Practical 2: Classifying Malicious Software

Writeup due 23:59 on Friday 6 March 2015
Kaggle submission closes at 11:59am on Friday 6 March 2015

You will do this assignment in groups of three. You can seek partners via Piazza. Course staff can also help you find partners. Submit one PDF writeup per team via the Canvas site.

In this practical, you will classify executable files collected from people's computers (over a period of several days) into any of 14 known malware classes, or determine that the executables are not malware. You will train on 3086 executables of known provenance that were collected on a single day. Your predictions on 1900 out of 3724 executables collected on a subsequent day will appear on the public leaderboard, and your predictions on the remaining 1824 executables will appear in the private leaderboard. In making your predictions, you will primarily have at your disposal logs of the system calls (and arguments) made by the processes invoked by the executables when run.

Identifying malware can be tricky because there are often many variants of any particular class of malware that exhibit slightly different behavior. Malware can also behave differently depending on the environment in which it finds itself, or act with some randomness. Being able to classify malware into broader classes is useful, however, because it can both suggest ways of disinfecting infected systems, as well as allow us to easily identify new variants of particular classes that are being introduced. The malware classes under consideration in this practical are: `Agent`, `AutoRun`, `FraudLoad`, `FraudPack`, `Hupigon`, `Krap`, `Lipler`, `Magania`, `Poison`, `Swizzor`, `Tdss`, `VB`, `Virut`, and `Zbot`.

Data Files

There are three files of interest, which can be downloaded from Kaggle:

- `train.tar.gz` and `test.tar.gz` – These file contains information about the 3086 executables in the training set, and the 3724 executables in the test set, respectively. They are gzipped tarballs of directories, which contain an XML file for each datum. The training files have the form

  `<hex_string>.<malware_class>.xml`

  where `hex_string` is a unique identifier, and `malware_class` is either one of the 14 malware classes under consideration or is `None`. For example:

  `fc9b35928deb723b0e0105263d1661e38ad033337.FraudLoad.xml`

  You will use the `malware_class` label in the filename when training. After unzipping `test.tar.gz`, you should also have a directory containing XML files named according to the above convention, except the `malware_class` in the filename has been replaced in each instance with an `X`.

When submitting your predictions, you will use the `hex_string` in the filename as a unique identifier. For example, when predicting on the test file

`ffc47163a530c51ef2e6572d786aefbaed99890f2.X.xml`

you will use `ffc47163a530c51ef2e6572d786aefbaed99890f2` as the `Id` in your submission file. Your prediction for each unique `hex_string` will be an integer between 0 and 14 (inclusive). See the "Evaluation" section for more details.

Each train and test file is a valid XML document containing a log of the executable's execution history as well as some metadata. The XML adheres to the following format:

```
<?xml version="1.0"?>
<processes>
  <process ...>
    <thread>
      <all_section>
        <system_call1 ...>
        </system_call1>
        ... more system calls
      </all_section>
    </thread>
    ... more threads
  </process>
  ... more processes
</processes>
```

That is, the root element is called `processes`, and it contains a list of `process` elements, each corresponding to one of the processes invoked by the executable. Each `process` element may contain some metadata as attributes, and its children are `thread` elements. The execution history of a particular thread is contained in an `all_section` element, which is likely the most important part of the document. The `all_section` element lists the system calls made by the thread (in order) together with various arguments. Note that this will not literally have `system_call1` elements, but the element names will correspond to system calls such as `load_dll` and `create_thread`.

The following is an example `system_call` element from an `all_section`:

```
<load_image \
  filename="c:\342c547b28e9517f6fcf6c703933c0d9.EX" \
  successful="1" address="&#x24;400000" \
  end_address="&#x24;414000" size="81920" \
  filename_hash="hash_error"/>
```

The name of the system call, in this case `load_image`, is given by the tag of the element, and its arguments appear as attributes. The above system call element does not have children, though some system call elements do.

- `sample_predictions.csv` – A sample submission file. You will produce a similar file. The format is comma-delimited, with the first column being the `hex_string` and the second column being your class prediction, an integer between 0 and 14 (inclusive).

```
Id,Prediction
0aefbb082e0461675d05e3147473045acdf2894cb,2
7070018d4360b1b45a6dcb001acc4e463369d2e9f,2
4fcb33dd28a6f88533562958c22f26d5bfbb683b1,2
a2be4cf8927a6f2dbff67a02f9487982511768e21,13
a7abe16d5197f7d2257b81724a241c4b0b3f35bed,13
e44f52dfce3fdef015ee4f77f4564d1e18bc908a9,13
a3b09caec6edcfeb2f3ef321b62a5100a6c2f23f9,2
...
```

The class numbers correspond to the predicted classes, in alphabetical order; see table below. Note that `None` is a special class indicating that the executable is not malware.

Class Distribution

The distribution of malware classes in the training data is approximately as follows. It may be worthwhile to keep in mind that some classes are very infrequent.

0	Agent	3.69%
1	AutoRun	1.62%
2	FraudLoad	1.20%
3	FraudPack	1.03%
4	Hupigon	1.33%
5	Krap	1.26%
6	Lipler	1.72%
7	Magania	1.33%
8	None	52.14%
9	Poison	0.68%
10	Swizzor	17.56%
11	Tdss	1.04%
12	VB	12.18%
13	Virut	1.91%
14	Zbot	1.30%

Evaluation

The evaluation metric for this practical is categorization accuracy. That is, you will be scored on the percentage of the test executables that are correctly classified. In math:

$$\text{Categorization Accuracy} = \frac{\text{Number Correctly Classified Examples}}{\text{Total Number of Examples}}.$$

Sample Code

Two Python files are available from the course website. The file `classification_starter.py` and `util.py` are meant to help you get going. You definitely don't have to use them, but they provide some potentially-useful tools in which you can fill in the gaps. Specifically, it helps you write some functions that can generate features from the data. The file has lots of comments, so hopefully you can figure out how it works. Thanks to Sam Wiseman for putting this together!

Solution Ideas

As in the previous practicals, you have a lot of flexibility in what you might do. You could focus on feature engineering, i.e., coming up with fancy inputs for your method, or you could focus on fancy classification techniques that use the features. Here are some ideas to get you started:

- **Logistic regression on basic features:** A good place to start is to turn the data into a vectorial feature representation, and use a logistic regression technique. You could use quantitative features such as the number of times each system call was made.
- **Use a generative classifier:** You could build a model for the class-conditional distribution associated with each type of malware and compute the posterior probability for prediction.
- **Use a neural network:** If you think there isn't enough flexibility, you could implement a multi-layer perceptron and train it with backpropagation.
- **Use a support vector machine:** If you prefer your objectives convex, you could jump the gun and learn about support vector machines.
- **Go totally Bayesian:** Worried that you're not accounting for uncertainty? You could take a fully Bayesian approach to classification and marginalize out your uncertainty in a generative or discriminative model.
- **Use a decision tree:** If you think a linear classifier is too simple but don't want to train a neural network, you could try a decision tree.
- **Use KNN:** Have a great way to think about similarities between the executables? You could try K nearest neighbors and see how that works.