MALICIOUS URL CLASSIFICATION AND DETECTION USING MACHINE LEANRING

Abstract:

Malicious URL, a.k.a. malicious website, is a common and serious threat to cybersecurity. Malicious URLs host unsolicited content (spam, phishing, drive-by exploits, etc.) and lure unsuspecting users to become victims of scams (monetary loss, theft of private information, and malware installation), and cause losses of billions of dollars every year. It is imperative to detect and act on such threats in a timely manner. Traditionally, this detection is done mostly through the usage of blacklists. However, blacklists cannot be exhaustive, and lack the ability to detect newly generated malicious URLs. To improve the generality of malicious URL detectors, machine learning techniques have been explored with increasing attention in recent years. In this project I would like to detect and classify the URL into categories like phishing/spam websites.

Introduction:

URL is the abbreviation of Uniform Resource Locator, which is the global address of documents and other resources on the World Wide Web. A URL has two main components: (i) protocol identifier, it indicates what protocol to use, (ii) resource name, it specifies the IP address or the domain name where the resource is located. The protocol identifier and the resource name are separated by a colon and two forward slashes. An example is shown in Figure 1.

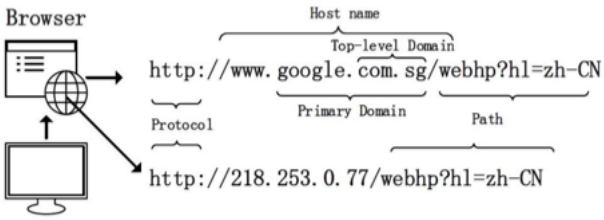


Fig. 1. Example of a URL - "Uniform Resource Locator"

Compromised URLs that are used for cyberattacks are termed as malicious URLs. A Malicious URL or a malicious web site hosts a variety of unsolicited content in the form of spam, phishing, or drive-by-exploits in order to launch attacks. Unsuspecting users visit such web sites and become victims of various types of scams, including monetary loss, theft of private information (identity, credit-cards, etc.), and malware installation. Effective systems to detect such malicious URLs in a timely manner can greatly help to counter large number of and a variety of cyber-security threats. The most common method to detect malicious URLs deployed by many antivirus groups is the black-list method. Black-lists are essentially a database of URLs that have been confirmed to be malicious in the past. This database is compiled over time (often through crowd-sourcing solutions, e.g. PhishTank, as and when it becomes known that a URL is malicious. However, it is almost impossible to maintain an exhaustive list of malicious URLs, especially since new URLs are generated every day. Attackers use creative techniques to evade blacklists and fool users by modifying the URL to "appear" legitimate via obfuscation. Once the

URLs appear legitimate, and user's visit them, an attack can be launched. To overcome these issues, in the last decade, researchers have applied machine learning techniques for Malicious URL Detection.
Machine Learning approaches, use a set of URLs as training data, and based on the statistical properties, learn a prediction function to classify a URL as malicious or benign. This gives them the ability to generalize to new URLs unlike blacklisting methods.
The primary requirement for training a machine learning model is the presence of training data. In the context of malicious URL detection, this would correspond to a set of large number of URLs.
After the training data is collected, the next step is to extract informative features such that they sufficiently describe the URL and at the same time, they can be interpreted mathematically by machine learning models. For example, simply using the URL string directly may not allow us to learn a good prediction model. Thus, one would need to extract suitable features based on some principles or heuristics to obtain a good feature representation of the URL. This may include lexical features (statistical properties of the URL string, bag of words, n-gram, etc.), host based features (WHOIS info, geo-location properties of the host, etc.). The quality of feature representation of the URLs is critical to the quality of the resulting malicious URL predictive model learned by machine learning. Finally, using the training data with the appropriate feature representation, the next step in building the prediction model is the actual training of the model. There are plenty of classification algorithms can be directly used over the training data (Naive Bayes, Support Vector Machine, Logistic Regression, etc.)

Methods (Code and Documentation):

Dataset:

The dataset I received is 500,000 URLs from Virus Total among which 375,786 benign urls (no hits in VT) and 124,214 malicious urls (7+ hits in VT) are definitely malicious.

	A	B	C
1	url	malicious	
2	http://www.facebook-log1in.com/	1	
3	http://getir.net/yg4t	1	
4	http://crosscitydental.com/	0	
5	http://www.tenttrails.com/jscripts/product.js	0	
6	http://103.224.193.105/	0	
7	http://www.utopiapalmsandcycads.com/images/caryotagigas_002	0	
8	http://www.nephilimmerch.com/	0	
9	http://santostefano.sextantio.it/wp-content/plugins/gk-tabs/gk-ta	0	
10	http://support.itum.no/	0	

Feature Engineering:

A URL can be used to generate an extensive feature list up to 3000. The feature I focused on in this project are lexical, domain and host based. The figure 2 below illustrates all the features I generated from the URL to train my model.

Features	Range
DotsInUrl	1, 45
countdelim	1, 93
DotsInSubdomain	0,25
Presence_Of_hyphen	0, 16
URLLength	14,512
Presence_Of_AT	0,0
Presence_Of_double_slash	0,1
No_Of_Subdir	1,15
No_Of_SubDomain	0,26
DomainLength	6,250
No_Of_Queries	0,480
isIP	1,15
Presence_Of_Suspicious_TLD	0,1
Presence_Of_Suspicious_Domain	0,0
No_Of_special_chars	5,165
ratio_special_chars	0.01,0.4
isMultipleDomains	0,2
topLevelDomainCount	0,2
suspicousWords	0,1
digitsInDomain	0,1
isLongURL	0,2
getEntropy	1.03,3.12
getKLDivergence	0.05,2.71
nameserver_count	0,13
URL Redirect Count	-110

The above features can be classified into the below three categories.

- Semantic features (only from name of URL itself)
- Meta features (from external sources: WHOIS, etc.)
- Host Features (number of redirections.)

Sample methods implemented to generate the above features

```
#method to count subdomains
def countSubDomain(subdomain):
    if not subdomain:
        return 0
    else:
        return len(subdomain.split('.'))
#method to count queries
def countQueries(query):
    if not query:
        return 0
    else:
        return len(query.split('&'))

#method to count special_chars
def special_chars(url):
    count=0
    for i in url:
        if not i.isalnum():
                count=count+1
    return count

#method to calculate ratio of special_chars
def ratio_special_chars(url):
    count=special_chars(url)
    return float(count)/float(len(url))
```

To generate host and whois data features I ran a PHP script that connects to the whois server and stores it in my AWS RDS instance

```
db_connection =  pymysql.connect(host="urlwhois.cfug8tk0icxc.us-east-2.rds.amazonaws.com",    # My AWS Hostname
                    user="urlwhois_master",         #  username
                    passwd="urlwhois_master",  #  password
                    db="urlwhois_db1")
whoisDF=pd.read_sql('SELECT d.nameserver_count, u.url  from url_domain d , url_info u where d.domain_id=u.url_domain_fk
FinalWhois=whoisDF.fillna(0)
FinalWhois.head()
```

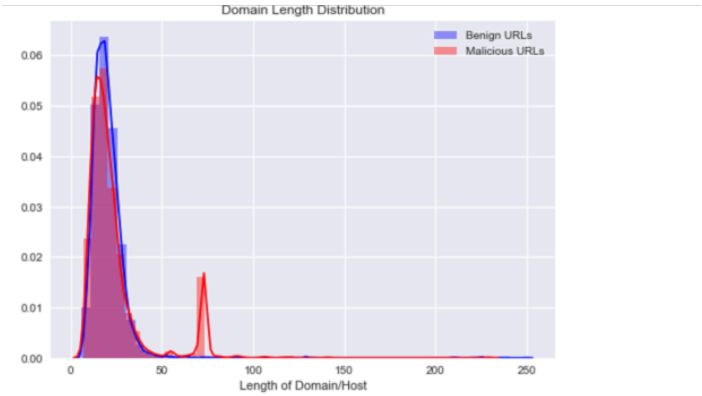
	nameserver_count	url
0	2.0	http://www.facebook-log1in.com/
1	2.0	http://getir.net/yg4t
2	2.0	http://crosscitydental.com/
3	2.0	http://www.tenttrails.com/jscripts/product.js
4	0.0	http://103.224.193.105/

```
rcDF=pd.read_sql('SELECT url_redirect_count, url FROM urlwhois_db1.url_info ', con=db_connection)
finalRC=rcDF.fillna(0)
finalRC.tail()
```

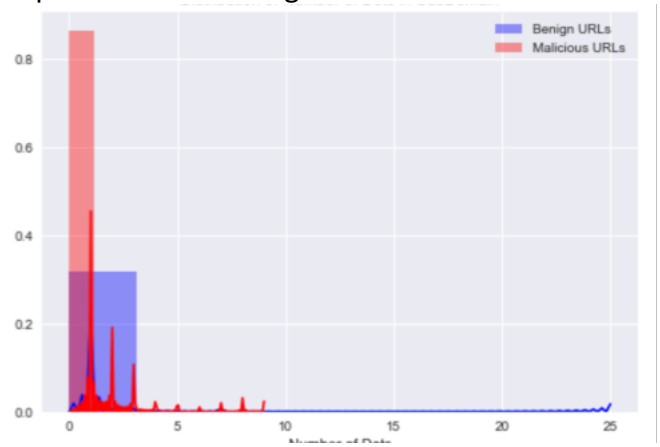
Combine Lexical, Host and Domain Features into one dataframe

```
allFeatures=pd.merge(hostFeatures, featureExtract, how='inner', on='url')
allFeatures.head()
```

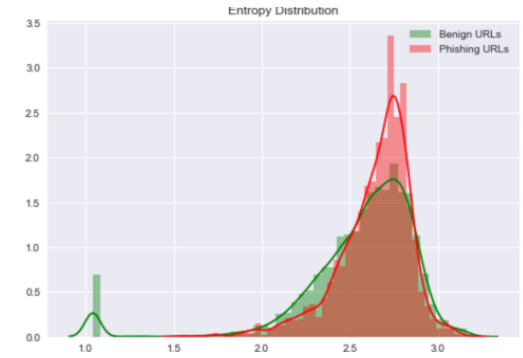
I combined all the features generated into one dataframe to perform Exploratory data analysis and identify the relationship between the features and the benign/malicious URLS.
Some of the key features I was able to identify are



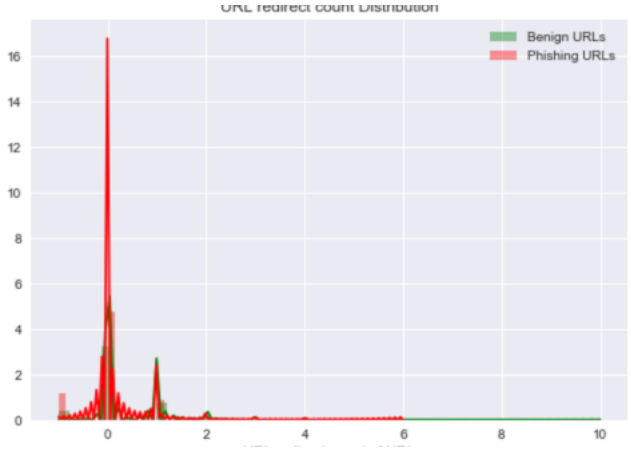
From the above plot we can see that for domain length under 50 there is a significant overlap of malicious and benign urls. However at higher domain length i.e. 75 or above there are more malicious urls in comparison to benign. This clearly indicates the domain length has a key impact on determining whether the URL is malicious or not.



Number if dots in a subdomain is significantly higher for malicious URL Vs benign also making it a key feature



Entropy of l is also taken as a key predictor and the above plot shows a malicious URL tend to have a higher entropy range in comparison to the benign urls.



Another key distinguishing feature is the URL redirect count. Malicious URLS have a -1 and sometimes lower redirect counter. The dataset is 6 months old and majority of the urls are shutdown. Although is not a valid indicator in providing accuracy for the model but is also providing a deviation in learning.

Classification:

Here is the initial result of using all the features to train classifier I choose relevant for this

```
results = {}
for algo in model:
    clf = model[algo]
    clf.fit(X_train,y_train)
    score = clf.score(X_test,y_test)
    print ("%s : %s " %(algo, score))
    results[algo] = score
```

```
DecisionTree : 0.822734101152
RandomForest : 0.844266399599
GNB : 0.544817225839
LogisticRegression : 0.807210816224
```

Model	Accuracy
DecisionTree	82.60%
RandomForest	84.40%
GNB	54.40%
LogisticRegression	80.70%

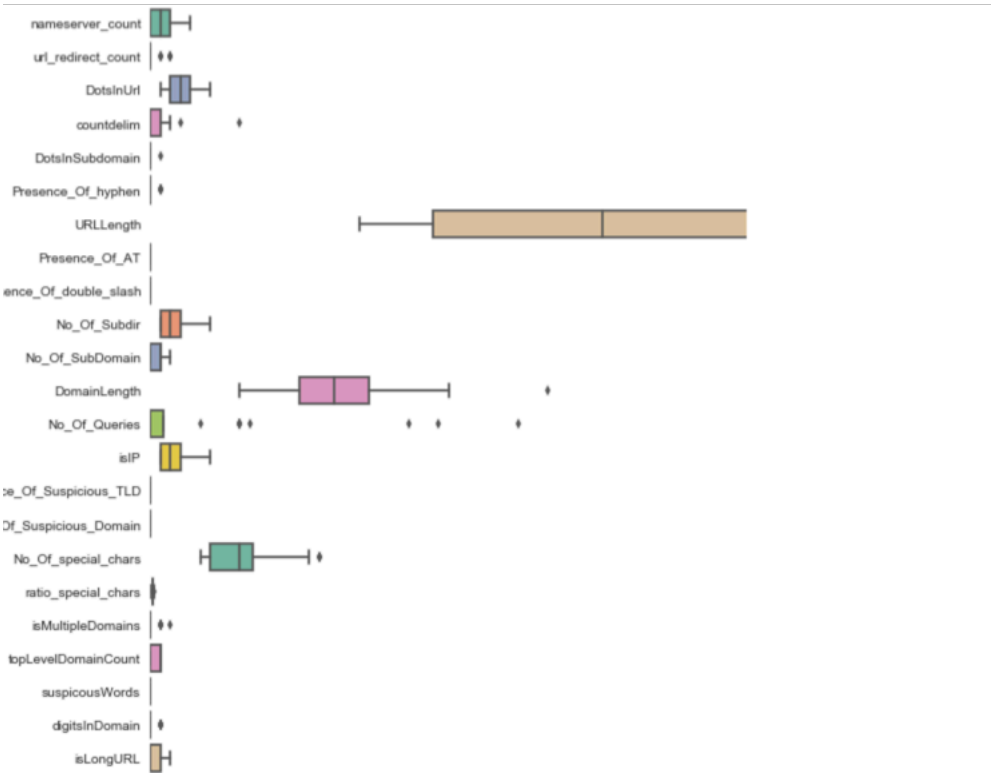
I noticed that using all the features did help achieve 80% accuracy for majority of the classifiers except for Naïve Bayes. I further researched on the classifiers and derived below advantages and drawbacks

Model	Features	Drawbacks
Decision Tree Classifier	Decision tree learning uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).	Create over-complex trees that do not generalize the data well. Requires pruning of trees
RandomForest	Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.	Overfitting for some datasets with some noisy classification tasks
NaiveBayes	Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes is a good tool. It is fast, robust and relatively insensitive to missing values and even data imbalance problems.	When an attribute is continuous, computing the probabilities by the traditional method of frequency counts is not possible. Attribute independence
Logistic Regression	Logistic regression measures the relationship between categorical dependent variable and one or more independent variables by estimating probablities using a logistic function	Problem of Overfitting

Naive Bayes seems most suited algorithm given the fact that the URL features can have null or missing values. In order to further improve accuracy of the NaiveBayes algorithm I choose to remove the features that are not relevant in identifying malicious URL's.

Converting Features to Predictors:

As a first step in identifying the relevant features I generated a boxplot to determine the dependencies among the features. The range of the dataset is highly unscaled. As an example the URL length range for benign and malicious range 14-512 while the kl Divergence and entropy are in range of 1-3 and 0.05,2.71



The features are not properly scaled and there are some dependent and independent features. Although it is not required or mandatory to scale all the elements I attempted to scale some features to help understand the dependencies ranges and the distributions when URL is malicious.

After scaling the features I used RandomForest Classifier to identify the predictors which have high weightage

```
high_info_gain = RFm.feature_importances
index_ig = np.argsort(high_info_gain)[::
index_ig
```

```
array([21, 20, 14, 10,  6, 13,  9, 12,
       18, 19, 15,  4,  7])
```

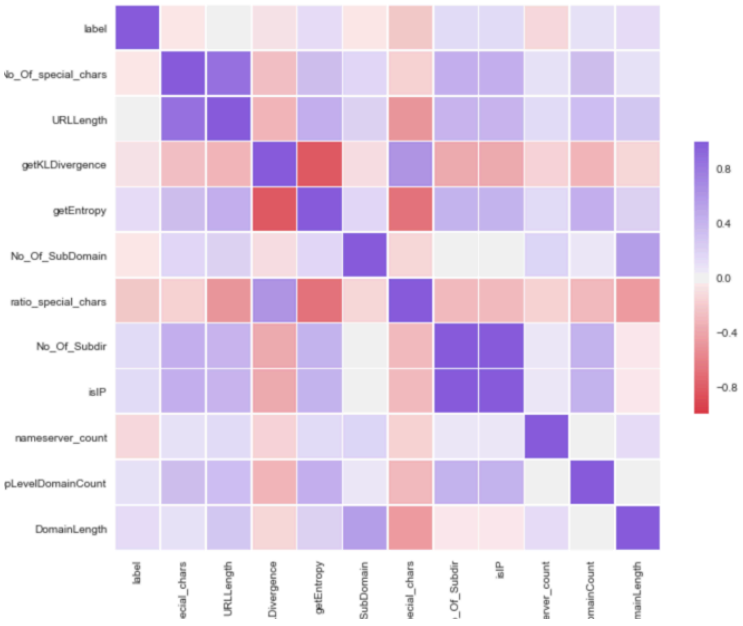
```
temp= allFeatures.drop(['url','label','Pr
featuresOnly=allFeatures.drop(['url','Pre

predictor_names=temp.columns.tolist()
predictor_names
```

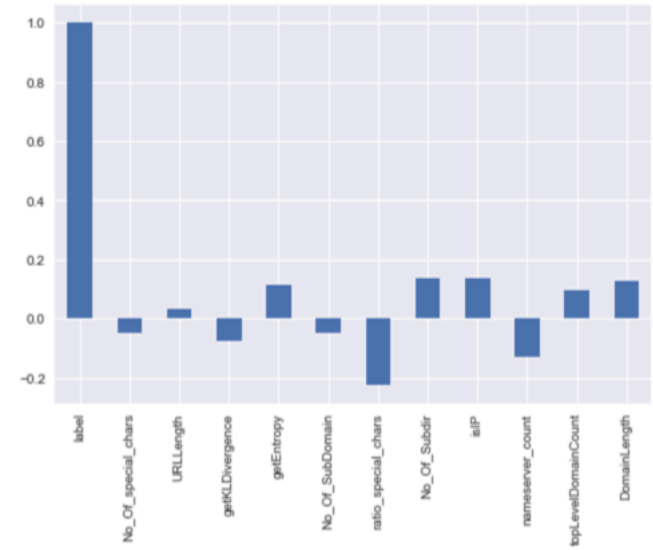
```
['nameserver_count',
 'url_redirect_count',
 'DotsInUrl',
 'countdelim',
 'DotsInSubdomain',
 'Presence_Of_hyphen',
 'URLLength',
 'Presence_Of_double_slash',
 'No_Of_Subdir',
 'No_Of_SubDomain',
 'DomainLength',
 'No_Of_Queries',
 'isIP',
 'No_Of_special_chars',
 'ratio_special_chars',
 'isMultipleDomains',
 'topLevelDomainCount',
 'suspicousWords',
 'digitsInDomain',
 'isLongURL',
 'getEntropy',
 'getKLDivergence']
```

I also used a function to rank the predictors. The idea here is that the best predictors have the best separation between the ranges of the distributions. The worst, doesn't have good separation between malicious and benign URLs.

```
Feature ranking:
1. The feature 'getKLDivergence'     has a Information Gain of 0.118397
2. The feature 'getEntropy'     has a Information Gain of 0.116142
3. The feature 'ratio_special_chars'     has a Information Gain of 0.110657
4. The feature 'DomainLength'     has a Information Gain of 0.106932
5. The feature 'URLLength'     has a Information Gain of 0.093771
6. The feature 'No_Of_special_chars'     has a Information Gain of 0.057034
7. The feature 'No_Of_SubDomain'     has a Information Gain of 0.052635
8. The feature 'isIP'     has a Information Gain of 0.043149
9. The feature 'No_Of_Subdir'     has a Information Gain of 0.041936
10. The feature 'nameserver_count'     has a Information Gain of 0.041012
11. The feature 'url_redirect_count'     has a Information Gain of 0.037323
12. The feature 'DotsInUrl'     has a Information Gain of 0.032823
13. The feature 'countdelim'     has a Information Gain of 0.027940
14. The feature 'suspicousWords'     has a Information Gain of 0.027278
15. The feature 'No_Of_Queries'     has a Information Gain of 0.021728
16. The feature 'Presence_Of_hyphen'     has a Information Gain of 0.015919
17. The feature 'topLevelDomainCount'     has a Information Gain of 0.015907
18. The feature 'digitsInDomain'     has a Information Gain of 0.014770
19. The feature 'isLongURL'     has a Information Gain of 0.010573
20. The feature 'isMultipleDomains'     has a Information Gain of 0.008291
21. The feature 'DotsInSubdomain'     has a Information Gain of 0.005280
22. The feature 'Presence_Of_double_slash'     has a Information Gain of 0.000502
```

After obtaining the feature ranking as above I plotted the correlation of features with each other and to the labels (malicious or beign)as illustrated in below





Now that I have the subset of the original feature set. I used this to train the NaiveBayes classifier and achieved an improved accuracy of 68% with an F-1 score of 62% and a Precision score of 61%

```
# setting Naive Bayes classifier
nb = GaussianNB()

#dividing data to have a training and a testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= .4, random_state=0)

# Naive Bayes cross validation
Kfold = KFold(len(featuresOnly_ranked), n_folds=5, shuffle=False)
print("KfoldCrossVal mean score using Naive Bayes is %s" %cross_val_score(nb,X,y,cv=10).mean())

# Naive Bayes metrics
NBm = nb.fit(X_train, y_train)

y_pred = NBm.predict(X_test)

print(f1_score(y_test, y_pred, average="macro"))
print(precision_score(y_test, y_pred, average="macro"))
print(recall_score(y_test, y_pred, average="macro"))
print("Accuracy score using Naive Bayes is %s" %metrics.accuracy_score(y_test, y_pred))
```

```
KfoldCrossVal mean score using Naive Bayes is 0.68926942975
0.620850208401
0.617343711312
0.644884078383
Accuracy score using Naive Bayes is 0.681772658988
```

Results:

A few tests performed on the trained model from known malicious URL from PhishTank was able to generate desired results.

```
0]: for algo in model:
        print ("%s " %(algo))
        clf = model[algo]
        result =pd.DataFrame(columns=('url','DotsInUrl','countdelim','DotsInSubdomain','Presence_Of_hyphen','URLLength','Pr
    'Presence_Of_double_slash','No_Of_Subdir','No_Of_SubDomain','DomainLength','No_Of_Queries','Presence_Of_Suspicious_TLD'
    'Presence_Of_Suspicious_Domain','No_Of_special_chars','ratio_special_chars','isMultipleDomains','topLevelDomainCount','
     'digitsInDomain','isLongURL','getEntropy','getKLDivergence' ,'label'))
        results = getFeaturesLexicalTest('http://clubdasjoaninhas.com/Sip/Indexxatt.htm', '1')
        result.loc[0] = results
        result['isip']=0
        result['nameserver_count']=2
        result['url_redirect_count']=3
        result = result.drop(['url','label'],axis=1).values
        print(clf.predict(result))
```

```
DecisionTree
[1]
RandomForest
[1]
GNB
[1]
LogisticRegression
[1]
```

Additional experimentation with Logistic Regression:

In an attempt to learn how to vectorize a URL and use TF-IDF and bag of words features I used just the TF-IDF weights a feature for logistic regression. This feature alone surprisingly results in 90% accuracy.

```
]:
##following code is creating a vectorizer from url data.
df=pd.read_csv(path +"sample_500k.csv")
#X_train, X_test, y_train, y_test = train_test_split(data, data["malicious"], test_size=0.1, random_state=42)
vectorizer = TfidfVectorizer(tokenizer=getTokens) #get a vector for each url but use our customized tokenizer
X = vectorizer.fit_transform(df["url"])
y = df["malicious"]
```

```
]:
#print(X.data)
#print(X)
```

```
]:
lr = LogisticRegression()

#dividing data to have a training and a testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= .4, random_state=0)

# Logistic regression cross validation
#Kfold = KFold(len(ranked_predictors), n_folds=5, shuffle=False)
#print("KfoldCrossVal mean score using Logistic regression is %s" %cross_val_score(lr,X,y,cv=10).mean())

# Logistic regression metrics
LRm = lr.fit(X_train, y_train)
print(LRm.score(X_test,y_test))
#LRm.predict_proba(X_test)  # The returned estimates for all classes are ordered by the label of classes
```

```
0.906985
```

Deep learning and implementing a simple LSTM model:

About 80% of the work that goes into machine learning based cyber-attack detection is focused on feature engineering – identifying what values to extract from binaries, network packets, and other signals to use as inputs to our machine learning systems. In order to further improve detection capabilities and improve performance deep neural networks are used. Researchers are focusing on designing deep neural networks which automatically learn to extract features, identifying attributes of the data that lead to better detection performance than we could have achieved otherwise.

The computational flow involves convert an input URL string which is encoded using a method called embedding, features are detected through learned pattern matchers (convolutions), and then successive layers of neurons (a feed forward network) compute the probability the URL is malicious based on these features.

I implemented a method to convert raw URL string in list of lists where characters are stored encoded as integer. This includes digits, letters, punctuation and whitespaces. Used a simple LSTM model with sigmoid activation function and loss function as binary_crossentropy.

I ran 3 epochs and achieved an accuracy of 75% with a 10,000 record input.

```
## Deep Learning model Definition --- A --- (Simple LSTM)


def simple_lstm(max_len=512, emb_dim=32, max_vocab_len=100, lstm_output_size=32, W_reg=regulari
    # Input
    main_input = Input(shape=(max_len,), dtype='int32', name='main_input')
    # Embedding layer
    emb = Embedding(input_dim=max_vocab_len, output_dim=emb_dim, input_length=max_len,
                dropout=0.2, W_regularizer=W_reg)(main_input)

    # LSTM layer
    lstm = LSTM(lstm_output_size)(emb)
    lstm = Dropout(0.5)(lstm)

    # Output layer (last fully connected layer)
    output = Dense(1, activation='sigmoid', name='output')(lstm)

    # Compile model and define optimizer
    model = Model(input=[main_input], output=[output])

    sgd= SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)

    #adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
    #model.compile(optimizer=adam, loss='binary_crossentropy', metrics=['accuracy'])
    model.compile(optimizer=sgd, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

```
# Fit model and Cross-Validation, ARCHITECTURE 1 SIMPLE LSTM
nb_epoch = 3
batch_size = 32

model = simple_lstm()
model.fit(X_train, target_train, nb_epoch=nb_epoch, batch_size=batch_size)
loss, accuracy = model.evaluate(X_test, target_test, verbose=1)

print('\nFinal Cross-Validation Accuracy', accuracy, '\n')
print_layers_dims(model)
```

```
Epoch 1/3
7498/7498 [==============================] - 221s 30ms/step - loss: 0.6066 - acc: 0.7531
Epoch 2/3
7498/7498 [==============================] - 218s 29ms/step - loss: 0.5595 - acc: 0.7559
Epoch 3/3
7498/7498 [==============================] - 216s 29ms/step - loss: 0.5575 - acc: 0.7559
2500/2500 [==============================] - 13s 5ms/step

Final Cross-Validation Accuracy 0.7568

<keras.engine.topology.InputLayer object at 0x119812898>
Input Shape:  (None, 512) Output Shape:  (None, 512)
<keras.layers.embeddings.Embedding object at 0x1198129b0>
Input Shape:  (None, 512) Output Shape:  (None, 512, 32)
<keras.layers.recurrent.LSTM object at 0x119812908>
Input Shape:  (None, 512, 32) Output Shape:  (None, 32)
<keras.layers.core.Dropout object at 0x11a02db00>
Input Shape:  (None, 32) Output Shape:  (None, 32)
<keras.layers.core.Dense object at 0x11a02df28>
Input Shape:  (None, 32) Output Shape:  (None, 1)
```

Future Scope:

- Improve the accuracy and F-1 score of the model further. Train the classifier to give a score between 0-1 (not only prediction). This score would be an indication of how close it is to being a malicious Vs benign URL.
- Implement Deep Neural Network model with inputs from
    - https://arxiv.org/pdf/1702.08568.pdf

References:

- Malicious URL Detection using Machine Learning
  - https://arxiv.org/pdf/1701.07179.pdf
- Using Machine Learning to Detect Malicious URLs
  - https://www.kdnuggets.com/2016/10/machine-learning-detect-malicious-urls.html
- Detecting malicious URLs using machine learning techniques
  - http://ieeexplore.ieee.org/document/7850079/
- Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs.
  - http://www.cs. berkeley.edu/~jtma/papers/beyondbl-kdd2009.pdf
- Code References: GITHUB examples.
  - https://github.com/nikbearbrown/NEU_COE/blob/master/CSYE_7245/Week_3/