    |    |-- location: string (nullable = true)
 |    |    |-- behaviourv1: struct (nullable = true)
 |    |    |    |-- extra: array (nullable = true)
 |    |    |    |    |-- element: string (containsNull = false)
 |    |    |    |-- filesystem: struct (nullable = true)
 |    |    |    |    |-- copied: array (nullable = true)
 |    |    |    |    |    |-- element: struct (containsNull = false)
 |    |    |    |    |    |    |-- dst: string (nullable = true)
 |    |    |    |    |    |    |-- src: string (nullable = true)
 |    |    |    |    |    |    |-- success: boolean (nullable = true)
```

3.2.2 Collecting Features

One of the most work-intensive tasks in machine learning is feature engineering. To help reduce the amount of work involved, we design our framework in a way so that a developer only implements logic unique to their project. Custom logic for feature engineering is implemented in modules we call “featurizers”.

A featurizer takes a segment of the JSON analysis for a particular hash and outputs a set of string tokens that summarize the segment for that hash. After the JSON for every hash has been processed by the featurizer, all unique tokens are collected and a global ordering is created. Then a binary vector is created for each hash using the global ordering. This process is repeated for each featurizer. The final binary feature vector for a sample is simply the union of all the vectors coming from the individual featurizers. This whole process helps us create uniform-length, binary feature vectors used to build a matrix for machine learning.

Here, we'd like to highlight a few of the most common data types we find in the JSON and methods we use to turn them into feature values:

- **categories** - Each category will be a separate feature; the value will be 1 for every category for which this executable takes on.
- **numbers** - To transform numbers into binary values, we simply create bins spanning possible ranges and determine the bin that value falls in. For large ranges, we use schemes such as exponential binning ([1,2), [2,4), [4,8), ...).
- **sequences** - When order matters, such as a list of system API calls, we take N-grams over the list (mostly 2-grams and 3-grams). This keeps the ordering information while also chunking large lists into smaller patterns that can be associated across samples.
- **free-form strings** - If the string is simple, we use the string itself. When the string is obscure but follows a pattern, we normalize the string by finding a canonical representation. One such way to do this might be mapping all uppercase letters to 'A' all lowercase letters to 'a', all numbers to '0', etc.

Following is an example of a featurizer for the Network portion of the analysis JSON. The `HTTPSparseFeature` class examines the list of URLs the executable attempts to access dynamically. One such JSON segment looks like this:

```
http: array (nullable = true)
 |-- element: struct (containsNull = false)
 |--  |-- method: string (nullable = true)
 |--  |-- url: string (nullable = true)
 |--  |-- useragent: string (nullable = true)
```

with possible corresponding value

```
[
 ('POST',
 'http://b.coughstuffs.com/trackedevent.aspx?ver=2.0.654.0&rnd=641',
 'Custom_56562_HttpClient/VER_STR_COMMA')
]
```

We break this into the following tokens, annotated with descriptions:

Token String	**Description**
'$QKEYS: rnd ver'	sorted query keys, space separated
'$QVAL_bucket: 2'	log_3 of length of all query values
'/trackedevent.aspx'	path
'b'	partial split of hostname
'coughstuffs'	partial split of hostname
'com'	partial split of hostname
'b.coughstuffs.com'	full hostname
'$entropy_21'	entropy of url, captures character diversity and total length
'http://b.coughstuffs.com/trackedevent.aspx?ver=2.0.654.0&rnd=641'	full url
'$LEN_bucket: 3'	average length of urls
'$GET_bucket: 0'	# get requests, binned exponentially
'$POST_bucket: 1'	# post requests, binned exponentially
'$REQUESTS_bucket: 1'	# total requests, binned exponentially

Here we see that this featurizer generated a set of 13 string values. To create a binary vector corresponding to the `HTTPSparseFeature` for this particular sample, we join all sets generated by the featurizer and then mark the indices corresponding to these 13 values as 1. The length of this vector is in the order of tens of thousands since there are many unique values URLs take on. As a note, the third element in the entry is the user agent of the request, which has a separate featurizer, so we don't see any feature values for it here. We recognize that there is definitely room for improvement here.

For example, we do not consider query values individually, which was a decision made in an effort to reduce the size of the feature space as we did not believe the benefit was worth the cost of increase in size of the matrix.

3.3 Learning

3.3.1 Methodology

Once we've created a binary feature vector for each hash, we stack all the feature vectors to create a feature matrix. Each column corresponds to a particular feature and each row corresponds to a particular scan. The fact that rows are scans and not hashes means there are 5 million rows, some of which are exactly identical. The reason for having multiple rows for some hashes relates to our testing methodology.

In machine learning, it is typical to perform randomized k-fold cross validation, which roughly follows these steps:

```
Shuffle the dataset
Split the shuffled dataset into k equally sized segments
For i from 0 to k-1:
        Train model on a merge of segments {0,..., k-1} - {i}
        Test model on segment i
        Save performance/results
Average or aggregate the performance over all models
```

While k-fold cross validation has been widely used in the malware research community, we believe it is not ideal and does not carry over correctly to real-world situations. k-fold cross validation does not properly evaluate samples that evolve over time. When time is relevant, training on one set of samples and testing on another only makes sense if the data in the training set lives in a time period before the data in the testing set. If the dataset is shuffled and randomly split, the model will train on samples in the future to predict the labels of samples in the past. What should happen instead is that a model should only classify samples using information collected prior to the appearance that sample. This leads very naturally to a system that evaluates models in a temporally-consistent manner:

```
Order the dataset in time
Split the dataset into k segments by some time interval (ie. 1 day)
For i from 1 to k-1:
        Train model using segments {0,...,i-1}
        Test model using segment i
        Save performance/results
Average or aggregate the performance over all models
```

It is important to note that starting from segment 1 means your training set only spans one period of time. Since this is not really realistic, it may make sense to start evaluating models at some point later in the dataset where the training set is large enough to be meaningful.

3.3.2 Implementation

At this point, our pipeline has generated a matrix that contains a feature vector for each hash. Using parameters specified by the user, such as retrain period length, scan events are arranged chronologically and then split into the segments mentioned above. Remember that there may be more than one scan per hash and that we are interested in scan events because each scan represents one classification instance. Each scan of a particular hash, however, uses the same feature vector since the file is exactly the same and the JSON analysis rarely changes between scans. The vector of vendor classifications is converted to a single binary label by determining whether the vector contains at least 4 malicious classifications. We arrive at the number 4 through analysis of how vendor classifications stabilize over time. This single label is known as our "test" label. For each scan event, we also compute another "gold" label using the vendor classification vector of the latest scan for that hash. The models will train on test labels but performance will be measured with gold labels, since test labels are what is available when samples are scanned but gold labels are more correct since vendors have had more time to make a better classification. Scan events are then represented by a feature vector, test label, and gold label.

We then create a set of tasks, each consisting of a collection of training events and testing events. Each task represents one iteration in the for-loop above. Even though there is a temporal dependency, each task can be run independently from each of the others since we already have all of the data from segments 0 to k-1. The feature vectors and labels of all events in

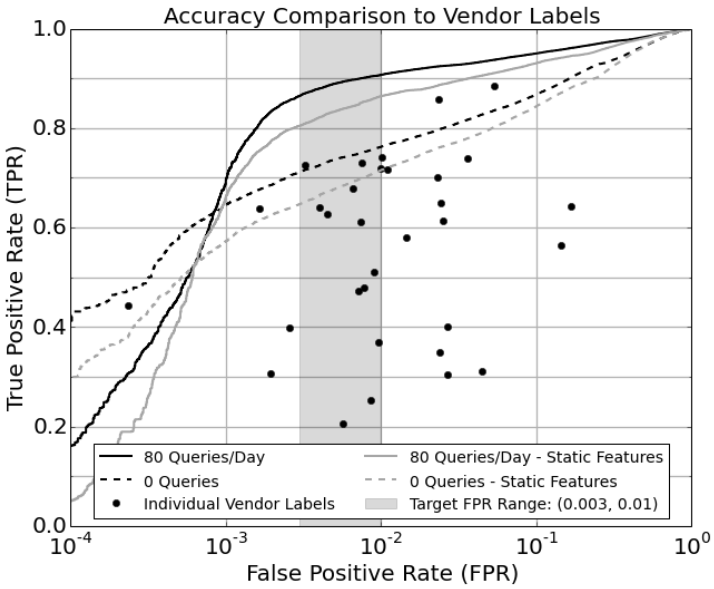


Figure 1: Results diagram for pipeline.

- The darker dotted line follows what has been described in this article so far.
- The lighter dotted line uses the same technique, but only uses static features.
- The darker solid line chooses the 80 samples that our classifier is most "uncertain" about, and queries an oracle for the ground truth to be incorporated into the model. This is known as active learning and simulates the ability to hire leverage humans to manually and correctly classify a number of samples each day.
- The lighter solid line is 80 queries as well but only using static features.

a particular iteration are stacked appropriately to create a feature matrix and two label vectors for the task. All tasks are forked off and run in parallel on a cluster.

Our pipeline has different implementations of various machine learning algorithms, but after extensive experimentation, we settled on using logistic regression for our particular malware use case. Support vector machines, random forests, and dimensionality reduction combined with other techniques yielded rather similar results. We use logistic regression because its linear nature offers better explainability in classifications, while providing similar, if not better, performance in comparison with many of the other models we tried. Much of the following user interface design is based on the fact that we use a linear model in order to explore the contributions of individual features to the final classification decision.

3.4 Results

Although the direct design and performance of the pipeline itself was not the focus of my individual work, a short summary of the results and conclusions are included here for completeness.

Included in the Figure 4 are several aggregate receiver operating characteristic (ROC) curves based on different evaluation/-modeling techniques. Vendors are represented by dots that are calculated directly from their performance in our dataset. Our system is represented by a curve because our model allows us to shift our decision function to any point on the curve.

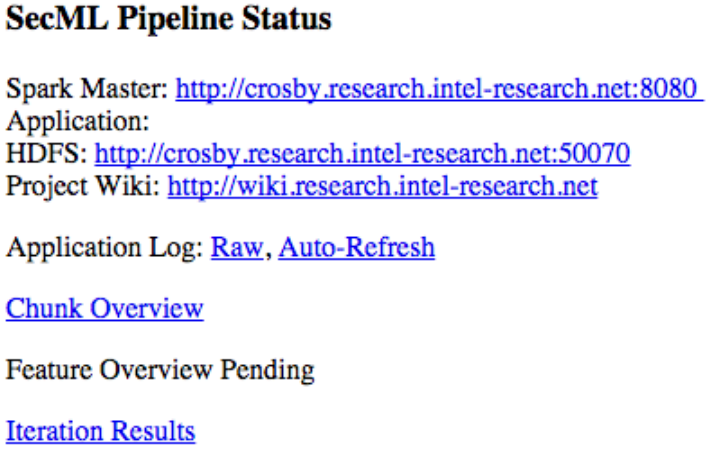
What we see here is that our pipeline is able to match the performance of the most accurate vendors and even beat them with limited human resources. Over 1 million samples overlaid into over a hundred models can be processed in a matter of few hours using a small cluster of about 10 nodes.

4 UI Stages of Development

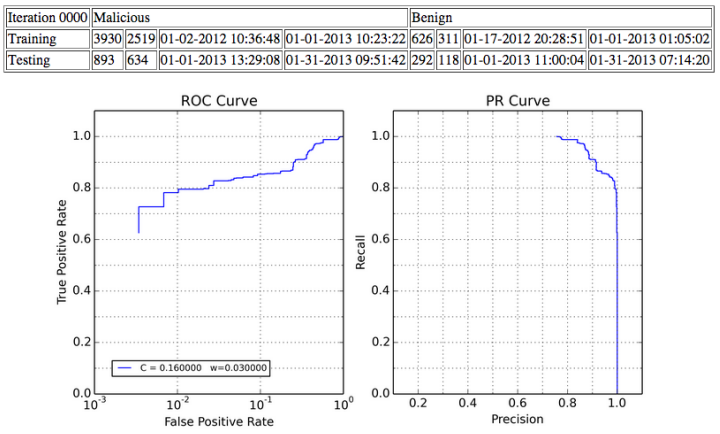
We now go through the history of the web interface for the pipeline as a story parallel to the development of the core pipeline. This was the focus of my work. While the results of the pipeline have always been exciting, it proved to be extremely difficult to gain any insight without digging deep into the data and the code used to manipulate it. This demonstrated the need for a user interface that could be used to explore information generated during certain stages of the pipeline. What first started out as a basic web server used to serve static ROC images, eventually became a web application for managing jobs and exploring results. As the pipeline progressed in its development, we designed a new interface to meet the needs of different types of users. We wanted it to be lightweight and fast, yet include powerful, interactive ways to explore the data.

4.1 Early Stage

The original web user interface was created by the primary pipeline architect, Brad Miller, to view the status of a job and collect ROC curves to view results. The application log page would simply open the log file and write it into a socket as an HTTP response. A second webpage would load a pre-generated HTML file that included an information table and ROC curves for each iteration. The interface was implemented as a bare bones write-into-socket HTTP server. It was simple, but it served its purpose for the developers who were using it at the time. There were many more important tasks at the time to focus on, such as better featurization and pipeline architecture, dataset acquisition, learning algorithms, etc.



(a) home page for a job



(b) results page

Figure 2: Two pages on old web UI.

4.2 Growing Pains

As more people joined the project, the interface underwent changes to suit the needs the new users. One notable addition to the user interface was the Feature Overview page, which was a simple HTML table that displayed all possible feature values for a featurizer and the number of occurrences of each feature. This was extremely helpful for those working on featurization since it provided a quick summary of all the token strings that were being generated over all the samples. While this worked well when testing on a 1% or 10% subset of the dataset, it soon became obvious that this was not scalable as some HTML pages were tens of megabytes in size and would freeze a web browser upon opening them. This drove the desire to scrap the static interface and design a new more scalable, interactive one from scratch.

4.3 Redesign

The original focus of the new design was to create a lightweight, interactive web interface that could scale with large datasets. One of the drawbacks of the original UI was that it ran a separate web server for each job to display results. The new UI would execute as a single service that makes it easier to access and compare different jobs, including jobs run by other users. You would also be able to check if there are enough available resources in the cluster to run more jobs. Job results would be interactive and help users gain insights on what kinds of classifications were made and why they were made that way. The design and implementation of this new user interface is the focus of the rest of the paper.

5 UI Redesign

Here we outline the two pages of the new redesign and go through the thought process behind the inclusion and design of each element on both pages.

5.1 Pipeline Overview Page

While Spark provides its own cluster management interface, it is much more convenient to have all the information you need in one place. Since the jobs we run sometimes take hours, we want to provide a single location to check on both the status of the cluster and the progress of our jobs. From this interface, we want to be able to see what is going on in the cluster, launch new jobs, check on the progress of live jobs, and see the results of completed jobs.

5.1.1 Cluster Status Header

The cluster status header includes mainly information pulled from the Spark cluster management interface. The cluster we used for this project consisted of 10 machines with a total of 40 cores and 400GB RAM. The status of the workers, cores, and memory are scraped and included here, as well as the amount of free RAM on the master node. We include this last number because our python driver uses local memory outside of the Spark JVM number provided by the Spark interface. The local memory on our master node, Crosby, is often a heavily contested resource when it is shared by multiple concurrent users.

The availability of this set of information is critical on a system with many users because of several reasons:

1. The number of workers specifies the configuration of the cluster and indicates if the cluster is up and running.
2. Spark will not execute a job if no cores are free.
3. Low cluster memory means lots of swapping and slow execution.
4. Running out of memory on the master node, Crosby, results in tasks being arbitrarily terminated, which may randomly kill the running jobs of various users.

Just below the header is the list of running applications pulled directly from the Spark interface, which is useful in viewing per-node output and debugging purposes.

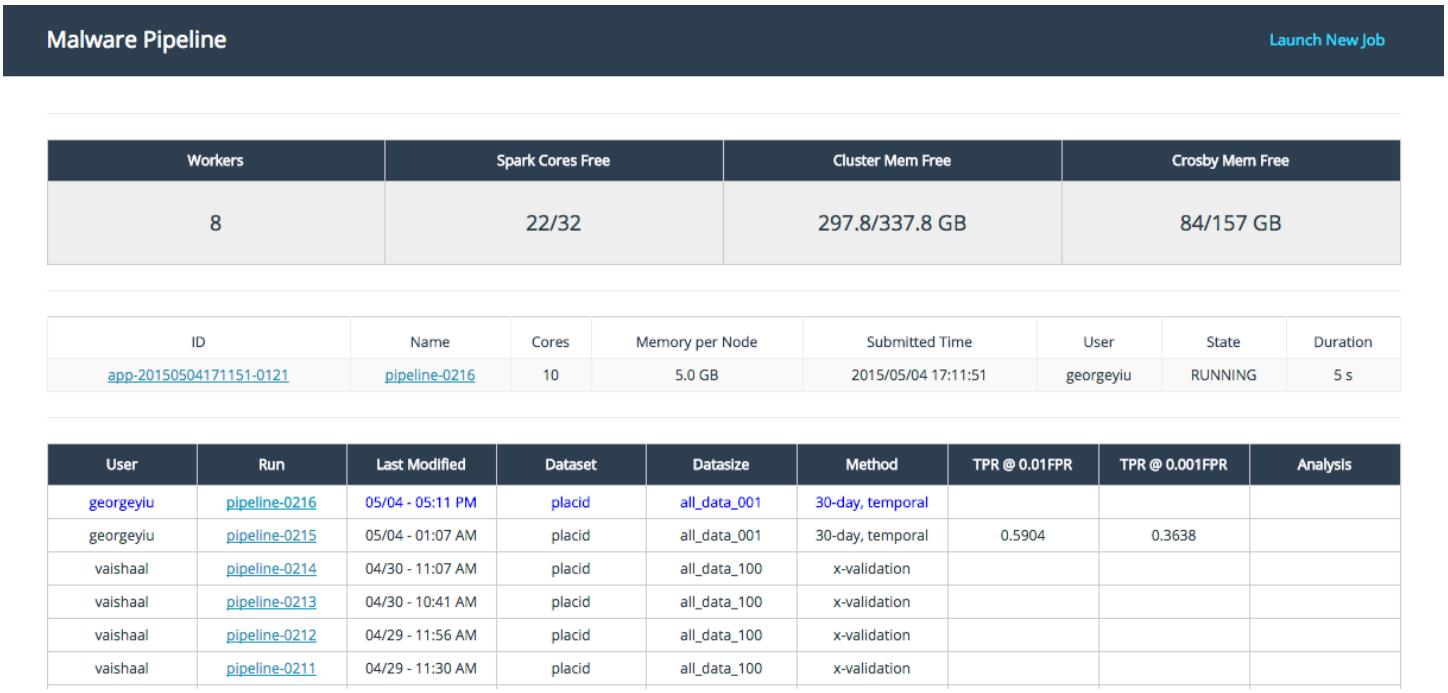


Figure 3: Portion of the pipeline overview page, including cluster status header, running spark job list, and job table.

5.1.2 Job Table

The job table lists all jobs that have been run up to this point in time. Jobs highlighted in blue are still running. The columns describe the most important characteristics of each job, including dataset size, evaluation period, and results. Each column is sortable, so you have easy access to things such as the job with the highest classification accuracy in its results. Clicking the link under the Run column will bring you to the job summary page for that job, described in Section 5.2.

5.1.3 Job Launcher

The job launcher, which is accessed through a link on the far right side of the navigation bar, lets anyone easily configure and start a job. Although we usually start jobs through a Python script via the command line, this interface provides a more friendly way to start jobs for those less code-inclined or new to the pipeline. Inclusion of this element is important in allowing a user to stay within the web UI from start to finish. While we as developers of the system don't really need this capability, it is a key piece in designing a piece of software for a wider audience to use and improve. In the future, we hope to make this launcher more dynamic so users only have the option to start jobs within the available resources.

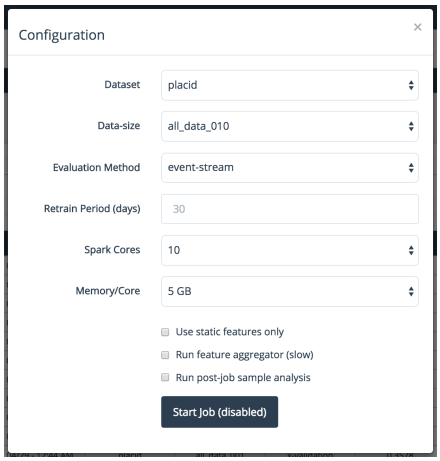


Figure 4: Job launcher modal.

5.2 Job Summary Page

The job summary page contains all the information and results for a single job in the pipeline. The page is immediately available as soon as a job is started. Each section has a header that controls whether the section is collapsed using the Collapsible [3] library.

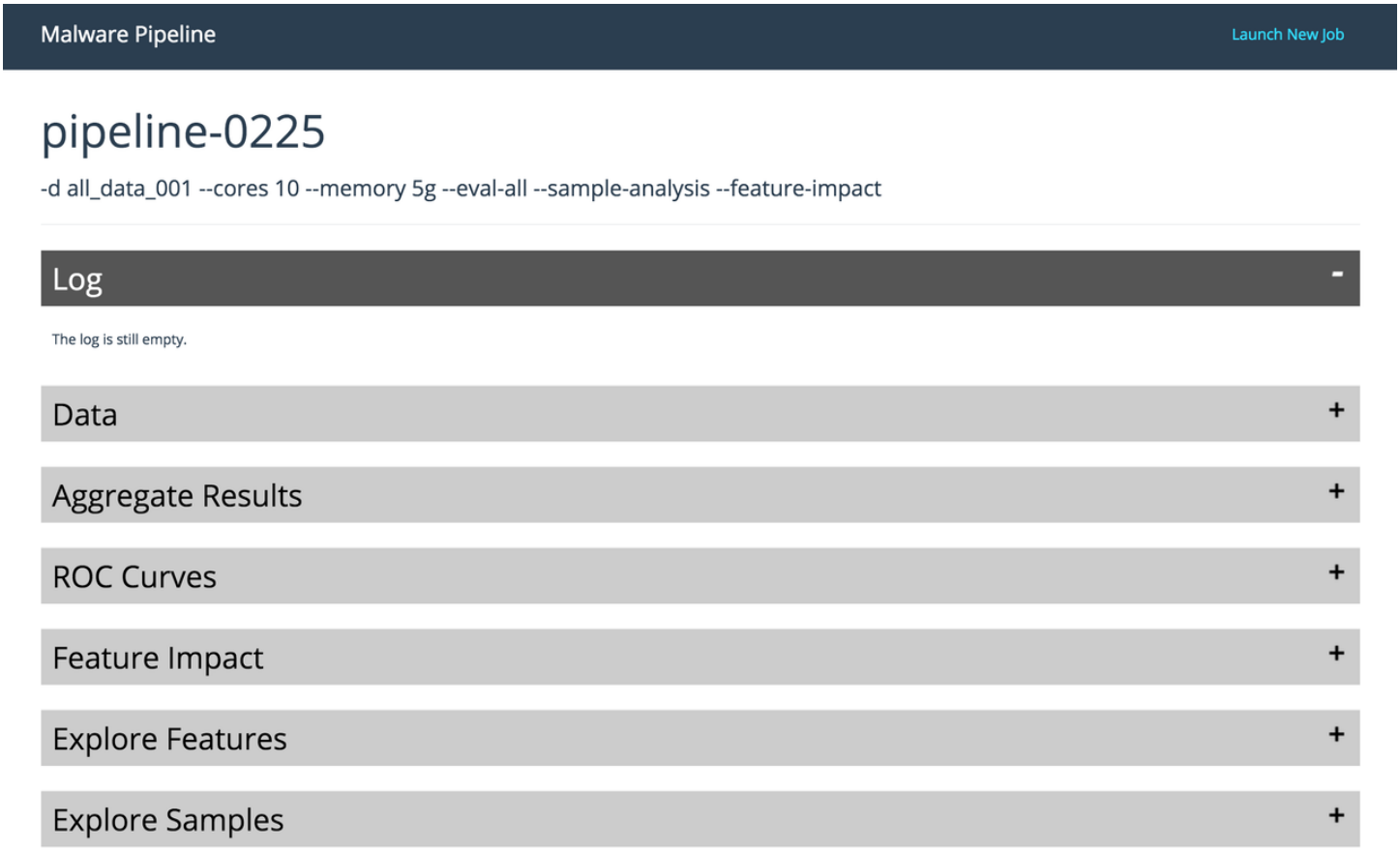


Figure 5: Job summary page with all sections collapsed except the log. The title shows the job number and the arguments passed to the script used to launch the job.

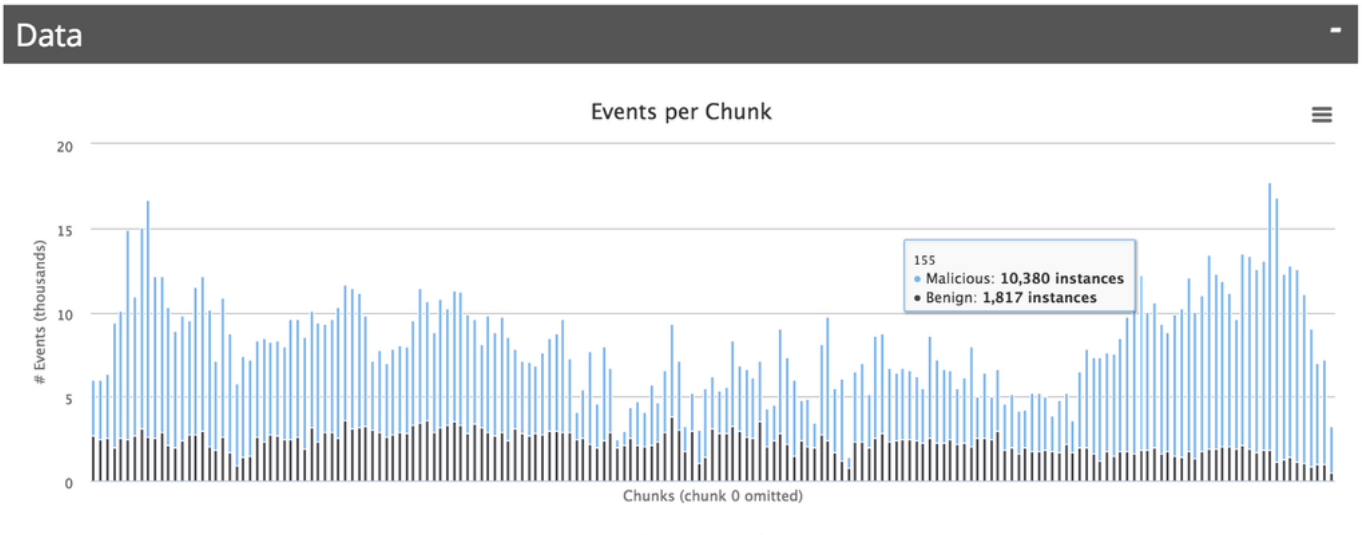
5.2.1 Log

The log was one of the two core parts of the original web interface driven by developer need. The log provides a way to monitor the progress of a job without wading through the verbose Spark status messages, while still being able to see them from the terminal. The log HTML is periodically updated every few seconds through an API call to the server so that the information is always live and available without the need to manually refresh. While the log currently contains information relevant mostly to a direct developer of the system, one can imagine this being transformed into a progress log of sorts that provides helpful updates on exactly how far the job has progressed for those less familiar with the system.

Figure 6: Example segment of a log.

5.2.2 Data

The data section shows the size of each segment through a bar chart and a table with similar information. In the figure below, we split our 30 month dataset into 3 day periods, but combine the first 12 months of segments into an initial training segment that we denote as segment 0, omitted on the graph for obvious visual reasons. The remaining 184 segments span the other 18 months, where each chunk represents the set of test samples for the model trained on all samples occurring before that chunk. This gives us a visualization on how evenly distributed our dataset is both over time and within each period (between malicious and benign samples).



Chunk	Malicious Events	Malicious Hashes	Malicious Start	Malicious End	Benign Events	Benign Hashes	Benign Start	Benign End
0	385561	239558	01-01-2012 00:34:44	12-31-2012 00:31:50	73081	34420	01-01-2012 00:35:24	12-31-2012 00:19:43
1	3305	2401	12-31-2012 00:38:39	01-03-2013 00:34:22	2673	2093	12-31-2012 00:58:14	01-03-2013 00:32:29
2	3570	2742	01-03-2013 00:35:44	01-06-2013 00:31:12	2457	1982	01-03-2013 00:35:59	01-06-2013 00:18:22
3	3808	3110	01-06-2013 00:43:21	01-09-2013 00:34:39	2523	2054	01-06-2013 00:37:32	01-09-2013 00:27:39

Figure 7: Graph of 18 months of data split into 3 day segments and table form (truncated).

5.2.3 Aggregate Results

The aggregate results section is arguably the most important section on the page. In this section, there is a single graph that plots our ROC curve versus vs the (false positive rate, true positive rate) of each of the vendors we are comparing against. Our ROC curve is computed by combining the results used to compute each individual ROC curve and computing an ROC curve over all of them together. In this way, the single ROC curve presented is represents the accuracy we can achieve over the entire dataset, since all segments test different subsets of the data. Each vendor can only be represented by a single point because we have a single measure of true positive rate vs false positive rate for their classifications, with no extra information on distance from decision boundary necessary to shift their decision function.

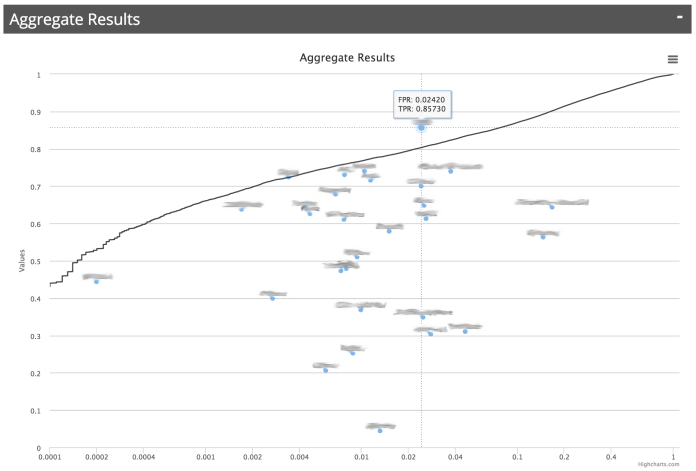


Figure 8: Aggregate ROC curve vs vendors. Vendors blurred to preserve privacy.

5.2.4 ROC Curves

The ROC Curve section is pulled from the first web user interface. We needed a way to see that our models made sense. The aggregate ROC curve section was added only once we needed to begin summarizing and analyzing our pipeline as a whole. In this section, ROC and precision-recall curves are generated for both training and testing sets for each iteration, where each iteration corresponds to one time period designated as the test set for the model trained on all samples prior to that period.

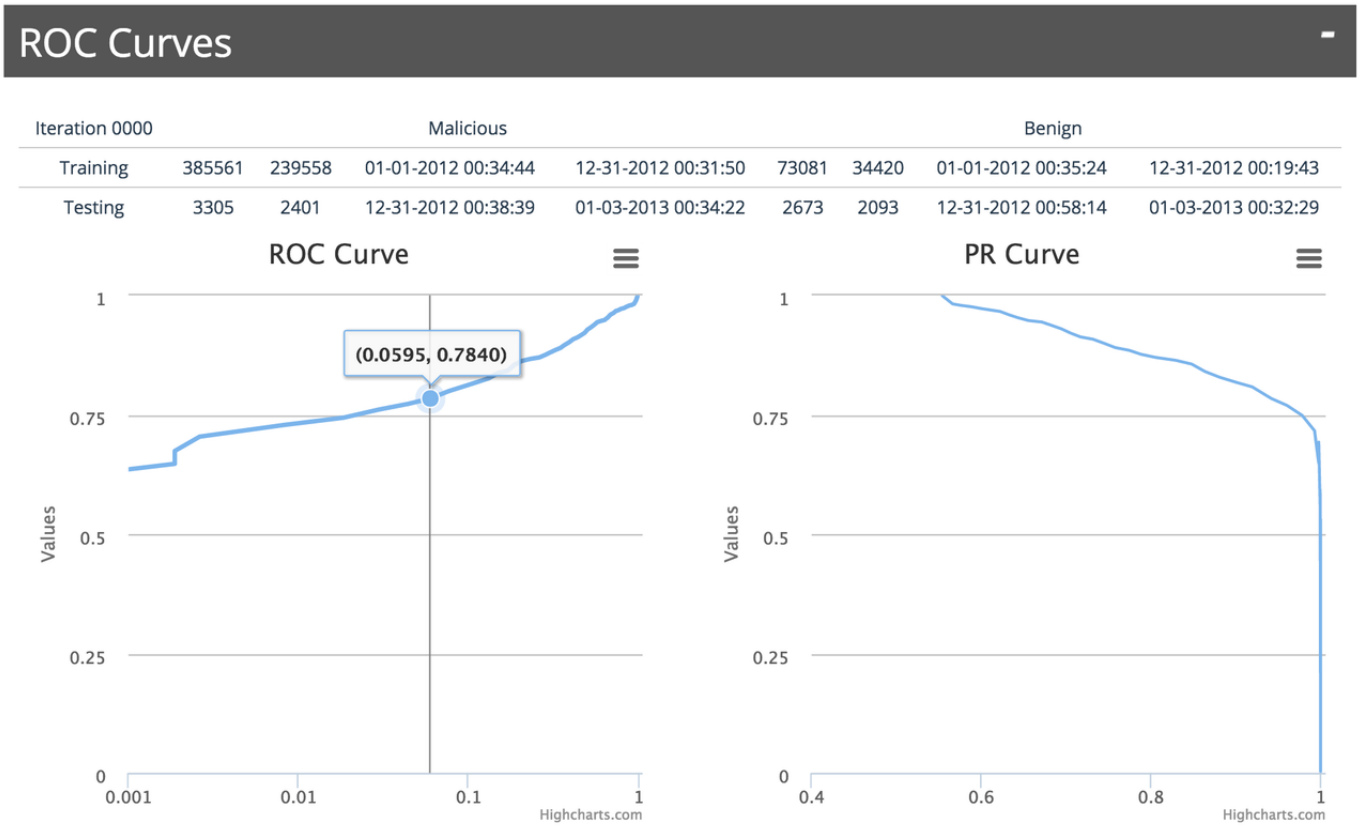


Figure 9: ROC and PR curves for first iteration.

5.2.5 Feature Impact

The feature impact page contains various different categorizations of features and their total impact by category. Impact incorporates both the weight of the features and the prevalence of each one, which we think of as a sort of deciding power in the classification of samples. This section can be used by the developer in identifying categories of features that are important in the classification decisions. Categories with low impact can either be further developed or scrapped during the feature engineering process.

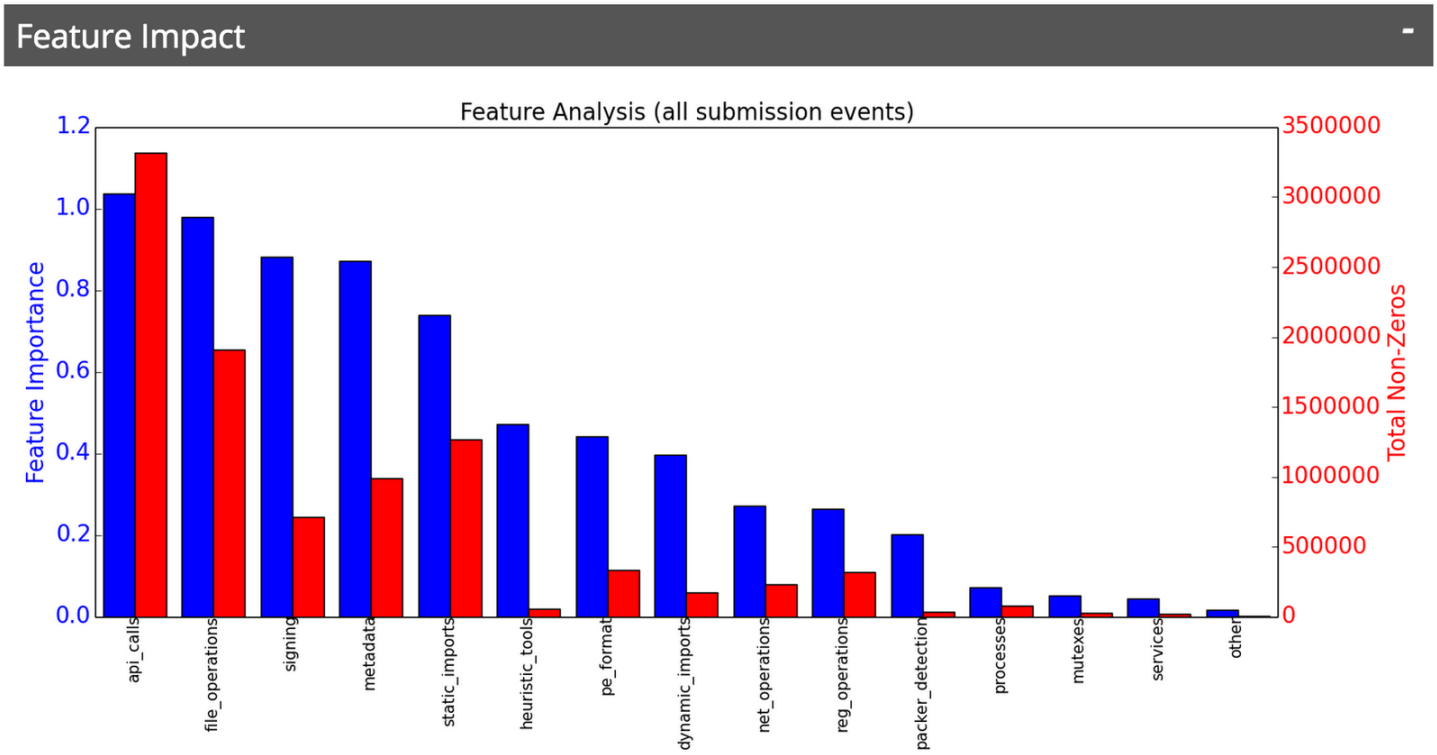


Figure 10: Feature impact example, generated as a static image. For this job, API calls and file operations have high impact and mutexes and services have low impact.

5.2.6 Explore Features

As mentioned before, we first realized the need to rebuild the web interface when opening 50MB HTML tables crashed our browsers. While they did open in text editors, scrolling through a text file in a command shell was not exactly user friendly. After several iterations, we found the DataTables [6] jQuery plug-in to be the most appropriate for the task of displaying large tables on webpages. This type of table allows a user to go through "pages" of the dataset and sort the data on any column. The table is updated client-side through an API call to the server that requests only the next rows that will be displayed. All of the processing is handled in the backend, so the user experience is smooth from the web page. We discuss the usage of DataTables.js and the backend built to support it later in the architecture section.

The table contains all of the feature tokens generated for the hashes during the feature extraction stage. The table includes the featurizer that generated the token, the token literal, and the average weight of that token across all models for which that token is a part of. In the learning phase of our pipeline, we discard all features that occur less than some number of times (for example 100) in our training set. We do this in order to limit the size of our matrix to speed up the training process. Depending on the dataset, our discarding methodology typically reduces the number of features in our matrix from several millions to a few hundreds of thousands.

Explore Features

Search:

First Previous Next Last

Family	Token	Avg Weight	Iters Present	Graph
PEResourceListSparseFeature	$hash/0effc6288b6ce1f933c8b97dc8ec5e6ee883f0628bea176538f65b0b2297d1fe	0.68994575	150	view
SectionsSparseFeature	$md5/df4c9346b66ab5297a90e5792bc8ba9b	0.66956446	8	view
PEResourceListSparseFeature	$hash/4a11d6a99295629f1109337d16a6cb23eb05521130ff0354cb3aff4a6c5ce55a	0.66956446	8	view
CommandUnpackerSparseFeature	RAR	0.62885123	183	view
FProtUnpackerSparseFeature	Themida	0.62139879	183	view
PEResourceListSparseFeature	$hash/ff1812c2fa2d29e789f78353eaad08053a313fb5027412c96e52c4a6c972c7ee	0.60028811	125	view
PEResourceListSparseFeature	$hash/414671cf903303df03ed8086c9cc69186e725bbc8007ba7c7825f9b355bf8536	0.57701554	32	view
PEResourceListSparseFeature	$hash/ac75389765d91a008492481ea432b023cb20b3dd97e8cfbd2e1b4e9024bd4d35	0.57601360	135	view
SectionsSparseFeature	$md5/951400509feec7cf4475f13ea729829a	0.57601360	135	view
SectionsSparseFeature	$md5/9da2ebc23cbd31a6dc45b5244d784707	0.56701364	127	view
PEResourceListSparseFeature	$hash/a73d73cbf6ca3050db0d45129a981cc306727e81e18b80c075a4804369b3a595	0.56223385	183	view
DNSSparseFeature	cdn.mxpnl.com	0.55000396	137	view
SectionsSparseFeature	$md5/68f934b578e606fc96cbaf59ed5f2767	0.54575471	134	view
PEResourceListSparseFeature	$hash/53eca32f5a82c8459bad8eddef7106aadc6d2643a3250c054d9ecbe5293e46b3	0.53510046	12	view
SectionsSparseFeature	$md5/ce96f59165cefd81694c6bd44bfafc9c	0.53510046	12	view

Showing 16 to 30 of 210,171 entries

Figure 11: DataTable displaying feature tokens and their average weight across all models.

Upon clicking the "view" link for a particular sample, a modal overlay comes up displaying a graph of the model weights for that feature over time. Through this graph, as seen in Figure 12, we can see how a particular feature changes in time from model to model. A jump in the graph might correspond to increased usage of that token in malicious samples, and vice versa for dips in the graph. A similar graph can also be made using our notion of feature impact from before.

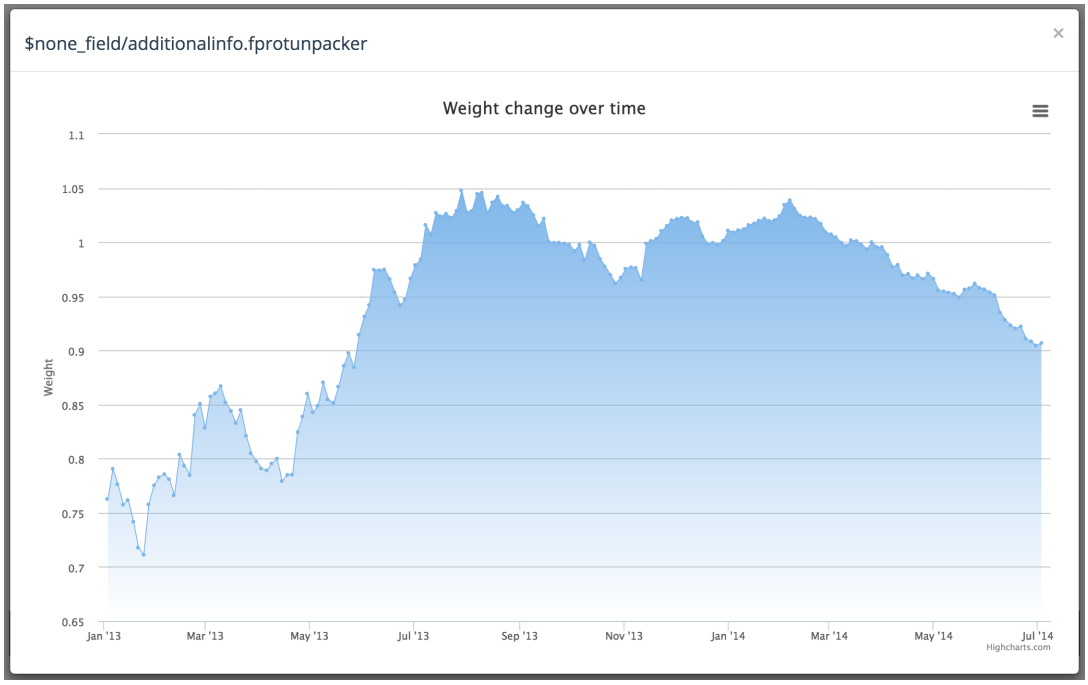


Figure 12: Graph of weight vs time for '$none_field/additionalinfo.fprotunpacker' token. The token was generated as a result of an empty fprotunpacker field in the JSON.

5.2.7 Explore Samples

The most interesting part of the page is the explore samples section. One of the most frustrating things about using machine learning is its limitations in helping the user understand why it chooses to classify a sample a certain way. On top of the black-box nature of many machine learning algorithms, the distributed nature and scale of our project made it difficult to assemble the necessary data to properly explore the results. Even after building out a pipeline that seemed to perform very well, it was hard to verify that our results even made sense. While we were satisfied with ROC curves, it was obvious that we needed to do more if we ever wanted to create a general purpose platform that was easy to use. This drove the need to build a section that made it easy to explore individual samples and verify that we were doing the right things with them.

We decided there were two fundamental ways to examine samples. The first way would be through a table similar to the one used to explore features. The table is searchable and sortable and contains a link to a modal that contains information for that particular sample. The following table explains each of the columns of the figure below it.

Column	**Description**
MD5 Hash	sample hash with link to open up modal for more information (the figure has SHA256 instead of MD5)
VT Pos - Test	The "truth" label of the sample during test time, which is computed using some subset of the labels given by vendors on VirusTotal. We use a threshold of 4+ vendors within a subset of about 30 vendors that label most samples. A green checkmark is benign and a red x is malicious.
Prediction	The prediction by our model. All samples depicted in the figure are predicted as malicious.
VT Label - Gold	The "gold" label is the farthest label in the future that we have for a particular sample. The label is constructed by taking a threshold of 4+ on the latest scan results that we have for a hash. As time progresses, what vendors think are benign samples often get rescanned and classified as malicious. We compare our prediction against this label, even though we train on labels only available at test time (VT Label - Test).
VT Pos - Gold	# vendors on VirusTotal classifying this sample as positive (malicious) at the latest scan in future
% Malicious	Decision function fitted to logistic curve. 50% denotes high uncertainty while 0% denotes very likely benign and 100% denotes very likely malicious, as predicted by our model.
Type	Classification type. True positive, false positive, true negative, false negative. Modifier added to allow for specific kinds of searches. For example "-novel" means novel detection, which we have filtered this table by. Novel detection means we disagree with the current assessment by vendors but correctly predict the future classification.
VT Scan	A link to the Virus Total report for that sample. The report shows all sorts of information such as vendor classifications and static/dynamic analysis.

After we first built the Explore Samples table, as seen in Figure 13, we immediately noticed something strange. As we were exploring our worst false positive classifications where the gold label (see row in previous table for definition) was benign but had a high malicious probability, we clicked through a few VT Scan links to look for more details on the samples in their VirusTotal reports. What we saw was that these samples were actually labeled as malicious by most vendors, meaning our VT gold labels were wrong. This exposed a bug in our dataset ingestion script where it failed to pull the latest updated scan from VirusTotal for a small percentage of the samples. Being able to look through the data in this way immediately helped us learn more about our own system and improve the accuracy of our results.

A second way to explore the samples is through a graph view, seen in Figure 14. We found some of the most natural properties of a sample to be the time at which it was discovered (and tested), the size of its feature space (number of nonzeros in its feature vector), the classification at test time, and the gold-label classification. From this data, we built a dynamic graph that can be updated with any of those choices as axes. Since there can be millions of samples, the graph only shows a random sample of them. We found 3000 to provide a good density, but users can also choose other numbers if their browser cannot handle that many points.

Another feature of the graph is the ability to sample different classification types at different rates. Since our model classifies the majority of samples correctly, most of the samples would be true positives or true negatives if a uniform random sample was taken. In order to see some of the more interesting points, false positives and false negatives, we can use sliders to adjust

Figure 13: DataTable displaying samples and their labels/predictions.

the rates such that more points show up on the graph. Notice that all malicious samples (circles and triangles) are colored red while all benign samples (squares and diamonds) are colored green. Those that lie on the opposite side of the blue line at 0 are misclassifications. Note, we can visualize this only because we are using the logistic regression model. A graph like the one shown below with axes (iteration, weight) can help us easily pick out patterns in the data such as "there is a particularly high rate of false positive samples in iteration x" or "there is a low number of sample points in general in iteration y". Each pair of axes helps us make similar, insightful observations on the data.

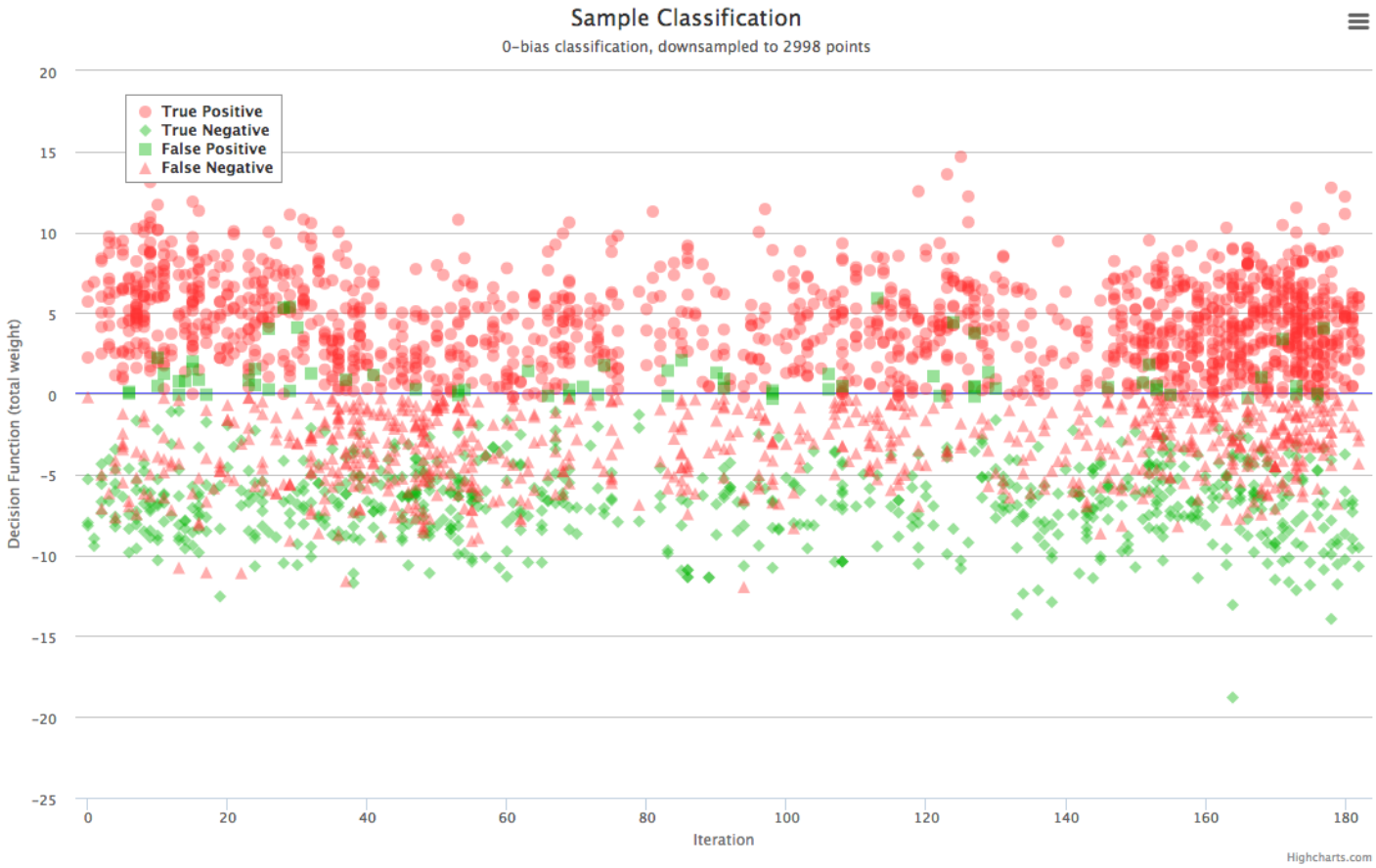


Figure 14: Graph of 3000 samples by iteration and weight; false positives sampled at a higher rate.

When you click on a hash link in the table or click on a point in the interactive graph, a modal such as the one in Figure 15 will pop up, similar to what happens when you click the graph link in the feature explorer table. There are three main parts to this pane:

- an information section (essentially the information from the row in the table)
- a chart of the feature categories and their corresponding aggregate weights
- a feature table containing tokens generated for that sample

The information section at the top currently just displays the same information as the row in the table. It is important to include it here in case the user opened the modal through the graph, where the information is not available (though it can be added to the tooltip that appears on hover).

The chart shows the aggregate weight of all the features from each featurizer. This helps the user quickly visualize what types of features the model thought were malicious or benign. If you click a bar in the chart, the table beneath is automatically filtered by that feature type. If no filter is applied to the table, all the tokens generated for the sample are listed and can be ordered by weight. These string values correspond to all of the nonzero indices in the feature vector generated for the sample. The tokens are colored in a gradient between green and red, based on the weight (but will likely be changed later for colorblind users).

Figure 15: Modal for a hash `fbcbb14...` Feature table filtered by the `SectionSparseFeature` featurizer.

6 UI Architecture

As simple as the front end seems, many technical challenges needed to be solved to deliver a good user experience. Here we will walk through the design choices and implementation details of the redesigned web user interface.

6.1 Data Pipeline

6.1.1 Storage

A new directory is created in a designated folder to hold the output for each job. Throughout the job, various files are placed in this directory for future reference and for the web interface, most notably:

- a text file for the log
- the information describing each chunk (size, timing, etc)
- a "pickled" (serialized) copy of the trained classifier for each iteration
- the indices of the columns used in the feature matrix for each iteration
- the indices of the test samples in the feature matrix for each iteration
- the labels corresponding to those samples for each iteration
- the ROC curves from each model, along with any other results

In addition to this set of data, which is saved on the local machine, several things are cached in HDFS:

- feature matrix for this dataset
- matrix column index -¿ string token mapping for all features in that matrix
- feature type (featurizer) -¿ range of column indices mapping
- label vector (containing labels given by all vendors) for each scan event

These items are stored on HDFS because they take up a lot of space and are rarely recomputed since most jobs (that we have run recently) do not alter the underlying feature matrix. Feature extraction is generally the slowest step, so we skip it whenever we can. The full matrix currently uses 1.65GB and the index to string token mapping uses 1.9GB. A different matrix is created and stored for each user/dataset/size triplet. Regarding size, our dataset is also partitioned into 1% and 10% portions, which allow us to iterate faster when we don't need full results.

6.1.2 Postprocessing

Once the core part of the pipeline has run to completion, a post-processing job to transforms the data into a form usable by the web interface. Specifically, data needs to be processed into a compact form that can be queried quickly by the client. Here we break down the most data-intensive UI elements and the forms of data needed to make each one possible:

UI Element	**Data Needed**
explore features table	(family, token, avg weight) per feature
features weight graph	all weights across all models per features
explore samples table	(hash, test label, gold label, prediction, classification confidence) per sample
explore samples graph	(hash, iteration #, # nonzeros, total weight, test label, gold label) per sample
explore features chart/table for sample	aggregate weight per feature type per sample, weight per token per sample

The post processing job serves to combine all the sources listed in the previous section into the above mappings containing minimal information stored as JSON on the local machine. We choose to keep the 1.65GB (sparse) feature matrix in HDFS since it is rather large. However, since we still need to access the feature matrix in order to find the nonzero indices for a particular sample (for the explore features table per sample), we shard the original feature matrix into 3000-row (5MB)

chunks within HDFS. This doubles the storage cost in HDFS, but still keeps this information available in sub-second, load times.

While a database fits the notion of table displays very well, we wanted to see if storing the data in local files could be space efficient and fast enough to query (ideally ¡100ms on average). We performed the following experiments on our largest index with 850,000 key value pairs, each using the memory of roughly ten 32-bit numbers and two 50-character strings.

Python Libraries	Dump w/ disk write (ms)	Load w/ disk read (ms)	Disk space used
pickle	39374	35094	156M
cPickle	8073	7077	143M
json	4543	8168	128M
ujson	1991	5891	113M
msgpack	1270	2734	96M

We observed that the MessagePack [2] library (msgpack) greatly outperformance even the C-based UltraJSON [4] library (ujson). While this is great to observe, we would like to see faster load times before believing that file-based storage is viable. From here, we combine ujson and msgpack with a couple compression libraries to see if we can reduce disk I/O time as well. We used the default compression level in each of these examples.

Python Libraries	Dump w/ disk write (ms)	Load w/ disk read (ms)	Disk space used
ujson + zlib	9006	4671	52M
ujson + lz4	2396	4746	85M
msgpack + zlib	9315	2316	55M
msgpack + lz4	1567	1642	79M

79MB is very reasonable for a job with over 1 million samples. As a note, we do not actually care about the time it takes to dump JSON to disk since we only need to perform it once when we run the post processing job. While 1.6 seconds is still relatively slow for use in a web application context, we are able to bring the average query time down by a significant amount with intermediate caching. That caching will be discussed in the next section. Although we have not done so yet, the next obvious step is to compare these benchmarks with those based on using a database.

6.2 Web Application

Coming from a simple socket-based HTTP server, any basic framework would have provided a lot more structure to our project. Since our project is Python based, it made the most sense to choose a framework that uses Python. We chose the lightweight Flask [1] framework, which some people in the research group were already familiar with, over the more popular Django framework, which seemed unnecessarily complex for the simple interface we had in mind. From there, we started building out the two main web pages of our application.

6.2.1 Cluster Status Page

The cluster status page is relatively simple in terms of implementation. The cluster status information is scraped directly from the Spark UI and inserted into the webpage. To create the job table, we run the directory containing all the output directories of the job through the stat command and extract the relevant pieces of information from file dumps inside that directory.

6.2.2 Job Summary Page

Around half of the sections on the job summary page involve simple conversion of static data on disk to HTML form. Those are not of particular interest. However, we use two dynamic Javascript libraries extensively throughout the job summary page, Highcharts.js [7] and DataTables.js [6].

Highcharts.js

Highcharts is a Javascript library that makes it extremely easy to create dynamic, interactive graphs. One of the most popular graphing libraries is the open source D3.js. With little to no experience in either, we chose to go with Highcharts since it seemed easier to create relatively standard graphs with few customizations. With Highcharts and a little bit of work, we were able to make the interactive sample explorer graph. Some simple features that made it all possible:

- click-event handlers on points in the graph
- custom tooltips to display whatever information we want while hovering over a point
- the ability to load and reload series on a graph with simple API calls
- select-to-zoom on the graph

Figure 16 includes some of some the different ways we are able to view the samples.

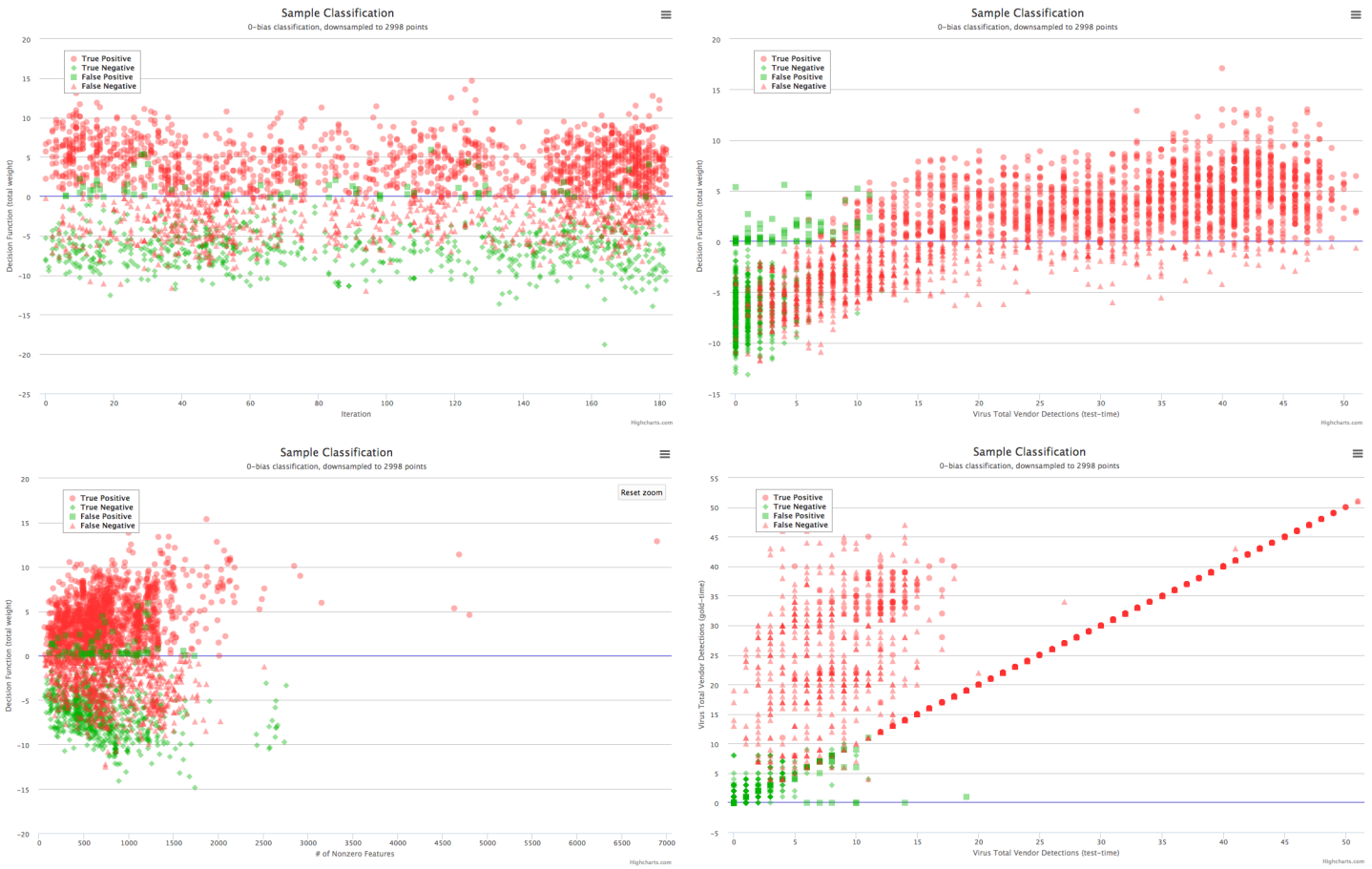


Figure 16: Four of the possible graphs. Time vs weight (top left). # nonzeros vs weight (bottom left). VT Pos - Test vs Weight (top right). VT Pos - Test vs VT Pos - Gold (bottom right)

DataTables.js

Earlier, we mentioned that it takes around 1600ms to load our largest table in memory. In order to reduce our average load time, we use Redis /citeRedis, an in-memory key-value cache. Since we use DataTables multiple times in our application, we write a small backend framework around receiving DataTables API calls in order to speed up table operations across the site.

The DataTables plug-in creates a highly customizable table that automatically makes structured API calls to a specified endpoint. The most important parameters in the API call are:

- column to sort on
- column sort direction (ascending vs descending)
- index of first element to display

- number of elements to display
- search query to filter on

These are the steps that we take in processing the request:

1. Attempt to load the page from Redis using above parameters as a key (saved in step 4).
2. Attempt to load all the data from Redis.
   (a) If not in memory, load from disk, then put in Redis
3. Filter all elements based on search query if one exists
4. Sort data based on column specified in request, up to and including rows on the 10th page
   (a) For example: if the request wants 15 rows starting from index 30, in ascending order on some column, find the 30+(15*10) smallest values with a min-heap, which runs in O(nlog(start+page_size*10)) = O(180)
   (b) Cache all pages past first page in Redis to be loaded in step 1
5. Cache rows in requested view for quick lookup (for explore sample modal, etc)
6. Return rows requested

Since our master node has plenty of RAM, we are able to cache a lot of data in Redis. After our initial load, we only ever have to hit the cache to get the data. Whenever we need to sort on a new column, it takes roughly 1200ms to load the entire table from Redis. Once we sort on a particular column and display a single view, the next 10 views are cached with minimal overhead (roughly 0.3ms per 15-row view). An API request that loads a view from cache takes 18ms on average. Since paging is the most common use-case, overall table usage extremely fast. Although a database may be the most natural way to store this data, we have not yet tested it since this currently works well.

7 Future Work

The web interface is just now starting to take shape but still has a lot of work to be done. Some possible future work includes:

- testing data storage in a database to improve the speed of table queries
- analysis of shifts in the models from iteration to iteration
- allow interactive exploration of dimensionality reduction techniques to aid in feature design
- build a panel to explore samples by most impactful features at the token level
- display timeline per sample of vendor classification and pipeline classification results
- allow input of independent MD5's on the web interface and automatically featurize and classify it
- allow expert input to manually classify samples from the interface

8 Conclusion

We have presented a scalable, general binary classification pipeline that can process over a million samples in more than a hundred temporal iterations in a matter of hours. We designed a modern web user interface alongside the pipeline that can be used to monitor and view the results of pipeline jobs. This interface is designed to allow interactive exploration of samples and features in order to understand the classification decisions that machine learning algorithms make. The interface helps not only a general user of the system glean information from the models but also a developer in verifying and debugging various stages of the pipeline. It is our hope that this pipeline serves as a model for both scalable core classification systems and interactive interface designs for machine learning.

References

[1] Armin Ronacher. Flask. https://pypi.python.org/pypi/Flask, 2015.

[2] INADA Naoki. Messagepack. https://pypi.python.org/pypi/msgpack-python/, 2015.

[3] John Snyder. Collapsible. https://github.com/juven14/Collapsible, 2015.

[4] Jonas Tarnstrom. Ultrajson. https://pypi.python.org/pypi/ujson, 2015.

[5] McAfee Labs. McAfee Labs Threats Report. Technical report, McAfee, August 2014.

[6] SpryMedia Ltd. Datatables. https://www.datatables.net/license/mit, 2015.

[7] Torstein Hnsi. Highcharts.js. http://www.highcharts.com/about, 2015.

[8] VirusTotal. https://www.virustotal.com/en/statistics/. Retrieved on July 30, 2014.

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2012.