Malware Classification

Avi Banerjee[1], Chris Tillman[*2],Elizabeth Wong[#3]
San Jose State University
CS 185c Semester Project

***Abstract*— Metamorphic malware is almost undetectable when it comes to traditional signature protection due to their polymorphic tendencies, but made possible using machine learning techniques. Here, we'll apply a combination of Hidden Markov Models with other machine learning techniques in an attempt to detect and properly classify Malware into their appropriate families. We'll be extracting static opcode sequences of our data and running it through an HMM.**

Taking the data, we'll experiment with several different analysis methods and compare results to determine the most appropriate approach for such a task.

***Keywords*— malware, metamorphic malware, classification, Hidden Markov Models, machine learning**

I. Introduction

In the past few years, the demand for the security and protection of software has dramatically increased. The development of malicious software has become a billion dollar industry. Despite the constant push for counter mechanisms, Malware-driven cyber crime continues to be an issue and further research regarding such approaches would be crucial for such security prevention.

Our suggested approach involves extracting op codes from our malware samples and retrieving the appropriate sequence and frequency data to classify such malware into their own respective families. From previous studies, such as in [15], Hidden Markov Models have proven much success compared to previous signature detection methods in metamorphic malware. We'd like to see the effectiveness of such models for classification of malware families

II. Background

In this section, we'll touch up on important topics for the remainder of the paper.

A. Malware

Malware is any software designed with malicious intent. Such activities include damaging or stealing data, stealing resources or even simple ad popups. Malware includes all members and families of different worms, viruses, backdoor, Trojans, spyware and adware. Each family has distinguishing characteristics when it comes to infecting a host computer. Some spread through host files and replicate themselves, while others manage to bypass security check completely. Ideally, people have designed anti-virus software to detect and remove malicious software.

Historically, malicious software was not even conceptualized until the 1980s. However, now more than ever, the malware industry is exponentially growing, with new types of malware everyday.

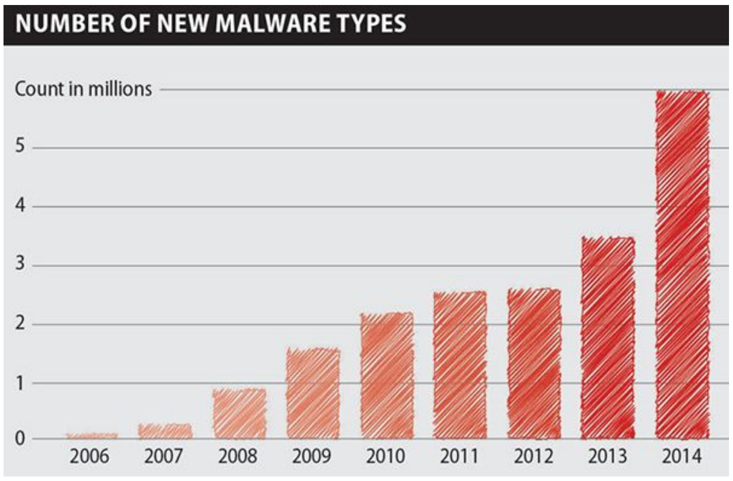


Figure 1: Increased number of new malware throughout the years [15]

B. Malware Detection vs Classification

Often times, it seems once malware detection software proves secure, malware writers come up with another solution to bypass such securities. This arms race has continued from generation to generation. Signature based detection was, and is still, the most common technique in terms of antivirus software. With such a program, the scanner searches through the executable to locate a specific pattern of bits that is common among malware types.

Overtime, malware writers have found a way to bypass these signature scanners by introducing metamorphic malware. Metamorphic malware is extremely hard to detect for common virus scanners. This is because metamorphic malware makes changes to the entire body of the virus, making one completely unlike the other while maintaining the same functions. Since the internal structure of the code changes, each virus signature is different and can easily evade detection.

Anomaly Based Detection and Hidden Markov Model Based Detection both provide alternate solutions to detecting this "undetectable" metamorphic malware [1]. Because of the high rate of false positives with Anomaly Based Detection, we'll be focussing more on Hidden Markov Model Detection and the prospects other machine learning models have in the field of malware classification.

As mentioned above, there are a wide range of types and families of malware. With each behaving so differently, we'd like to not only detect unknown malware samples, but also correctly classify the family of malware.

C. Hidden Markov Models

Markov Models consist of states and transition probabilities between states. The concept of a Hidden Markov Model involves states that cannot be directly observable, the hidden states provide more statistical potential than what can be singularly observed. Hidden Markov Models have become an industry favorite for various speech recognition and text deciphering tasks.

The process for training a Hidden Markov Model requires the use of a *forward-backward* algorithm. It'll help determine the probability of being at a given state q_i at time t, with an observation sequence O. Once trained, we can utilize the Baum-Welch Algorithm to score the model with a test set. With a well constructed HMM, attempting binary classification, we can distinguish a threshold separating the range of scores. Scores above the threshold are classified as one group and scores below are the other.

This success in Hidden Markov Model Detection has inspired us to utilize additional machine learning techniques and compare results between models.

III. Experimental Design

A. Tools and Data Processing

The tools used for these experiments were: ScikitLearn[6], python scripts for data processing, a modified version of Dr. Stamp's hmm used for sanity checks.

The malware dataset chosen for experimentation was from the Microsoft Malware Classification Challenge (BIG 2015) Kaggle competition [4]. This dataset includes thousands of executables from 9 different malware families: {Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, Gatak}. In our project we use the first four of these families. The benign samples came from fresh installation of Cygwin

The instruction sets we used came from the following sources: Wikipedia[12], NASM Intel x86 Assembly Language Cheat Sheet[13], Roger Jegerlehner's code table[14], and the 82 unique opcodes found in one of our virus samples. The challenge here was to determine which mnemonics should be used as observations. We tried the base set from Wikipedia[12], a couple sets merged from looking at instruction groupings in[13] and [14], and the subset derived

The executables were first decompiled into assembler files. We then extracted the opcode mnemonics from these files. We generated observation sequences by mapping each mnemonic in a sample into an integer between 0 and N-1, (where N is the number of mnemonics in the x86 subset). This integer was appended to the sequence. In our test cases, the 82 opcode subset was used.

B. Hidden Markov Models

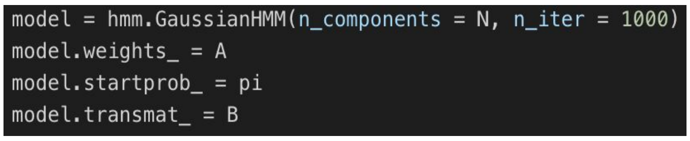
The tests we developed for the HMM used the following procedure.

1) Create training data for each family by appending the opcode sequences from 100 samples into one observation sequence.
2) Train an HMM model using the observation sequence for each family. The following families were used: Ramnit (family 1), Lollipop (family 2), Kelihos_ver3 (family 3), Vundo (family 4), Cygwin (benign). Additionally, train one model on all malware families, using the training opcode sequence from each family to generate the observation sequence.
3) For each of the 4 families and benign family, create 30 test samples, also in the format of an observation sequence.
4) Score each test sequence against each trained HMM model.

Family Name	**# Files in Dataset**	**# Files for training**	**Test Set**
Ramnit	1542	30	10
Lollipop	2477	30	10
Kelihos_ver3	2939	30	10
Vundo	474	30	10
Cygwin	61	30	10

Figure 2: Dataset used in training and testing

To implement this experiment, we chose to use GaussianHMM found in python library scikit-learn hmmlearn[6]. We experimented with using both 2 and 4 hidden states. Our observation set contained 82 symbols. We set N: {2,4}, M: 82. We initialized our A matrix, B matrix and Pi matrix as semi-random stochastic values roughly as 1/N, 1/M, and 1/N respectively.



```
model = hmm.GaussianHMM(n_components = N, n_iter = 1000)
model.weights_ = A
model.startprob_ = pi
model.transmat_ = B
```

Figure 3: Gaussian HMM models using 1000 iterations

The results of this experiment were not quite as expected. Our expectation was that there would be greater differentiation between families when testing malware samples against HMM trained on different malware families. One possibility is that the malware families may have similar opcode frequencies.

C. SVM

Support Vector Machines are good at separating non linear data by taking it into a higher dimensional space and finding a separating hyperplane. The resulting hyperplane will have 1 fewer dimensions than the mapped data.

In the SVM, we test for is a difference in the opcode frequency between malware samples and benign samples. As test data, we started with a vector of size 82, where each data point corresponded to the normalized opcode frequency of the sample. Through Redundant Feature Elimination, we reduced this to 4 meaningful features. These correspond to the frequencies of the MOV, OR, SUB, and PUSH instructions.

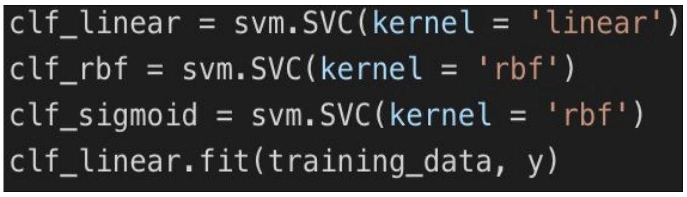
We trained one model on each family (the 4 malware families and 1 benign). Additionally we trained one model on all 4 malware families - intent here is to check to see if it can separate malware samples from benign samples.

The single family tests are done using an SVM trained on a single family, test data comprised of the trained malware family and benign.

The multi-family tests use samples from all 4 malware families and benign ware. The are scored on both the multi-family SVM and on the benign SVM. This is to see if there is any difference in the results.

1) Create training vectors for model training. Each training vector consists of 30 feature vectors (from 30 files) from a particular malware family and 30 feature vectors from the benign family. Each feature vector has the counts of each opcode normalized by the length of the entire files opcode sequence. Additionally, a training vector of weights is created. The malware and benign vectors are given a score of 1 and -1 respectively.

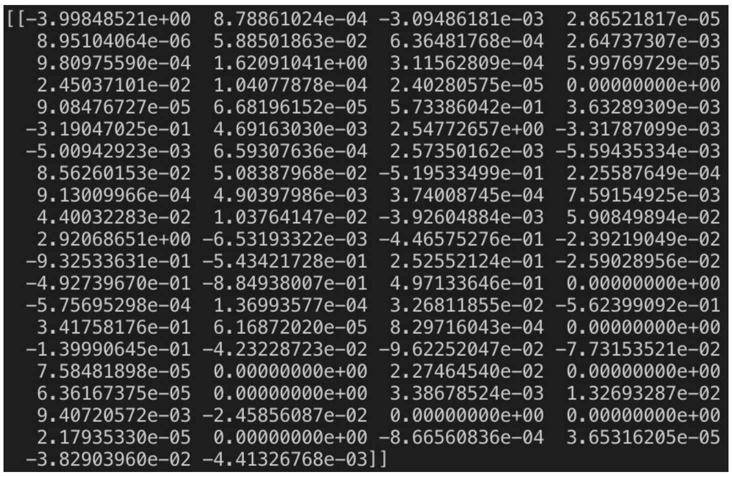
2) We trained 5 different SVMS, similarly to the HMM procedure. One for each family (families 1-4) and benign for binary classification. Additionally train 1 multiclass SVM that we hoped would be able to classify a test file as a specific family. Weights were given (1-5) for each family and benign for the multiclass model.
3) We tested each model with data from each family. We also experimented with different kernel functions including linear, rbf, and sigmoid.



```
clf_linear = svm.SVC(kernel = 'linear')
clf_rbf = svm.SVC(kernel = 'rbf')
clf_sigmoid = svm.SVC(kernel = 'rbf')
clf_linear.fit(training_data, y)
```

Figure 4: SVM code for Recursive Feature Elimination (RFE)

After training our SVMs, we performed Recursive Feature Elimination on our data. Pictured below is the original list of coefficients for a specific model.



```
[[-3.99848521e+00  8.78861024e-04 -3.09486181e-03  2.86521817e-05
   8.95104064e-06  5.88501863e-02  6.36481768e-04  2.64737307e-03
   9.80975590e-04  1.62091041e+00  3.11562809e-04  5.99769729e-05
   2.45037101e-02  1.04077878e-04  2.40280575e-05  0.00000000e+00
   9.08476727e-05  6.68196152e-05  5.73386042e-01  3.63289309e-03
  -3.19047025e-01  4.69163030e-03  2.54772657e+00 -3.31787099e-03
  -5.00942923e-03  6.59307636e-04  2.57350162e-03 -5.59435334e-03
   8.56260153e-02  5.08387968e-02 -5.19533499e-01  2.25587649e-04
   9.13009966e-04  4.90397986e-03  3.74008745e-04  7.59154925e-03
   4.40032283e-02  1.03764147e-02 -3.92604884e-03  5.90849894e-02
   2.92068651e+00 -6.53193322e-03 -4.46575276e-01 -2.39219049e-02
  -9.32533631e-01 -5.43421728e-01  2.52552124e-01 -2.59028956e-02
  -4.92739670e-01 -8.84938007e-01  4.97133646e-01  0.00000000e+00
  -5.75695298e-04  1.36993577e-04  3.26811855e-02 -5.62399092e-01
   3.41758176e-01  6.16872020e-05  8.29716043e-04  0.00000000e+00
  -1.39990645e-01 -4.23228723e-02 -9.62252047e-02 -7.73153521e-02
   7.58481898e-05  0.00000000e+00  2.27464540e-02  0.00000000e+00
   6.36167375e-05  0.00000000e+00  3.38678524e-03  1.32693287e-02
   9.40720572e-03 -2.45856087e-02  0.00000000e+00  0.00000000e+00
   2.17935330e-05  0.00000000e+00 -8.66560836e-04  3.65316205e-05
  -3.82903960e-02 -4.41326768e-03]]
```

Figure 5: SVM Coefficients

RFE left us with just 4 dominant features. These correspond to the frequencies of the MOV, OR, SUB, and PUSH instructions.

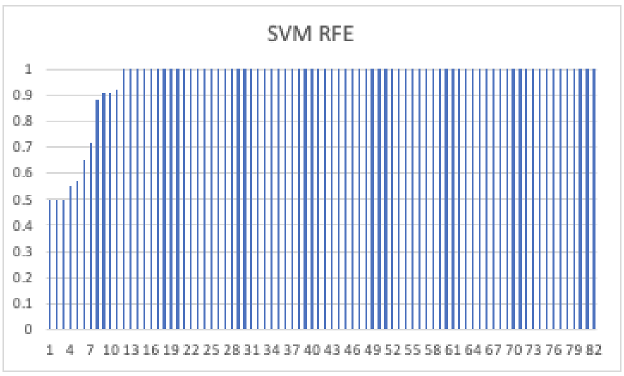
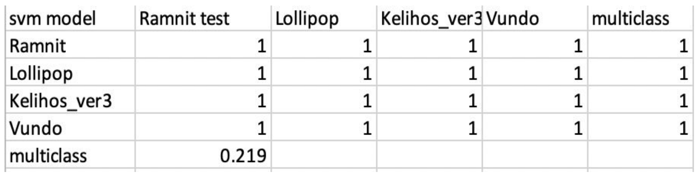


Figure 6: SVM accuracy with 1-82 features

We calculated the accuracy with different feature numbers and settled on 10 including the 4 dominant ones.



svm model	Ramnit test	Lollipop	Kelihos_ver3	Vundo	multiclass
Ramnit	1	1	1	1	1
Lollipop	1	1	1	1	1
Kelihos_ver3	1	1	1	1	1
Vundo	1	1	1	1	1
multiclass	0.219				

Figure 7: Scores for different SVM models

The results of our SVM testing are shown above. For the binary classification tests every model performed with 100% accuracy. However, when we attempted to classify specific malware families our results were poor. With a linear kernel our model classified 21.9% of our test set correctly. For rbf and sigmoid we got 20.4% and 17.7% respectively. Our suspicion is that the malware families we were testing are rather similar which made it difficult for the SVM to find a clear hyperplane between the different families.

D. ANN

We chose to use an Artificial Neural Network for its ability to find associations within a large data set. Luckily we had plenty of data accessible and hoped that by tuning the number of layers and their depths, we would get much better results than with the SVM. For our ANN, we used the exact same malware testing and training data as with the SVM but did not include the benign files.

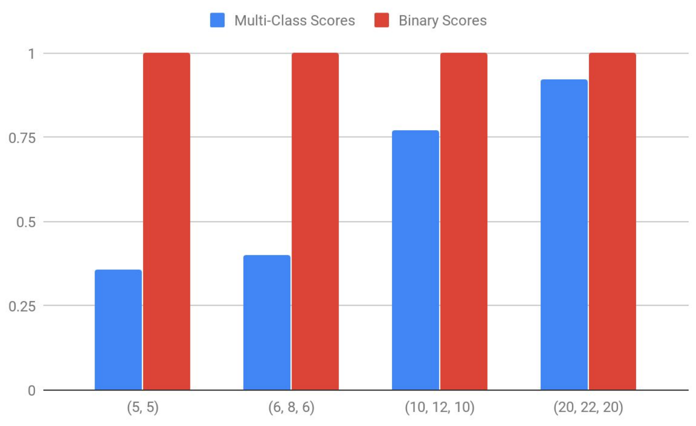
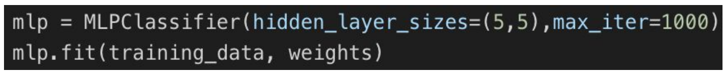


Figure 8: Scores for Different Layer/Depth ANN with(x, y) representing two layers with depth x, depth y



```
mlp = MLPClassifier(hidden_layer_sizes=(5,5),max_iter=1000)
mlp.fit(training_data, weights)
```

Figure 9: Construction of MLP

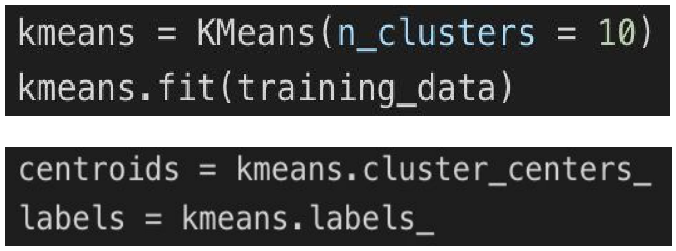
Our experimentation with the ANN yielded much better results than the SVM. For binary classification, just 2 layers with depths of 5 and 5 were sufficient. With 3 layers and depths of 20, 22, and 20, we achieved 94% accuracy with multi-family classification. Initially we had chosen a max iteration size of 500 however this was not enough for the model to converge. Increasing it to 1000 sufficed. Our choice of the number of layers and depth of each was semi random. We experimented with different combinations and were surprised to find that incrementally increasing the depth didn't always improve the performance. Testing many combinations of the hyperparameters was necessary in getting our results.

E. K-Means Clustering

Clustering is a good method for analyzing data in terms of similar grouping rather than binary classification.

The question we try to answer with clustering is whether or not a sample can be identified by family based on opcode frequencies and the SVM single family binary classifications.

Here, our sample data is a vector containing the normalized 82 opcodes counts (same as SVM malware data) and the 5 results from the single family SVMs.



```
kmeans = KMeans(n_clusters = 10)
kmeans.fit(training_data)
```

```
centroids = kmeans.cluster_centers_
labels = kmeans.labels_
```

Figure 10: Construction of Clustering Model

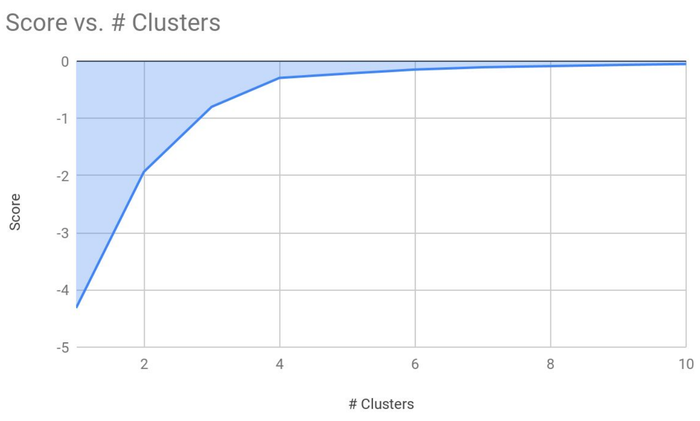


Figure 11: Resulting scores for k = 1 to k = 10

F. Conclusion

This project provided us with a tremendous amount of practical experience applying what we have learning in CS185c to a very relevant real-world problem. We were challenged to design a series of complicated and nuanced experiments and learned about everything from data cleaning to hyper-parameter tuning. We were very surprised that the HMM didn't perform as well as we had expected. It's possible that data similarity led the SVM's multiclass hyperplane to perform weakly as well. The fact that the score for clustering decreased significantly under 4 clusters tells us that there must still be enough variance to distinguish between families, requiring more that many clusters. Perhaps this is why the ANN was able to perform well given enough data, layers and depth.

G. *Challenges and Pitfalls*

a. Data

Clean data sets can be hard to come by. We were fortunate in finding [4] and in having access to Malicia [11] when [4] became unavailable. Even good sets can have arcane documentation, unlabeled samples, or simply be voluminous enough to cause confusion. Having a good data set is critical. The data set ultimately drives experiment design.

b. Tools

There are several choices available. Everything between using already developed packages, adapting pre-existing code, writing tools from scratch, to using a combination. There are tradeoffs: development time vs control, time spent learning how to use a package vs time spent testing a written tool. Each choice made in tool selection will have an impact on the process of data analysis. Sometimes the best tool for the job is ignored because the learning curve was too steep. Among the tools we sadly could not use were: IDAPro [10], TensorFlow [7], Jupyter Notebook [8], Pandas [9].

c. Environmental Factors

We ran into the challenge of trying to process windows virus executables on a windows machine. It is both frustrating and reassuring to watch anti-virus software delete samples as quickly as they are decompressed, and to verify that popular cloud services do not allow for the distribution of viruses through download. The correct solution would have been to use a virtual machine instead of a USB stick

IV. Acknowledgements

We would like to thank Dr. Stamp and Fabio Di Troya, without whom none of this would have been possible.

Dr. Stamp, thank you for opening the door to machine learning. Thank you for being there to generously answer our questions, including the ones we lacked experience to ask. You gave us a strong foundation for continued exploration of machine learning and data analysis.

Fabio, sifting through data is a lot like trying to find needles in a haystack. Thank you for providing: a haystack, a leaf blower, a magnet, and a metal detector. We may not have found needles, but the search has been highly educational.

References

[1] Annachhatre, Chinmayee, et al. "Hidden Markov Models for Malware Classification." Journal of Computer Virology and Hacking Techniques, vol. 11, no. 2, 2014, pp. 59–73., doi:10.1007/s11416-014-0215-x.

[2] Kalbhor, Ashwin, et al. "Dueling Hidden Markov Models for Virus Analysis." Journal of Computer Virology and Hacking Techniques, vol. 11, no. 2, 2014, pp. 103–118., doi:10.1007/s11416-014-0232-9.

[3] Kapratwar, Ankita, et al. "Static and Dynamic Analysis of Android Malware." Proceedings of the 3rd International Conference on Information Systems Security and Privacy, 2017, doi:10.5220/0006256706530662.

[4] Ronen, et al. "Microsoft Malware Classification Challenge." [Astro-Ph/0005112] A Determination of the Hubble Constant from Cepheid Distances and a Model of the Local Peculiar Velocity Field, American Physical Society, 22 Feb. 2018, arxiv.org/abs/1802.10135.

[5] Wong, Wing. (2006), "Analysis and Detection of Metamorphic Computer Viruses." Master's Projects. Paper 153.

[6] Scikit-learning. https://scikit-learn.org/stable/#

[7] Tensorflow. https://www.tensorflow.org/

[8] Jupyter. http://jupyter.org/index.html

[9] pandas. http://pandas.pydata.org/

[10] IDA. https://www.hex-rays.com/products/ida/index.shtml

[11] Stamp, Mark, and Fabio Di Troia. "Malicia Data Set."

[12] "X86 Instruction Listings." *Wikipedia*, Wikimedia Foundation, 24 Oct. 2018, https://en.wikipedia.org/wiki/X86_instruction_listings

[13] NASM Intel x86 Assembly Language Cheat https://www.ssucet.org/~jhudson/14/etec3701/nasmcheatsheet.pdf

[14] "Intel Assembler 80x86 CodeTable." Intel Assembler CodeTable 80x86 - Overview of Instructions (Cheat Sheet), http://www.jegerlehner.ch/intel/

[15] "New Malware Strains Recorded During Second Half of 2014 More Than Double in Volume." Remove Spyware & Malware with SpyHunter - Enigma Software Group USA LLC, 21 May 2015, www.enigmasoftware.com/malware-strains-recorded-h2-2014-more-than-double-volume/.

[16] Cygwin. Retrieved from http://www.cygwin.com/