Visualisation of Malicious & Benign Webpages Dataset

Basic Initialization

In [1]:

```
#Installing mandatory libraries

#!pip install geonamescache
#!pip install palettable
#!pip install -U textblob
#!pip install cufflinks
#!pip install seaborn
#!pip install plotly
#!pip install -c plotly plotly-orca
```

In [2]:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
# Common imports
import pandas as pd
import numpy as np
import time
import os
import sklearn
import seaborn as sns
import warnings
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from random import randrange
#Disabling Warnings
warnings.filterwarnings('ignore')
# to make this notebook's output stable across runs
np.random.seed(42)
# To plot figures
%matplotlib inline
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
sns.set_palette(['green','red'])#Fixing the Seaborn default palette
```

Loading Dataset

In [3]:

```
# Load Datasets
def loadDataset(file_name):
    df = pd.read_csv(file_name)
    return df

df_train = loadDataset("Webpages_Classification_train_data.csv")
df_test = loadDataset("Webpages_Classification_test_data.csv")
#Ensuring correct sequence of columns
df_train = df_train[['url','ip_add','geo_loc','url_len','js_len','js_obf_len','tld','who_is','https','content','label']]
df_test = df_test[['url','ip_add','geo_loc','url_len','js_len','js_obf_len','tld','who_is','https','content','label']]
```

Details of Dataset: Tabular

The Dataset (Training Dataset comprising of 1.2 million records) is shown below in tablular form. Please note the eleven Attributes/Features in the dataset. The last attribute is the Class Label, with categorical values 'good' and 'bad' for Benign and Malicious webpages respectively.

In [4]:

```
df_train
```

Out[4]:

	url	ip_add	geo_loc	url_len	js_len	js_obf_ler
0	http://members.tripod.com/russiastation/	42.77.221.155	Taiwan	40	58.0	0.(
1	http://www.ddj.com/cpp/184403822	3.211.202.180	United States	32	52.5	0.(
2	http://www.naef-usa.com/	24.232.54.41	Argentina	24	103.5	0.(
3	http://www.ff-b2b.de/	147.22.38.45	United States	21	720.0	532.8
4	http://us.imdb.com/title/tt0176269/	205.30.239.85	United States	35	46.5	0.(
...	...	...	...	...	...	..
1199995	http://csrc.nist.gov/rbac/	62.120.245.128	Saudi Arabia	26	106.0	0.(
1199996	http://www.unm.edu/~hist/	72.178.170.132	United States	25	36.0	0.(
1199997	http://www.syfyportal.com/news423380.html	181.240.45.113	Colombia	41	178.5	0.(
1199998	http://www.wardkenpo.ie	15.75.59.60	United States	23	121.0	0.(
1199999	http://homepages.gotadsl.co.uk/~jgm/ekmm/	168.239.57.229	United States	41	68.0	0.(

1200000 rows × 11 columns

As can be seen from the training dataset above, it has 1.2 million records. The test dataset has 0.364 million records (not shown here for the sake of simplicity). The dataset comprises of 10 features as mentioned earlier (apart from Class Label). While each of these attributes will be discussed and visualised in detail in this notebook, it is worth mentioning few aspects about the content attribute here itself. The content attribute contains the text retrived from the webpages (both textual data and JavaScript Code) after carrying out necessary cleaning. This 'content' attribute is importatnt as it can be used for extracting more features or carrying our detailed analysis of the webpages.

Analysis of Class Label & its Imbalance

The Class Label for this dataset is given in the last column. It has two values- 'good' and 'bad' corresponding to Benign and Malicious Webpages respectively. On the Internet, Malicious Webpages are few compared to Benign Webpages. This inequality shows in our dataset as well, since it has been scraped from Internet. The Class Label and its inequality is visualised and analysed below in detail.

In [5]:

```
# Class Distribution of Labels
df_train.groupby('label').size()
```

Out[5]:

```
label
bad        27253
good     1172747
dtype: int64
```

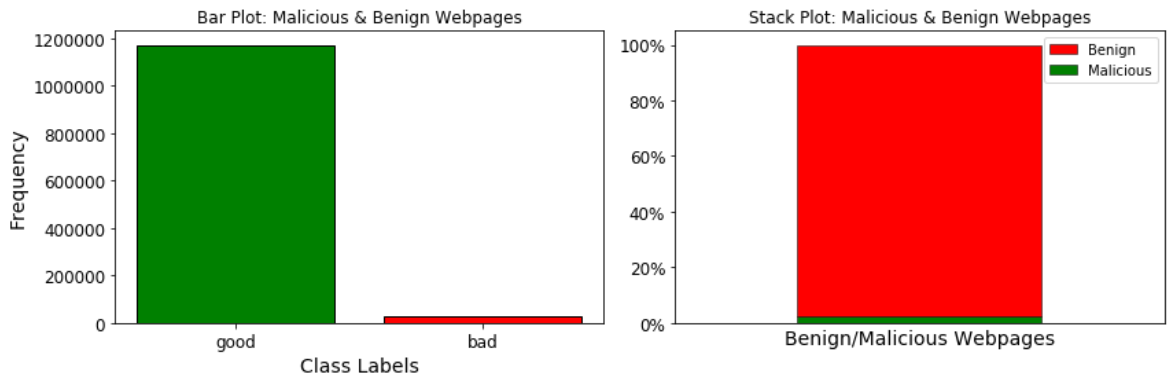
In [6]:

```
# Analysis of Postives and Negatives in the Dataset
pos,neg = df_train['label'].value_counts()
total = neg + pos
print ('Total of Samples: %s'% total)
print('Positive: {} ({:.2f}% of total)'.format(pos, 100 * pos / total))
print('Negative: {} ({:.2f}% of total)'.format(neg, 100 * neg / total))
```

```
Total of Samples: 1200000
Positive: 1172747 (97.73% of total)
Negative: 27253 (2.27% of total)
```

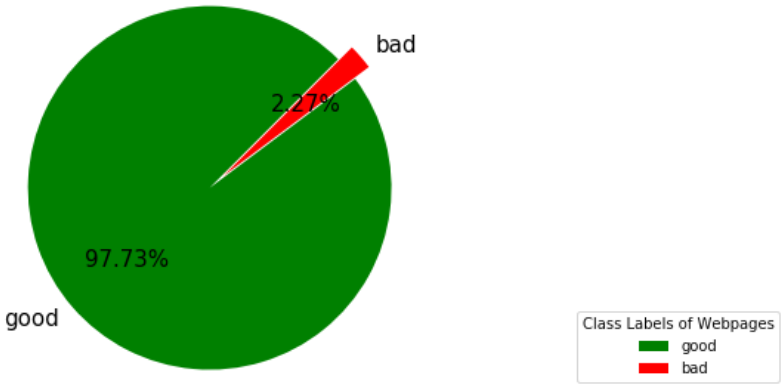
In [7]:

```
# Bar Plot of Malicious and Benign Websites
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick

fig = plt.figure(figsize = (12,4))
#title = fig.suptitle("Plot of Malicious and Benign Webpages", fontsize=14)
fig.subplots_adjust(top=0.85, wspace=0.3)
#Bar Plot
ax1 = fig.add_subplot(1,2,1)
ax1.set_xlabel("Class Labels")
ax1.set_ylabel("Frequency")
ax1.title.set_text('Bar Plot: Malicious & Benign Webpages')
labels = df_train['label'].value_counts()
w = (list(labels.index), list(labels.values))
ax1.tick_params(axis='both', which='major')
bar = ax1.bar(w[0], w[1], color=['green','red'], edgecolor='black', linewidth=1)
#Stacked Plot
ax2 = fig.add_subplot(1,2,2)
ax2.title.set_text('Stack Plot: Malicious & Benign Webpages')
# create dummy variable then group by that set the legend to false because we'll fix i
t later
df_train.assign(dummy = 1).groupby(['dummy','label']).size().groupby(level=0).apply(
    lambda x: 100 * x / x.sum()).to_frame().unstack().plot(kind='bar',stacked=True,leg
end=False,ax=ax2,color={'red','green'}, linewidth=0.50, ec='k')
ax2.set_xlabel('Benign/Malicious Webpages')# or it'll show up as 'dummy'
ax2.set_xticks([])# disable ticks in the x axis
current_handles, _ = plt.gca().get_legend_handles_labels()#Fixing Legend
reversed_handles = reversed(current_handles)
correct_labels = reversed(['Malicious','Benign'])
plt.legend(reversed_handles,correct_labels)
plt.gca().yaxis.set_major_formatter(mtick.PercentFormatter())
#Saving the Figs
figc = plt.gcf()
plt.tight_layout()
figc.savefig("imgs/Fig01&02: Bar Plot & Stack Plot of Malicious & Benign Webpages.svg"
)
extent = ax1.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig01: Bar Plot of Class Labels.svg",bbox_inches=extent.expanded(1.
5, 1.4))
extent = ax2.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig02: Stack Plot of Class Labels.svg",bbox_inches=extent.expanded(
1.5, 1.4))
```



In [8]:

```
# Pie Chart of Malicious and Benign Webpages Distribution
fig = plt.figure(figsize = (14,5))
Explode = [0,0.1]
plt.pie(w[1],explode=Explode,labels=w[0],shadow=False,startangle=45,
        colors=['green','red'],autopct='%.2f%%',textprops={'fontsize': 15})
plt.axis('equal')
plt.legend(title='Class Labels of Webpages',loc='lower right')
fig.savefig('imgs/Fig03:Pie Chart Distribution of Class Labels.svg')
plt.show()
```



As can be seen from the visualisations above, this dataset has significant class imbalance. Hence, during any machine learning process, adequate measures will have to be undertaken to handle or compensate this imbalance in order to get accurate results

Analysis of 'url' Attribute

The first column of the dataset has the 'url' attribute. Below, we carryout visualisation and analysis of webpage URLs by splitting it into its constituent words. These words are then used to generate vectorized value for each URL, giving a Profanity score based on good or bad words found in the URL. This score is then plotted.

In [9]:

```
#vectorising the URL Text
from urllib.parse import urlparse
from tld import get_tld

start_time= time.time()
#Function for cleaning the URL text before vectorization
def clean_url(url):
    url_text=""
    try:
        domain = get_tld(url, as_object=True)
        domain = get_tld(url, as_object=True)
        url_parsed = urlparse(url)
        url_text= url_parsed.netloc.replace(domain.tld," ").replace('www',' ') +" "+ u
rl_parsed.path+" "+url_parsed.params+" "+url_parsed.query+" "+url_parsed.fragment
        url_text = url_text.translate(str.maketrans({'?':' ','\\':' ','.':' ',';':' ',
'/':' ','\'':' '}))
        url_text.strip(' ')
        url_text.lower()
    except:
        url_text = url_text.translate(str.maketrans({'?':' ','\\':' ','.':' ',';':' ',
'/':' ','\'':' '}))
        url_text.strip(' ')
    return url_text

df_train['url_vect'] = df_train['url'].map(clean_url)
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))

# give profanity score to each URL using the Profanity_Check Library
from profanity_check import predict_prob

start_time= time.time()
#Function for calculating profanity in a dataset column
def predict_profanity(df):
    arr=predict_prob(df['url_vect'].astype(str).to_numpy())
    arr= arr.round(decimals=3)
    df['url_vect'] = pd.DataFrame(data=arr,columns=['url_vect'])
    #df['url']= df_test['url'].astype(float).round(decimals=3) #rounding probability t
o 3 decimal places
    return df['url_vect']

df_train['url_vect']= predict_profanity(df_train)

print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
```

```
***Total Time taken --- 55.30449104309082 seconds ---***
***Total Time taken --- 16.079758167266846 seconds ---***
```

In [10]:

```
import plotly.graph_objects as go

#df_trial
df_trial =df_train.iloc[0:1000,]
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']
fig = go.Figure()
t1= go.Histogram(x=df_trial_good['url_vect'],name='Benign Webpages',marker_color='gree
n')
t2= go.Histogram(x=df_trial_bad['url_vect'],name='Malicious Webpages',marker_color='re
d')
fig.add_trace(t1)
fig.add_trace(t2)
fig.update_layout(title="URL Analysis:Profanity Score of Vectorized URLs",xaxis_title=
"Profanity Score",yaxis_title="Count")
# Overlay both histograms
fig.update_layout(barmode='overlay')
# Reduce opacity to see both histograms
fig.update_traces(opacity=0.75)
fig.write_image("imgs/Fig04:URL Analysis-Profanity Score.svg")
fig.show()
```

While we see more red bars on the right, most green bars are on the left. Thus,the avergage profanity score of URLs for malicious webpages is more.

Analysis of 'ip_add' & 'geo_loc' Attributes

The 'ip_add' and 'geo_loc' are the 2nd and 3rd columns in the dataset. The 'ip_add' gives IP address of the web server where the webpage is hosted. The 'geo_loc' has been computed from the IP Address using the GeoIP Database and gives the country where the IP Address is located. The country wise distribution of IP Addresses of webpages in the dataset is plotted below on world map.

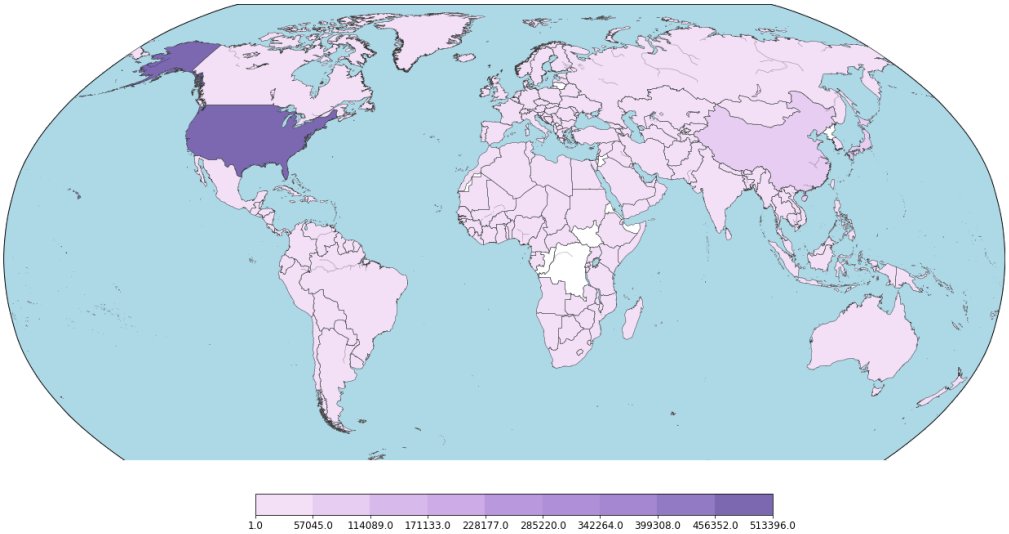
In [11]:

```
#Imports
from matplotlib.collections import PatchCollection
from mpl_toolkits.basemap import Basemap
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
from palettable.cartocolors.sequential import Purp_5
from palettable.colorbrewer.sequential import Reds_6

# Making of a DataFrame of Countrywise Count and categorized as Malcious and Benign
df_malicious = df_train.loc[df_train['label']=='bad']
df_benign = df_train.loc[df_train['label']=='good']
df_geo = pd.DataFrame(df_train['geo_loc'].value_counts())
df_geo_malicious = pd.DataFrame(df_malicious['geo_loc'].value_counts())
df_geo_benign = pd.DataFrame(df_benign['geo_loc'].value_counts())
df_geo.reset_index(inplace=True)
df_geo.rename(columns = {'index':'country','geo_loc':'count'}, inplace = True)
df_geo_malicious.reset_index(inplace=True)
df_geo_malicious.rename(columns = {'index':'country','geo_loc':'count'}, inplace = Tru
e)
df_geo_benign.reset_index(inplace=True)
df_geo_benign.rename(columns = {'index':'country','geo_loc':'count'}, inplace = True)
# Mapping ISO Codes
from geonamescache.mappers import country
mapper = country(from_key='name', to_key='iso3')
df_geo['country'] = df_geo['country'].apply(lambda x: mapper(x))
df_geo_malicious['country'] = df_geo_malicious['country'].apply(lambda x: mapper(x))
df_geo_benign['country'] = df_geo_benign['country'].apply(lambda x: mapper(x))
#Droping NAN values and Making ISO Codes as index
df_geo.dropna(inplace=True)
df_geo_malicious.dropna(inplace=True)
df_geo_benign.dropna(inplace=True)
df_geo.reset_index(inplace=True, drop=True)
df_geo_malicious.reset_index(inplace=True, drop=True)
df_geo_benign.reset_index(inplace=True, drop=True)
df_geo.set_index("country",inplace=True)
df_geo_malicious.set_index("country",inplace=True)
df_geo_benign.set_index("country",inplace=True)
```

In [12]:

```
# Plotting all IP Addresses on World Map
shapefile = 'shapefile/ne_10m_admin_0_countries_lakes'# Shape File in folder Shapefile
num_colors = 10
title = 'Geographical Distribution of IP Addresses Captured in Dataset'
description = "   Note: IP Addresses represent Addresses of the Webservers where these
Webpages were hosted. Total IP Addresses Captured : 1.2 million"
#Adding bin values to dataset df_geo for the Color
values = df_geo['count']
cm = Purp_5.mpl_colormap
#cm = plt.get_cmap('Blues') #Using Matploit's Color Map API
scheme = [cm(i / num_colors) for i in range(num_colors)]
bins = np.linspace(values.min(), values.max(), num_colors)
df_geo['bin'] = np.digitize(values, bins) -1
fig = plt.figure(figsize=(22, 12))
ax = fig.add_subplot(111, facecolor='w', frame_on=False)
fig.suptitle(title, fontsize=30, y=.95)
m = Basemap(lon_0=0, projection='robin')
m.drawmapboundary(color='w')
m.readshapefile(shapefile,'units', color='#444444', linewidth=.2)
m.drawmapboundary(color='w')
m.readshapefile(shapefile,'units', color='#444444', linewidth=.2)
m.drawcoastlines(linewidth=0.1)
m.drawmapboundary(fill_color='#add8e6')
m.drawcountries(linewidth=0.1)
for info, shape in zip(m.units_info, m.units):
    try:
        iso3 = info['ADM0_A3']
        if iso3 not in df_geo.index:
            color = '#FFFFFF'
        else:
            color = scheme[df_geo.loc[iso3]['bin']]
    except Exception as msg:
        print(iso3)
        print(msg)
    patches = [Polygon(np.array(shape), True)]
    pc = PatchCollection(patches)
    pc.set_facecolor(color)
    ax.add_collection(pc)
# Cover up Antarctica so legend can be placed over it.
ax.axhspan(0, 1000 * 1800, facecolor='w', edgecolor='w', zorder=2)
# Draw color legend.
ax_legend = fig.add_axes([0.32, 0.14, 0.4, 0.03], zorder=3)
cmap = mpl.colors.ListedColormap(scheme)
cb = mpl.colorbar.ColorbarBase(ax_legend, cmap=cmap, ticks=bins, boundaries=bins, orie
ntation='horizontal')
cb.ax.set_xticklabels([str(round(i)) for i in bins])
# Set the map footer.
plt.annotate(description, xy=(-.8, -3.2), size=14, xycoords='axes fraction')
fig.savefig('imgs/Fig05:Geographic Distribution of all IP Addresses.svg')
```

Geographical Distribution of IP Addresses Captured in Dataset

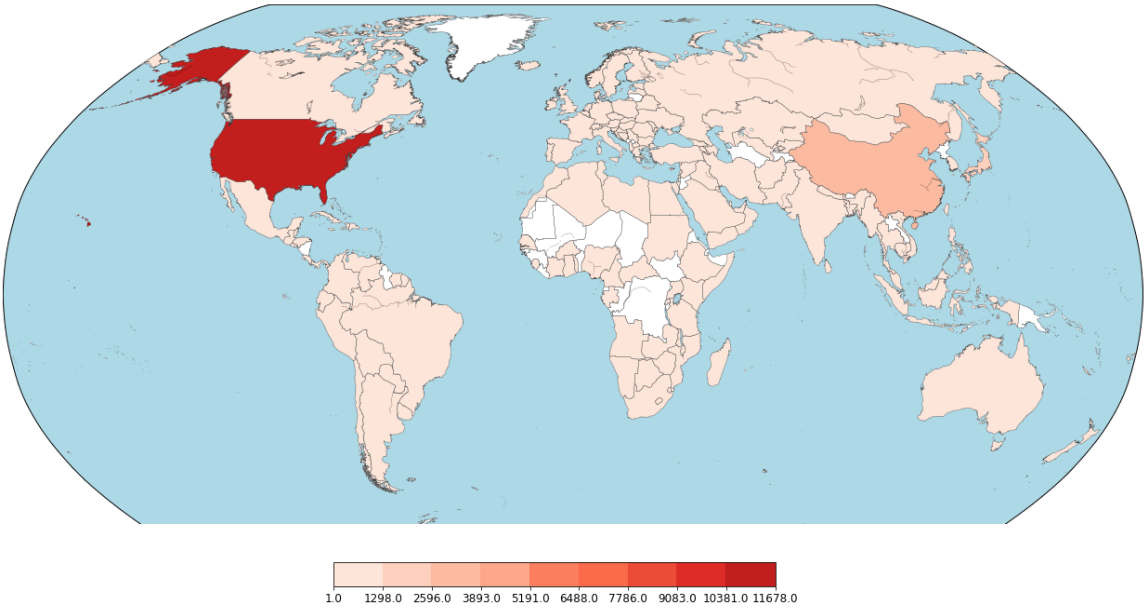


Note: IP Addresses represent Addresses of the Webservers where these Webpages were hosted. Total IP Addresses Captured : 1.2 million

In [13]:

```
#Plotting IP Addresses of Malicious Webpages
shapefile = 'shapefile/ne_10m_admin_0_countries_lakes'# Shape File in folder Shapefile
num_colors = 10
title = 'Geographical Distribution of IP Addresses: Malicious Webpages'
description = "Note: Location shown here depicts the Webserver where these Webpages we
re hosted. Total Malicious Webpages : 27253"
#Adding bin values to dataset df_geo_malicious for the Color
values = df_geo_malicious['count']
cm = Reds_6.mpl_colormap
#cm = plt.get_cmap('autumn_r') #Using Matploit's Color Map API
scheme = [cm(i / num_colors) for i in range(num_colors)]
bins = np.linspace(values.min(), values.max(), num_colors)
df_geo_malicious['bin'] = np.digitize(values, bins) -1
fig = plt.figure(figsize=(22, 12))
ax = fig.add_subplot(111, facecolor='w', frame_on=False)
fig.suptitle(title, fontsize=30, y=.95)
m = Basemap(lon_0=0, projection='robin')
m.drawmapboundary(color='w')
m.readshapefile(shapefile,'units', color='#444444', linewidth=.2)
m.drawcoastlines(linewidth=0.1)
m.drawmapboundary(fill_color='#add8e6')
m.drawcountries(linewidth=0.1)
for info, shape in zip(m.units_info, m.units):
    try:
        iso3 = info['ADM0_A3']
        if iso3 not in df_geo_malicious.index:
            color = '#ffffff'
        else:
            color = scheme[df_geo_malicious.loc[iso3]['bin']]
    except Exception as msg:
        print(iso3)
        print(msg)
    patches = [Polygon(np.array(shape), True)]
    pc = PatchCollection(patches)
    pc.set_facecolor(color)
    ax.add_collection(pc)
# Cover up Antarctica so legend can be placed over it.
ax.axhspan(0, 1000 * 1800, facecolor='w', edgecolor='w', zorder=2)
# Draw color legend.
ax_legend = fig.add_axes([0.35, 0.14, 0.3, 0.03], zorder=3)
cmap = mpl.colors.ListedColormap(scheme)
cb = mpl.colorbar.ColorbarBase(ax_legend, cmap=cmap, ticks=bins, boundaries=bins, orie
ntation='horizontal')
cb.ax.set_xticklabels([str(round(i)) for i in bins])
# Set the map footer.
plt.annotate(description, xy=(-.8, -3.2), size=14, xycoords='axes fraction')
fig.savefig('imgs/Fig06:Geographic Distribution of Malicious IP Addresses.svg')
```

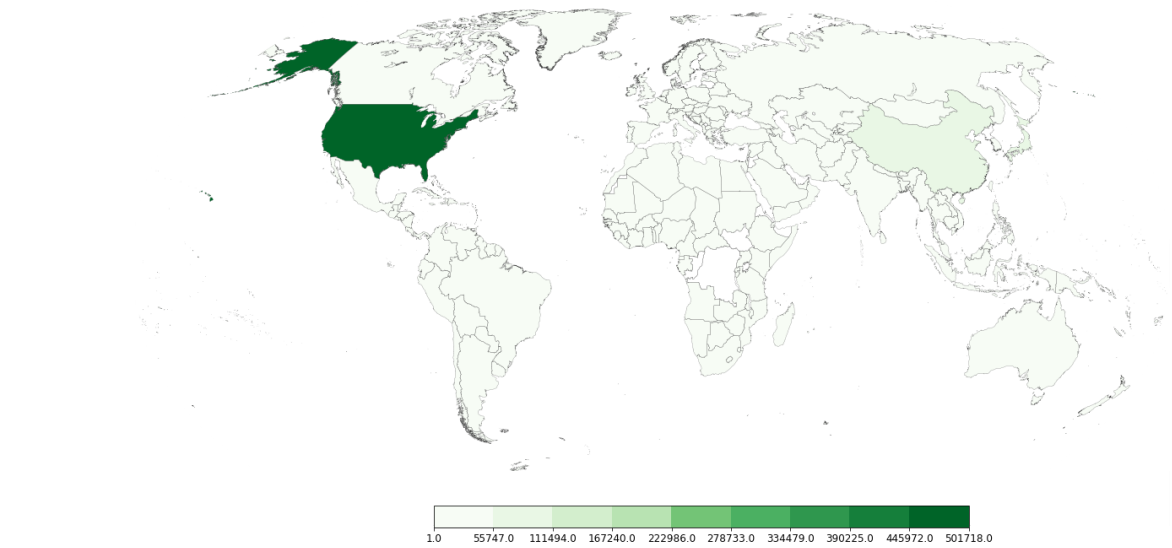
Geographical Distribution of IP Addresses: Malicious Webpages



Note: Location shown here depicts the Webserver where these Webpages were hosted. Total Malicious Webpages : 27253

In [14]:

```
#Plotting IP Addresses of Benign Webpages
shapefile = 'shapefile/ne_10m_admin_0_countries_lakes'# Shape File in folder Shapefile
num_colors = 10
title = 'Geographical Distribution of IP Addresses: Benign Webpages'
description = "Location shown here depicts the Webserver where these Webpages were hos
ted. Total Benign Webpages: 1.172 million"
#Adding bin values to dataset df_geo for the Color
values = df_geo_benign['count']
cm = plt.get_cmap('Greens') #Using Matploit's Color Map API
scheme = [cm(i / num_colors) for i in range(num_colors)]
bins = np.linspace(values.min(), values.max(), num_colors)
df_geo_benign['bin'] = np.digitize(values, bins) -1
fig = plt.figure(figsize=(22, 12))
ax = fig.add_subplot(111, facecolor='w', frame_on=False)
fig.suptitle(title, fontsize=30, y=.95)
m = Basemap(lon_0=0, projection='robin')
m.drawmapboundary(color='w')
m.readshapefile(shapefile,'units', color='#444444', linewidth=.2)
for info, shape in zip(m.units_info, m.units):
    try:
        iso3 = info['ADM0_A3']
        if iso3 not in df_geo_benign.index:
            color = '#FFFFFF'
        else:
            color = scheme[df_geo_benign.loc[iso3]['bin']]
    except Exception as msg:
        print(iso3)
        print(msg)
    patches = [Polygon(np.array(shape), True)]
    pc = PatchCollection(patches)
    pc.set_facecolor(color)
    ax.add_collection(pc)
# Cover up Antarctica so legend can be placed over it.
ax.axhspan(0, 1000 * 1800, facecolor='w', edgecolor='w', zorder=2)
# Draw color legend.
ax_legend = fig.add_axes([0.35, 0.14, 0.4, 0.03], zorder=3)
cmap = mpl.colors.ListedColormap(scheme)
cb = mpl.colorbar.ColorbarBase(ax_legend, cmap=cmap, ticks=bins, boundaries=bins, orie
ntation='horizontal')
cb.ax.set_xticklabels([str(round(i)) for i in bins])
# Set the map footer.
plt.annotate(description, xy=(-.8, -3.2), size=14, xycoords='axes fraction')
fig.savefig('imgs/Fig07:Geographic Distribution of Benign IP Addresses.svg')
```



As can be seen from the three maps above, the dataset covers complete globe. Majority of the IP addresses are active in USA and China, but that is because majority of web servers exist there. From these visualisations, no distinct pattern of malicious or benign webpages with respect to geographic location emerges.

Analysis of Numerical Attributes: 'url_len', 'js_len' and 'js_obf_len'

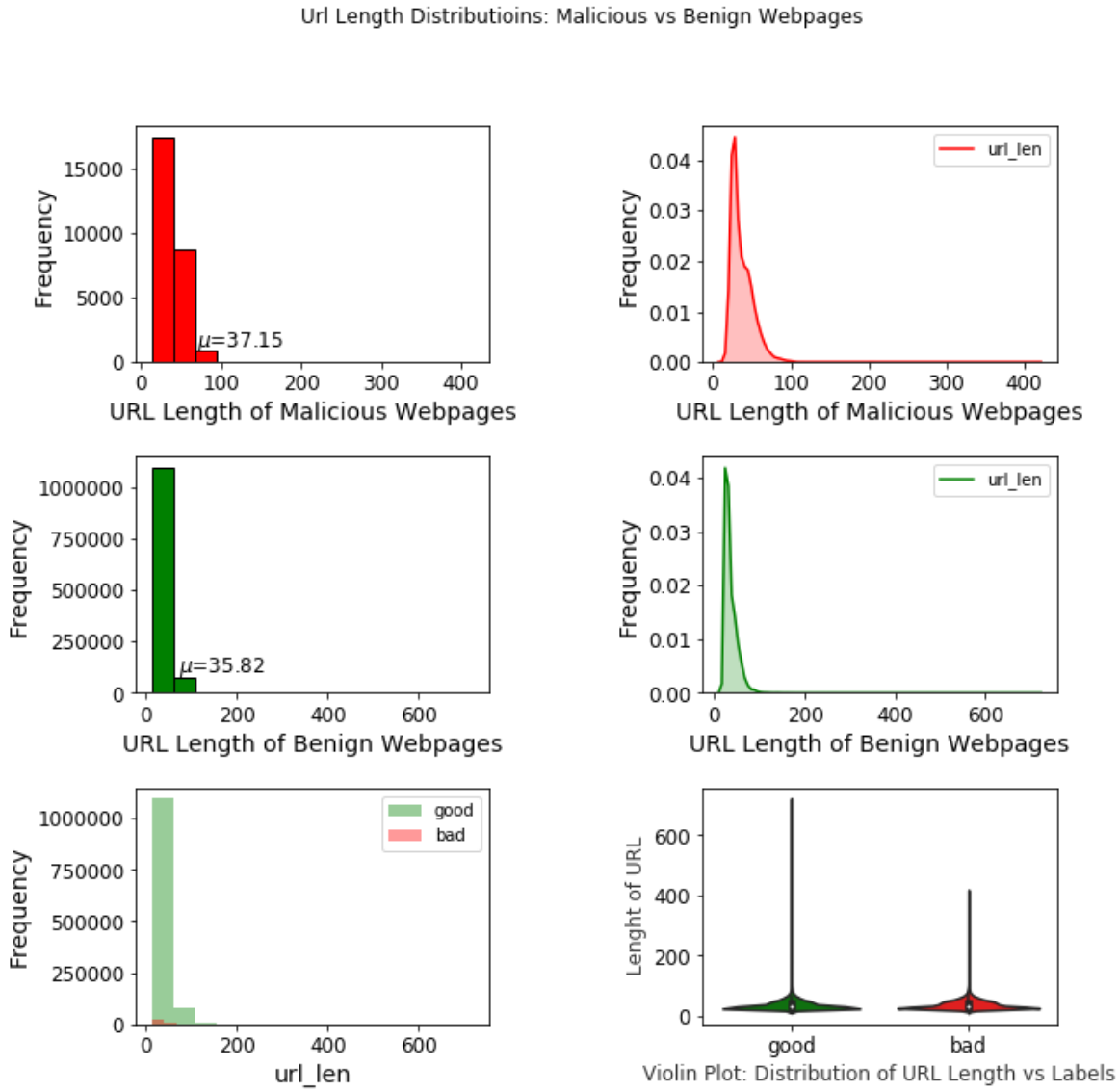
The 'url_len', 'js_len' and js_obf_len' are the 4th, 5th and 6th columns of the dataset respectively. All these three are numerical attributes. Hence, they have been discussed and visualised together in this section. First, a univariate visualisation of these individual attributes will be carried out. Thereater, a trivaritate visualisation of these attributes will be carried out to detect patterns/correlations amongst them. Then, based on correlation found, bivariate analysis of related attributes will be carried out. First, let us see with the univariate visualisation of these attributes.

Checking the 'url_len' Attribute using Univariate Plots:

In [15]:

```
# url_len analysis vis-a-vis malicious and benign webpages
df_train_bad=df_train.loc[df_train['label']=='bad']
df_train_good=df_train.loc[df_train['label']=='good']
# Histogram of Url Length: Malicious Webpages
fig = plt.figure(figsize =(10,10))
title = fig.suptitle("Url Length Distributioins: Malicious vs Benign Webpages")
fig.subplots_adjust(wspace=0.6,hspace=0.4)
ax = fig.add_subplot(3,2,1)
ax.set_xlabel("URL Length of Malicious Webpages")
ax.set_ylabel("Frequency")
ax.text(70, 1200, r'$\mu$='+str(round(df_train_bad['url_len'].mean(),2)), fontsize=12)
freq, bins, patches = ax.hist(df_train_bad['url_len'], color='red', bins=15, edgecolor
='black', linewidth=1)

# Density Plot of url_len: Malicious Webpages
ax1 = fig.add_subplot(3,2,2)
ax1.set_xlabel("URL Length of Malicious Webpages")
ax1.set_ylabel("Frequency")
sns.kdeplot(df_train_bad['url_len'], ax=ax1, shade=True, color='red')

# Histogram of url_len: Benign Webpages
ax2 = fig.add_subplot(3,2,3)
ax2.set_xlabel("URL Length of Benign Webpages")
ax2.set_ylabel("Frequency")
ax2.text(70, 100000, r'$\mu$='+str(round(df_train_good['url_len'].mean(),2)), fontsize
=12)
freq, bins, patches = ax2.hist(df_train_good['url_len'], color='green', bins=15, edgec
olor='black', linewidth=1)

# Density Plot of url_len: Benign Webpages
ax3 = fig.add_subplot(3,2,4)
ax3.set_xlabel("URL Length of Benign Webpages")
ax3.set_ylabel("Frequency")
sns.kdeplot(df_train_good['url_len'], ax=ax3, shade=True, color='green')

#Combined Plot of Malicious & Benign Webpages using Histogram
ax4 = fig.add_subplot(3,2,5)
ax4.set_ylabel("Frequency")
g = sns.FacetGrid(df_train, hue='label', palette={"good": "g", "bad": "r"})
g.map(sns.distplot, 'url_len', kde=False, bins=15, ax=ax4)
ax4.legend(prop={'size':10})
plt.tight_layout()

# Violin Plots of 'url_len'
ax5 = fig.add_subplot(3,2,6)
sns.violinplot(x="label", y="url_len", data=df_train, ax=ax5)
ax5.set_xlabel("Violin Plot: Distribution of URL Length vs Labels",size = 12,alpha=0.8
)
ax5.set_ylabel("Lenght of URL",size = 12,alpha=0.8)
#Saving the Figs
figc = fig
figc.savefig("imgs/Fig08-13: All Plots- URL Length Univariate Analysis.svg")
extent = ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig08: URL Length Histogram Malicious.svg",bbox_inches=extent.expan
ded(1.6, 1.5))
extent = ax1.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig09:URL Length Density Plot Malicious.svg",bbox_inches=extent.exp
anded(1.6, 1.5))
extent = ax2.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig10:URL Length Histogram Benign.svg",bbox_inches=extent.expanded(
1.6, 1.5))
extent = ax3.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig11:URL Length Density Plot Benign.svg",bbox_inches=extent.expand
```

```
ed(1.6, 1.5))
extent = ax4.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig12:URL Length Histogram-Benign & Malicious.svg",bbox_inches=exte
nt.expanded(1.6, 1.5))
extent = ax5.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig13:URL Length Violin Plot-Benign & Malicious.svg",bbox_inches=ex
tent.expanded(1.6, 1.5))
plt.close()
```



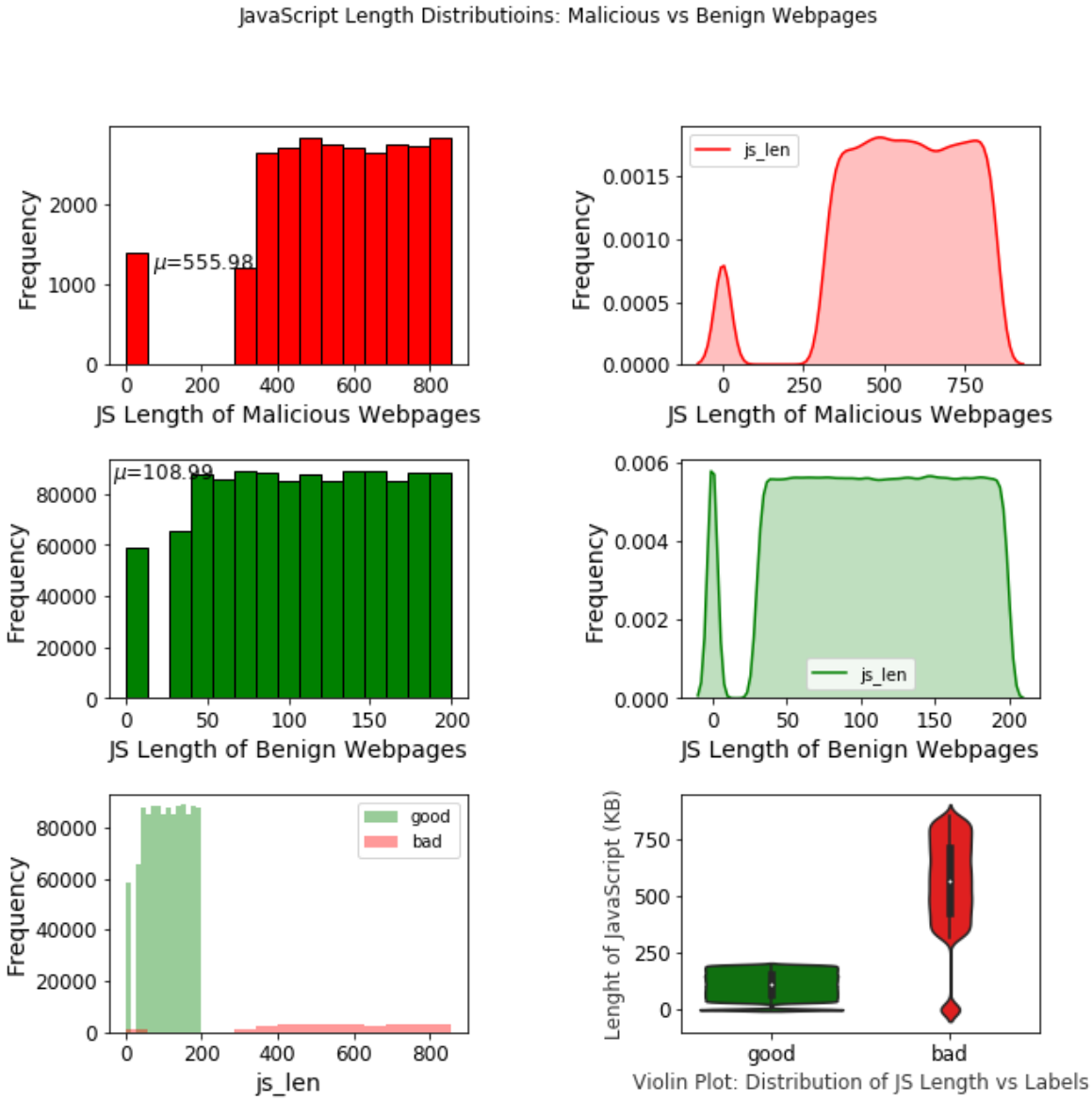
As can be seen from above plots of 'url_len, average URL length of malicious webpages is slightly more than benign webpages. However, no distinct pattern emerges.

Checking the 'js_len' Attribute using Univariate Plots:

In [16]:

```
# js_len analysis vis-a-vis malicious and benign webpages

# Histogram of JavaScript Length: Malicious Webpages
fig = plt.figure(figsize =(10,10))
title = fig.suptitle("JavaScript Length Distributioins: Malicious vs Benign Webpages")
fig.subplots_adjust(wspace=0.6,hspace=0.4)
ax = fig.add_subplot(3,2,1)
ax.set_xlabel("JS Length of Malicious Webpages")
ax.set_ylabel("Frequency")
ax.text(70, 1200, r'$\mu$='+str(round(df_train_bad['js_len'].mean(),2)), fontsize=12)
freq, bins, patches = ax.hist(df_train_bad['js_len'], color='red', bins=15, edgecolor=
'black', linewidth=1)

# Density Plot of js_len: Malicious Webpages
ax1 = fig.add_subplot(3,2,2)
ax1.set_xlabel("JS Length of Malicious Webpages")
ax1.set_ylabel("Frequency")
sns.kdeplot(df_train_bad['js_len'],ax=ax1,shade=True,color='red')

# Histogram of js_len: Benign Webpages
ax2 = fig.add_subplot(3,2,3)
ax2.set_xlabel("JS Length of Benign Webpages")
ax2.set_ylabel("Frequency")
ax2.text(-8, 86000, r'$\mu$='+str(round(df_train_good['js_len'].mean(),2)), fontsize=1
2)
freq, bins, patches = ax2.hist(df_train_good['js_len'], color='green', bins=15, edgeco
lor='black', linewidth=1)

# Density Plot of js_len: Benign Webpages
ax3 = fig.add_subplot(3,2,4)
ax3.set_xlabel("JS Length of Benign Webpages")
ax3.set_ylabel("Frequency")
sns.kdeplot(df_train_good['js_len'], ax=ax3, shade=True, color='green')

#Combined Plot of Malicious & Benign Webpages using Histogram
ax4 = fig.add_subplot(3,2,5)
ax4.set_ylabel("Frequency")
g = sns.FacetGrid(df_train, hue='label', palette={"good": "g", "bad": "r"})
g.map(sns.distplot, 'js_len', kde=False, bins=15, ax=ax4)
ax4.legend(prop={'size':10})
plt.tight_layout()

# Violin Plots of 'js_len'
ax5 = fig.add_subplot(3,2,6)
sns.violinplot(x="label", y="js_len", data=df_train, ax=ax5)
ax5.set_xlabel("Violin Plot: Distribution of JS Length vs Labels",size = 12,alpha=0.8)
ax5.set_ylabel("Lenght of JavaScript (KB)",size = 12,alpha=0.8)

#Saving the Figs
figc = fig
figc.savefig("imgs/Fig14-19: All Plots- JS Length Univariate Analysis.svg")
extent = ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig14: JS Length Histogram Malicious.svg",bbox_inches=extent.expand
ed(1.6, 1.5))
extent = ax1.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig15:JS Length Density Plot Malicious.svg",bbox_inches=extent.expa
nded(1.7, 1.5))
extent = ax2.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig16:JS Length Histogram Benign.svg",bbox_inches=extent.expanded(
1.6, 1.5))
extent = ax3.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig17:JS Length Density Plot Benign.svg",bbox_inches=extent.expande
d(1.6, 1.5))
```

```
extent = ax4.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig18:JS Length Histogram-Benign & Malicious.svg",bbox_inches=exten
t.expanded(1.6, 1.5))
extent = ax5.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig19:JS Length Violin Plot-Benign & Malicious.svg",bbox_inches=ext
ent.expanded(1.6, 1.5))
plt.close()
```



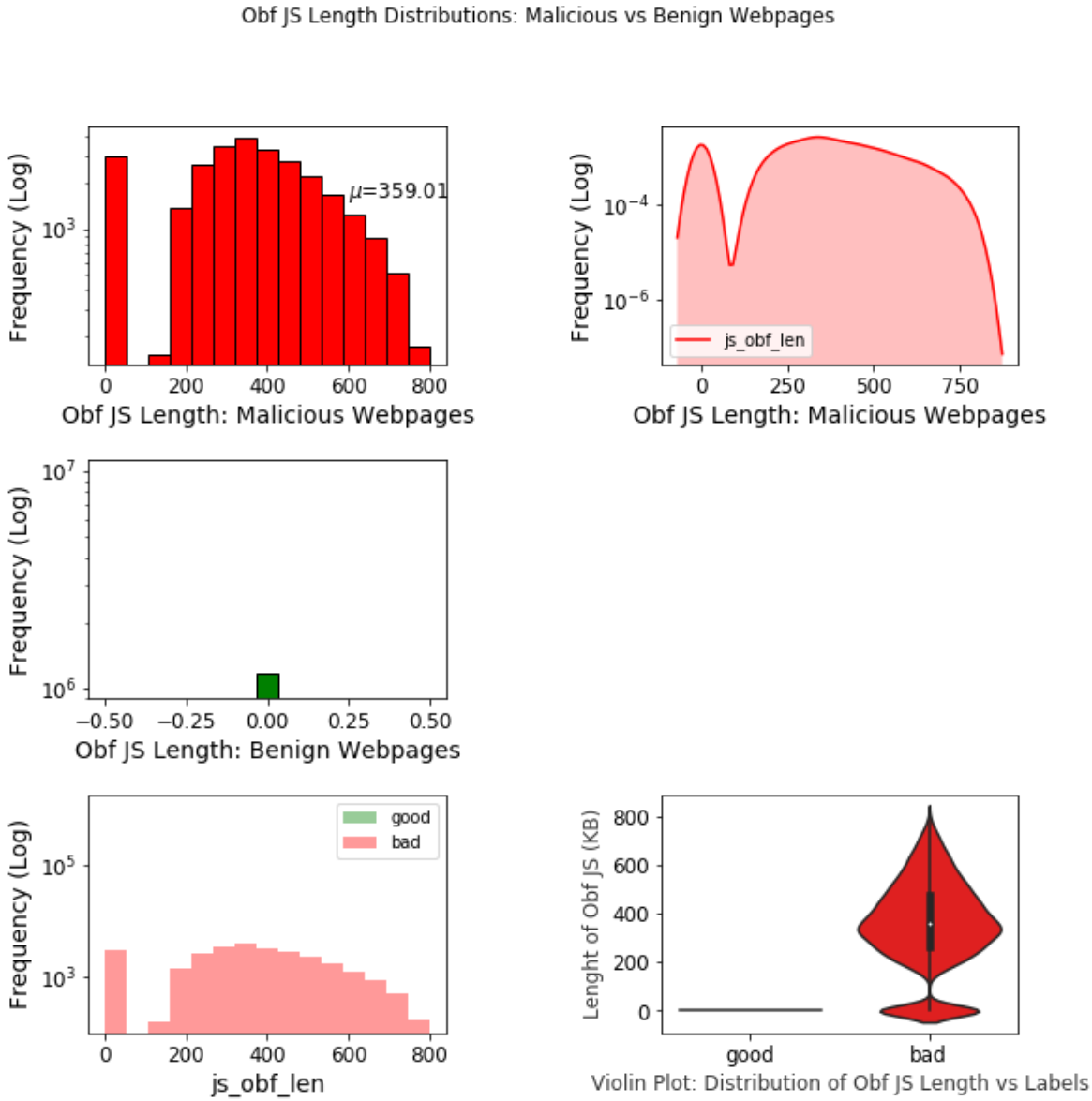
As seen from plots above, average JavaScript length of Malicious Webpages is 555.98 KB, while that of Benign Webpages is less at 108.99 KB. Thus, a clear distinct pattern can be visualised between the 'js_len' of two classes.

Checking the 'js_obf_len' Attribute using Univariate Plots:

In [17]:

```
# js_obf_len analysis vis-a-vis malicious and benign webpages

# Histogram of Obfuscated JavaScript Length: Malicious Webpages
fig = plt.figure(figsize =(10,10))
title = fig.suptitle("Obf JS Length Distributions: Malicious vs Benign Webpages")
fig.subplots_adjust(wspace=0.6,hspace=0.4)
ax = fig.add_subplot(3,2,1)
ax.set_xlabel("Obf JS Length: Malicious Webpages")
ax.set_ylabel("Frequency (Log)")
plt.yscale('log', nonposy='clip')
ax.text(600, 1600, r'$\mu$='+str(round(df_train_bad['js_obf_len'].mean(),2)), fontsize
=12)
freq, bins, patches = ax.hist(df_train_bad['js_obf_len'], color='red', bins=15, edgeco
lor='black', linewidth=1)

# Density Plot of js_obf_len: Malicious Webpages
ax1 = fig.add_subplot(3,2,2)
ax1.set_xlabel("Obf JS Length: Malicious Webpages")
ax1.set_ylabel("Frequency (Log)")
plt.yscale('log', nonposy='clip')
sns.kdeplot(df_train_bad['js_obf_len'],ax=ax1,shade=True,color='red')

# Histogram of js_obf_len: Benign Webpages
ax2 = fig.add_subplot(3,2,3)
ax2.set_xlabel("Obf JS Length: Benign Webpages")
ax2.set_ylabel("Frequency (Log)")
plt.yscale('log', nonposy='clip')
ax2.hist(df_train_good['js_obf_len'], color='green', bins=15, edgecolor='black', linew
idth=1)

#Combined Plot of Malicious & Benign Webpages using Histogram
ax3 = fig.add_subplot(3,2,5)
ax3.set_ylabel("Frequency (Log)")
plt.yscale('log', nonposy='clip')
g = sns.FacetGrid(df_train, hue='label', palette={"good": "g", "bad": "r"})
g.map(sns.distplot, 'js_obf_len', kde=False, bins=15, ax=ax3)
ax3.legend(prop={'size':10})
plt.tight_layout()

# Violin Plots of 'js_obf_len'
ax4 = fig.add_subplot(3,2,6)
sns.violinplot(x="label", y="js_obf_len", data=df_train, ax=ax4)
ax4.set_xlabel("Violin Plot: Distribution of Obf JS Length vs Labels",size = 12,alpha=
0.8)
ax4.set_ylabel("Lenght of Obf JS (KB)",size = 12,alpha=0.8)

#Saving the Figs
figc = fig
figc.savefig("imgs/Fig20-24: All Plots- Obf_JS Length Univariate Analysis.svg")
extent = ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig20: Obf_JS Length Histogram Malicious.svg",bbox_inches=extent.ex
panded(1.6, 1.5))
extent = ax1.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig21:Obf_JS Length Density Plot Malicious.svg",bbox_inches=extent.
expanded(1.7, 1.5))
extent = ax2.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig22:Obf_JS Length Histogram Benign.svg",bbox_inches=extent.expand
ed(1.6, 1.5))
extent = ax3.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig23:Obf_JS Length Density Plot Benign.svg",bbox_inches=extent.exp
anded(1.6, 1.5))
extent = ax4.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
figc.savefig("imgs/Fig24:Obf_JS Length Violin Plot-Benign & Malicious.svg",bbox_inches
```

```
=extent.expanded(1.6, 1.5))
plt.close()
```



As seen from plots above, very few (almost negligible) Benign Webpages have obfuscated JavaScript code. On the other hand, Malicious Webpages have an average Obfuscated JavaScript length of 359.01 KB. Thus, a clear pattern emerges here.

Trivariate Analysis of all three Numerical Attributes: 'url_len', 'js_len' & 'js_obf_len'

The statistical values of these three numerical columns is given below in two tables.

In [18]:

```
#Statistical Values of all three numerical Columns
df_train.describe()
```

Out[18]:

	url_len	js_len	js_obf_len	url_vect
count	1.200000e+06	1.200000e+06	1.200000e+06	1.200000e+06
mean	3.585337e+01	1.191463e+02	8.153424e+00	1.118906e-01
std	1.441089e+01	9.046649e+01	6.001398e+01	3.581809e-02
min	1.200000e+01	0.000000e+00	0.000000e+00	3.000000e-03
25%	2.600000e+01	6.650000e+01	0.000000e+00	1.030000e-01
50%	3.200000e+01	1.120000e+02	0.000000e+00	1.210000e-01
75%	4.200000e+01	1.580000e+02	0.000000e+00	1.210000e-01
max	7.210000e+02	8.541000e+02	8.028540e+02	1.000000e+00

In [19]:

```
#Statistical Values of all three numerical Columns: Segregated Based on Class Labels
df_train_good= df_train.loc[df_train['label']=='good']
df_train_bad= df_train.loc[df_train['label']=='bad']
subset_attributes = ['url_len', 'js_len', 'js_obf_len']
g = round(df_train_good[subset_attributes].describe(),2)
b = round(df_train_bad[subset_attributes].describe(),2)
pd.concat([g,b], axis=1, keys=['Benign Webpages Statistics', 'Malicious Webpages Statistics'])
```

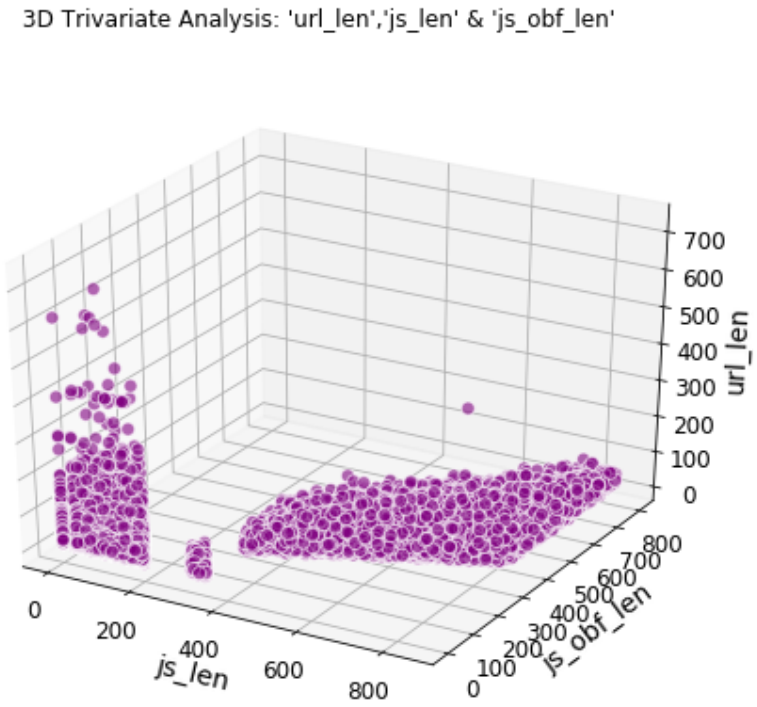
Out[19]:

	Benign Webpages Statistics			Malicious Webpages Statistics		
	url_len	**js_len**	**js_obf_len**	**url_len**	**js_len**	**js_obf_len**
count	1172747.00	1172747.00	1172747.0	27253.00	27253.00	27253.00
mean	35.82	108.99	0.0	37.15	555.98	359.01
std	14.42	53.97	0.0	14.02	199.36	180.63
min	12.00	0.00	0.0	13.00	0.00	0.00
25%	26.00	65.50	0.0	27.00	431.10	261.86
50%	32.00	110.00	0.0	33.00	569.70	361.35
75%	42.00	155.00	0.0	45.00	714.60	478.86
max	721.00	199.50	0.0	416.00	854.10	802.85

Please see the distinction that emerges, in the table above, for the values of 'js_len' and 'js_obf_len' for the two class labels.

In [20]:

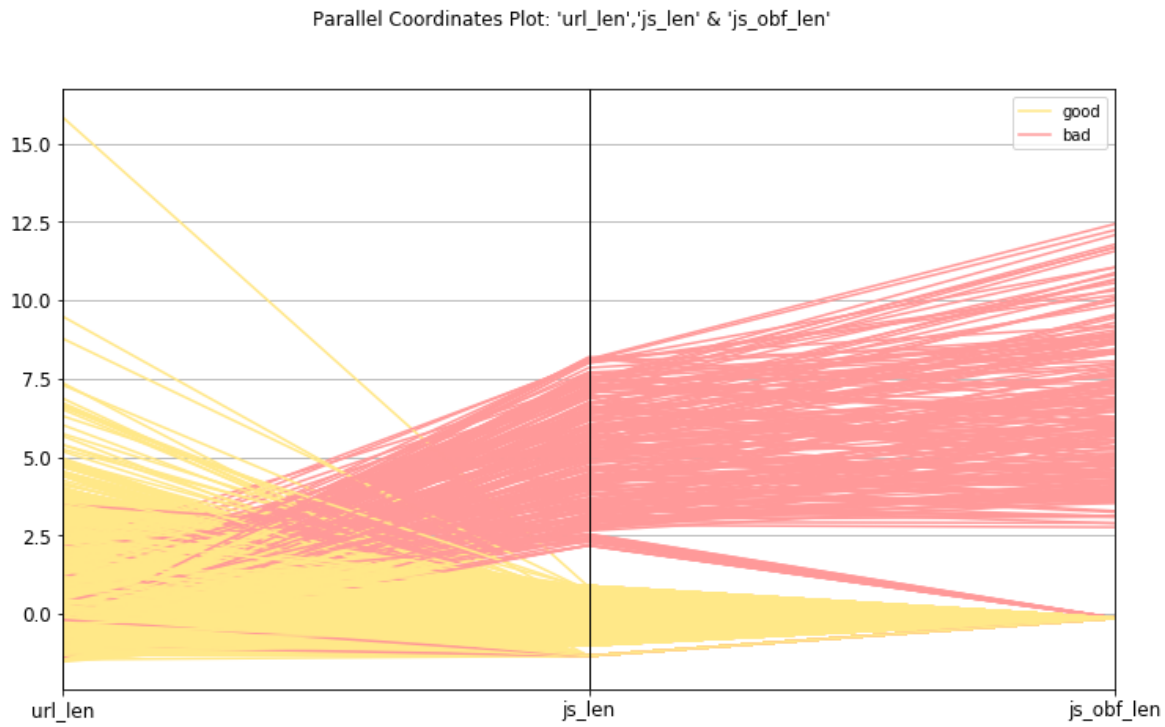
```
# Visualizing 3-D numeric data with Scatter Plots
# length, breadth and depth
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
title = fig.suptitle("3D Trivariate Analysis: 'url_len','js_len' & 'js_obf_len'")
xs = df_train.iloc[:,]['js_len']
ys = df_train.iloc[:,]['js_obf_len']
zs = df_train.iloc[:,]['url_len']
ax.scatter(xs, ys, zs, s=50, alpha=0.6, edgecolors='w',color='purple')
ax.set_xlabel('js_len')
ax.set_ylabel('js_obf_len')
ax.set_zlabel('url_len')
fig.savefig("imgs/Fig25: 3D Scatter Trivariate Analysis.png")
```



In [21]:

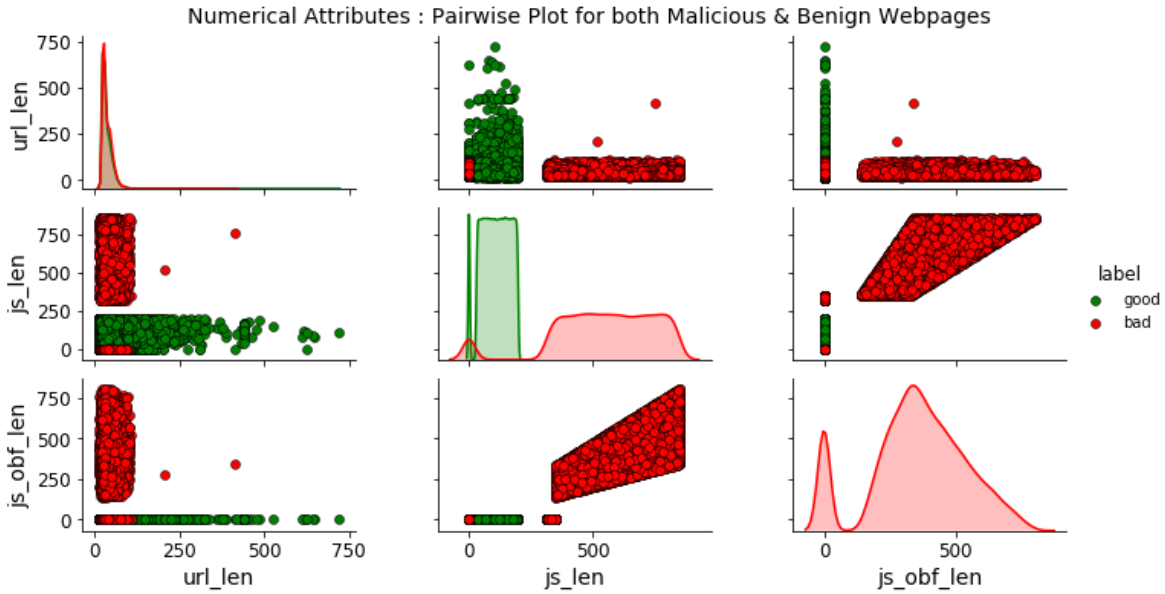
```
#Parallel Coordinates Plot:url_len, js_len & js_obf_len vs Malicious & Benign Webpages
from sklearn.preprocessing import StandardScaler
from pandas.plotting import parallel_coordinates

start_time= time.time()
# Scaling attribute values to avoid few outiers
cols = ['url_len','js_len','js_obf_len']
subset_df = df_train.iloc[:10000,][cols]
ss = StandardScaler()
scaled_df = ss.fit_transform(subset_df)
scaled_df = pd.DataFrame(scaled_df, columns=cols)
final_df = pd.concat([scaled_df, df_train.iloc[:10000,]['label']], axis=1)
final_df
# plot parallel coordinates
fig=plt.figure(figsize = (12,7))
title = fig.suptitle("Parallel Coordinates Plot: 'url_len','js_len' & 'js_obf_len'")
pc = parallel_coordinates(final_df, 'label', color=('#FFE888', '#FF9999'))
fig.savefig("imgs/Fig26: Parallel Coordinates Plot-Trivariate Analysis.png")
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
```

```
***Total Time taken --- 90.4581778049469 seconds ---***
```



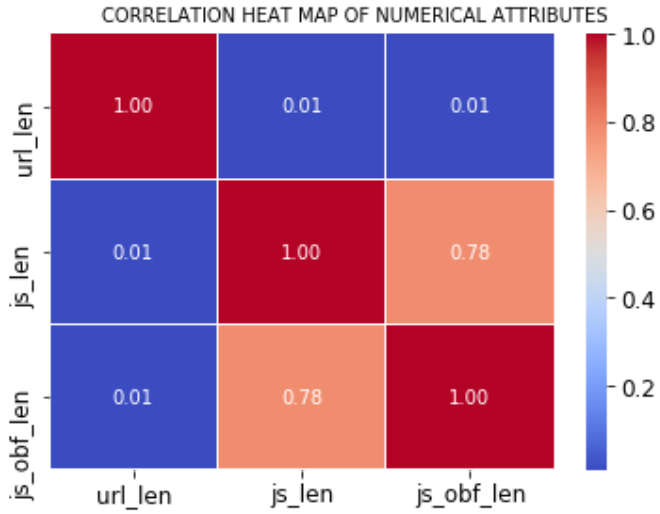
In [22]:

```
# Scatter Plot with Hue for visualising data in 3-D
cols = ['url_len', 'js_len', 'js_obf_len','label']
pp = sns.pairplot(df_train[cols], hue='label', size=1.8, aspect=1.8,
                  palette={"good": "green", "bad": "red"},
                  plot_kws=dict(edgecolor="black", linewidth=0.5))
fig = pp.fig
fig.subplots_adjust(top=0.93, wspace=0.3)
t = fig.suptitle('Numerical Attributes : Pairwise Plot for both Malicious & Benign Web pages', fontsize=14)
fig.savefig("imgs/Fig27: Scatter Plot-Trivariate Analysis.png")
```



In [23]:

```
# Correlation Matrix Heatmap of Numerical Attributes
f, ax = plt.subplots(figsize=(6, 4))
corr = df_train[['url_len','js_len','js_obf_len']].corr()
hm = sns.heatmap(round(corr,2), annot=True, ax=ax, cmap="coolwarm",fmt='.2f',linewidth
s=.05)
f.subplots_adjust(top=0.93)
t= f.suptitle('CORRELATION HEAT MAP OF NUMERICAL ATTRIBUTES', fontsize=10)
extent =ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
f.savefig("imgs/Fig28: Correlation Matrix-Trivariate Analysis.png",bbox_inches=extent.
expanded(1.6, 1.5))
```

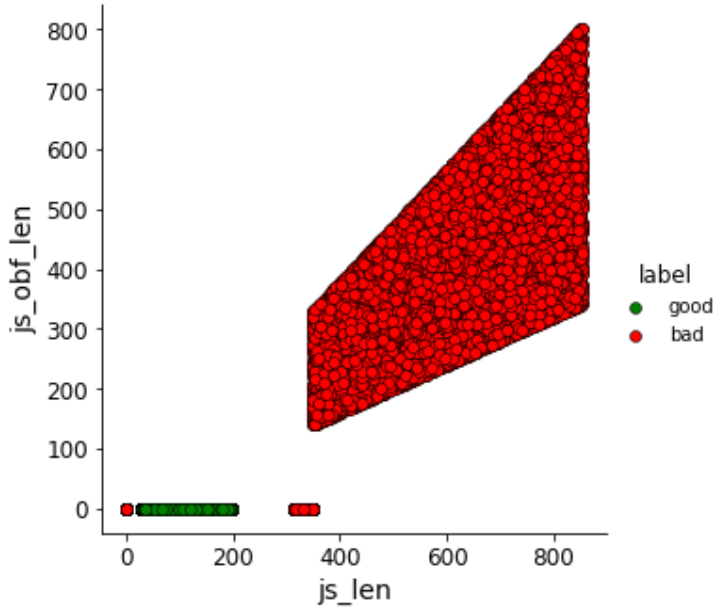


From the trivariate analysis above, it clearly emerges that 'js_len' and 'js_obf_len' are highly correlated and have a distinct pattern for the two class labels. Hence, these two variables will be analysed further through bivariate analysis to gain further insight.

Bivariate Analysis of : 'js_len' & 'js_obf_len'

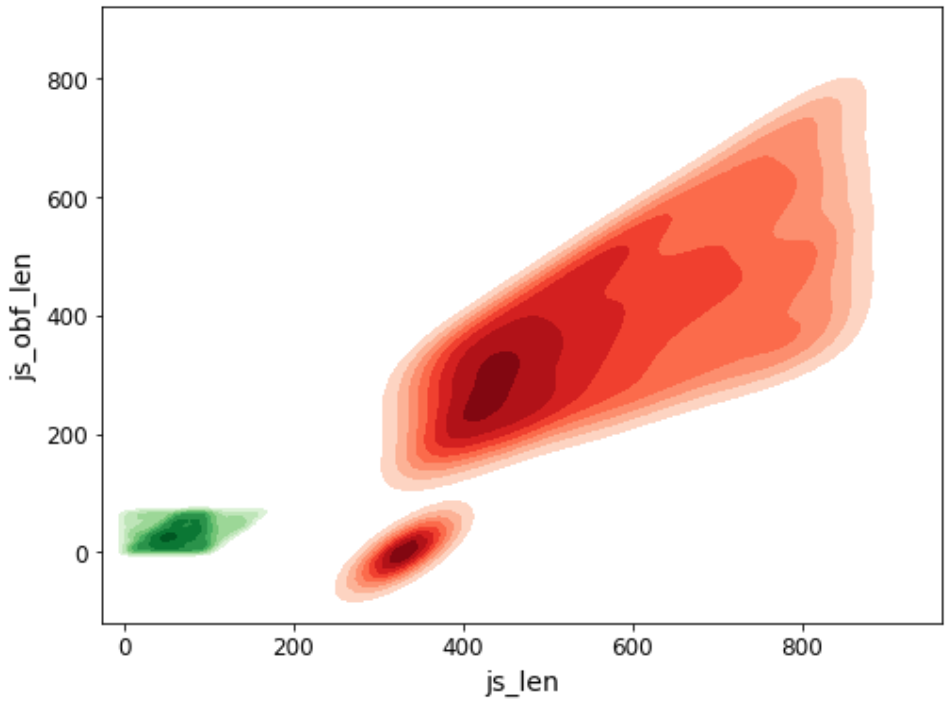
In [24]:

```
# Scatter Plot of 'js_len' and 'js_obf_len'
pp=sns.pairplot(df_train,x_vars=["js_len"],y_vars=["js_obf_len"],size=4.5,hue="label",
 palette={"good": "green", "bad": "red"},plot_kws=dict(edgecolor="k",linewidth=0.5))
fig = pp.fig
fig.savefig("imgs/Fig29: Pair Plot-Bivariate Analysis.png")
```



In [25]:

```
#Bivariate Density Plot: 'js_len' & 'js_obf_len'
fig = plt.figure(figsize=(8, 6))
df_trial_good= df_train_good.iloc[:5000,]
df_trial_good['js_obf_len']= df_trial_good['js_obf_len'].apply(lambda x: randrange(70
))
df_trial_good['js_len']= df_trial_good['js_obf_len'].apply(lambda x: x*randrange(2)+ra
ndrange(100))
df_trial_good.dropna(inplace=True)
df_trial_bad= df_train_bad.iloc[:5000,]
df_trial_bad= df_trial_bad.loc[df_trial_bad['js_len']>50]
ax = sns.kdeplot(df_trial_bad['js_len'], df_trial_bad['js_obf_len'],hue='label',
                 cmap='Reds',shade=True, shade_lowest=False)
ax = sns.kdeplot(df_trial_good['js_len'], df_trial_good['js_obf_len'],hue='label',
                 cmap='Greens',shade=True, shade_lowest=False)
extent =ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
fig.savefig("imgs/Fig30: Density Plot-Bivariate Analysis.png",bbox_inches=extent.expan
ded(1.6, 1.5))
```



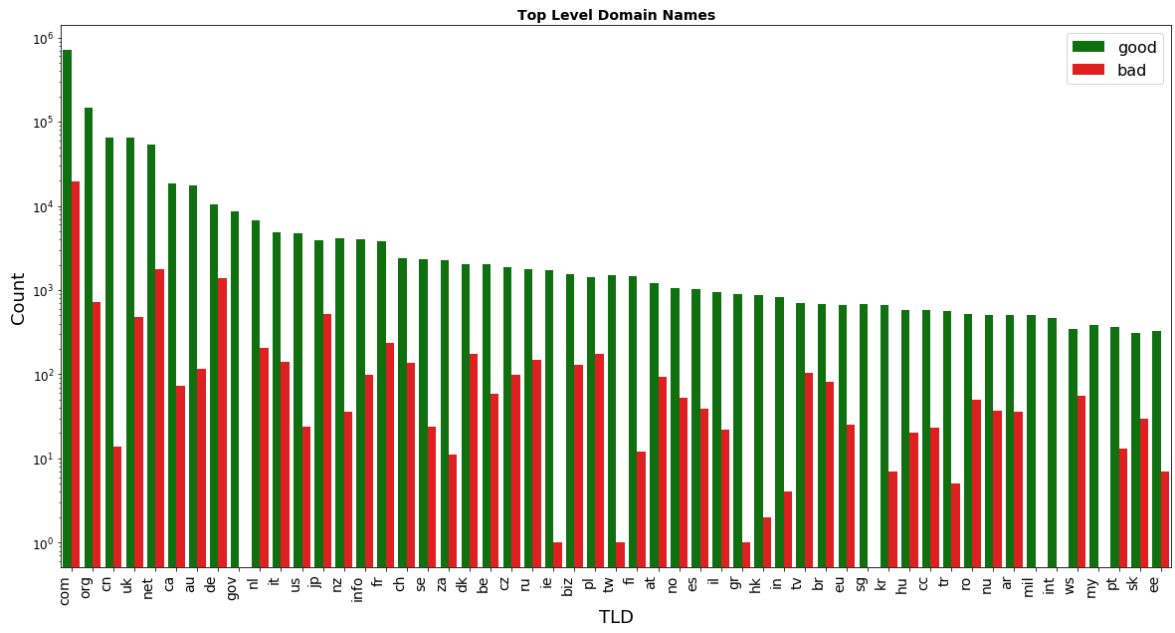
The above two bivariate graphs clearly show that 'js_len' and 'js_obf_len' can segregate the two classes with low overlap.

Analysis of Top Level Domain: 'tld' Attribute

The 'tld' attribute is the 7th column in the dataset. It is a categorical attribute that gives the Top Level Domain name of the webpage.

In [26]:

```
# 'tld' Histogram
import re

def tld(s):
    p= re.split('\.',s)
    return p[-1]
df_trial = df_train.iloc[:,]
df_trial['tld']= df_trial['tld'].apply(tld)
df_trial['tld'].replace({'edu':'cn'},inplace=True)
df_trial= df_trial.groupby('tld').filter(lambda x : len(x)>300)
fig=plt.figure(figsize=(20,10))
ax = sns.countplot(x='tld',data=df_trial,hue='label',
                   order=df_trial['tld'].value_counts().index)
ax.set_xticklabels(ax.get_xticklabels(),rotation=90, ha="right",fontsize=14)
plt.title('Top Level Domain Names', fontsize=14, fontweight='bold')
ax.legend(loc='upper right',fontsize=16)
plt.xlabel('TLD',fontsize=18)
plt.ylabel('Count',fontsize=18)
ax.set_yscale("log")
fig.savefig("imgs/Fig31:TLD Histogram.png")
plt.show()
```



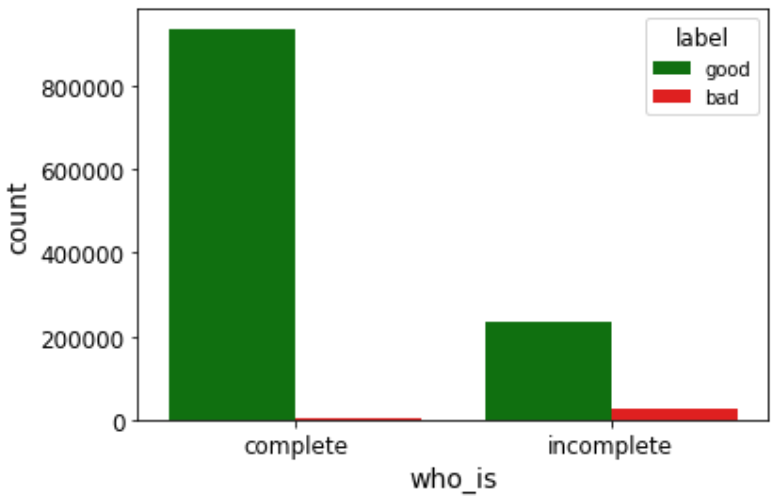
Seeing the graph above, no clear pattern emerges with respect to 'tld' when plotted for both the classes.

Analysis of WHO IS Registration Information: 'who_is' Attribute

The 'who_is' attribute is the 8th column of the dataset. It is a categorical attribute with two values - 'complete' and 'incomplete', reflecting whether the registration details are complete or not.

In [27]:

```
# Multi-bar Plot of 'who_is' attribute: Malicious vs Benign Webpages
fig= plt.figure(figsize = (6,4))
cp = sns.countplot(x="who_is", hue="label", data=df_train,
                   palette={"good": "green", "bad": "red"})
fig.savefig("imgs/Fig32: WHO_IS Plot.png")
```



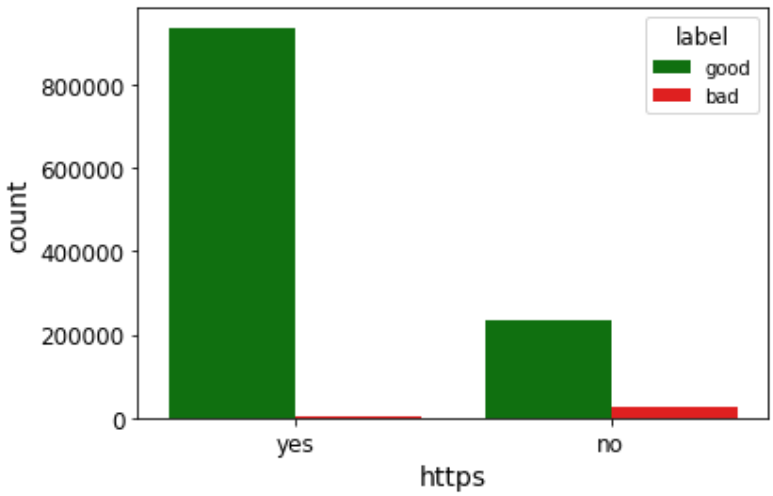
As seen above, Malicious webpages are more likely to have incomeplete registration details vis-a-vis Benign webpages.

Analysis of HTTP Status: 'https' Attribute

The 'https' attribute is the 9th attribute in the dataset. It is a categorical attribute with two values- 'yes' and 'no', indicating whether the webpage is delivered using the secure HTTPS protocol or otherwise.

In [28]:

```
# Multi-bar Plot of 'https' attribute: Malicious vs Benign Webpages
fig= plt.figure(figsize = (6,4))
cp = sns.countplot(x="https", hue="label", data=df_train,
                   palette={"good": "green", "bad": "red"})
fig.savefig("imgs/Fig33: HTTPS Plot.png")
```



As seen above, more number of Benign webpages use HTTPS protocol vis-a-vis Malicious webpages.

Visualization of Web Content (Raw Web content Including JavaScript)

The 10th column of the dataset has the 'content' attribute. This attribute has the raw web content of the webpage, including JavaScript code. However, this raw web content was cleaned and processed to remove punctuations, stop words, etc., in order to reduce data size. The web content has been stored as a separate attribute in the dataset, so that more attributes could be extracted for future requirements. Also, this raw content may be used in machine learning techniques that can use unstructured data, for example, Deep Learning.

In this section we carryout visualisation of this raw web content data using various techniques.

Sentiment Polarity Analysis of Web Content

In [29]:

```
from textblob import TextBlob
import plotly.graph_objects as go

# Adding Sentiment Polarity Column to a new Dataset
start_time = time.time()
df_trial =df_train.iloc[0:10000,]
df_trial['polarity'] = df_trial['content'].map(lambda content: TextBlob(content).senti
ment.polarity)
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
#df_trial
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']
fig = go.Figure()
t1= go.Histogram(x=df_trial_good['polarity'],name='Benign Webpages',marker_color='gree
n')
t2= go.Histogram(x=df_trial_bad['polarity'],name='Malicious Webpages',marker_color='re
d')
fig.add_trace(t1)
fig.add_trace(t2)
fig.update_layout(title="Sentiment Analysis of Web Content",xaxis_title="Sentiment Pol
arity Score",yaxis_title="Count")
# Overlay both histograms
fig.update_layout(barmode='overlay')
# Reduce opacity to see both histograms
fig.update_traces(opacity=0.75)
fig.write_image("imgs/Fig33:Sentiment Analysis-Web Content.svg")
fig.show()
```

```
***Total Time taken --- 26.348896026611328 seconds ---***
```

The sentiment analysis of the web content is displayed above. This analyis gives a score based on sentiments deduced from sentences on the webpage. As seen, Benign web pages have a higher positive sentiment score vis-a-vis Malicious Webpages.

Profanity Analysis of Web Content

In [30]:

```
# give profanity score to Web Content using the Profanity_Check Library
from profanity_check import predict_prob
import plotly.graph_objects as go

df_trial = df_train.iloc[:100000,]
start_time= time.time()
#Function for calculating profanity in a dataset column
def predict_profanity(df):
    arr=predict_prob(df['content'].astype(str).to_numpy())
    arr= arr.round(decimals=3)
    df['content_profanity'] = pd.DataFrame(data=arr,columns=['content_profanity'])
    #df['url']= df_test['url'].astype(float).round(decimals=3) #rounding probability t
o 3 decimal places
    return df['content_profanity']

df_trial['content_profanity']= predict_profanity(df_trial)
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))

#df_trial : good and bad
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']
#Plotting it on Histograms
fig = go.Figure()
t1= go.Histogram(x=df_trial_good['content_profanity'],name='Benign Webpages',marker_co
lor='green')
t2= go.Histogram(x=df_trial_bad['content_profanity'],name='Malicious Webpages',marker_
color='red')
fig.add_trace(t1)
fig.add_trace(t2)
fig.update_layout(title="Profanity Analysis of Web Content",xaxis_title="Profanity Sco
re",yaxis_title="Count")
# Overlay both histograms
fig.update_layout(barmode='overlay')
# Reduce opacity to see both histograms
fig.update_traces(opacity=0.75)
fig.write_image("imgs/Fig34:Profanity Analysis-Web Content.svg")
fig.show()
```

```
***Total Time taken --- 24.195387840270996 seconds ---***
```

The profanity analysis of the web content is displayed above. This analyis gives a score based on bad/obscene words found on the webpage. As seen, Malicious webpages have a higher Profanity score vis-a-vis Benign webpages

Length of Web Content Analysis

In [31]:

```
df_trial['content_len'] = df_trial['content'].astype(str).apply(len)
#df_trial : good and bad
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']
#Plotting it on Histograms
fig = go.Figure()
t1= go.Histogram(x=df_trial_good['content_len'],name='Benign Webpages',marker_color='g
reen')
t2= go.Histogram(x=df_trial_bad['content_len'],name='Malicious Webpages',marker_color=
'red')
fig.add_trace(t1)
fig.add_trace(t2)
fig.update_layout(title="Length of Web Content",xaxis_title="Length",yaxis_title="Coun
t")
# Overlay both histograms
fig.update_layout(barmode='overlay')
# Reduce opacity to see both histograms
fig.update_traces(opacity=0.75)
fig.write_image("imgs/Fig35:Content Length Analysis-Web Content.svg")
fig.show()
```

The length of web content is displayed above. As seen, Benign web pages have lesser web content lengths vis-a-vis Malicious Webpages

Word Count Analysis

In [32]:

```
df_trial['content_word_count'] = df_trial['content'].apply(lambda x: len(str(x).split
()))
#df_trial : good and bad
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']
#Plotting it on Histograms
fig = go.Figure()
t1= go.Histogram(x=df_trial_good['content_word_count'],name='Benign Webpages',marker_c
olor='green')
t2= go.Histogram(x=df_trial_bad['content_word_count'],name='Malicious Webpages',marker
_color='red')
fig.add_trace(t1)
fig.add_trace(t2)
fig.update_layout(title="Word Count Analysis",xaxis_title="Words",yaxis_title="Count")
# Overlay both histograms
fig.update_layout(barmode='overlay')
# Reduce opacity to see both histograms
fig.update_traces(opacity=0.75)
fig.write_image("imgs/Fig36:Word Count Analysis-Web Content.svg")
fig.show()
```

The word count analysis of web content is displayed above. As seen, Malicious webpages have higher word counts compared to Benign webpages.

Vector Plotting of Web Content

For the purpose of further mathematical and visual analysis, the content is converted into a 20 code vector using TensorFlow Text Encoder. These 20 code vectors are then stored in a new dataset as 20 different columns. This new dataset is then used for visualisation.

In [33]:

```
# Using Transfer Learning from Tensorflow hub- Universal Text Encoder
import tensorflow_hub as hub

start_time= time.time()
# Text Encoder with Output fixed 512 vector
#encoder = hub.load('https://tfhub.dev/google/universal-sentence-encoder/4')
# Word Embedder with fixed 20 vector output
encoder = hub.load("https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1")
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
#encoder(['Hello World']) #For Testing the Encoder
```

```
***Total Time taken --- 0.502626895904541 seconds ---***
```

In [34]:

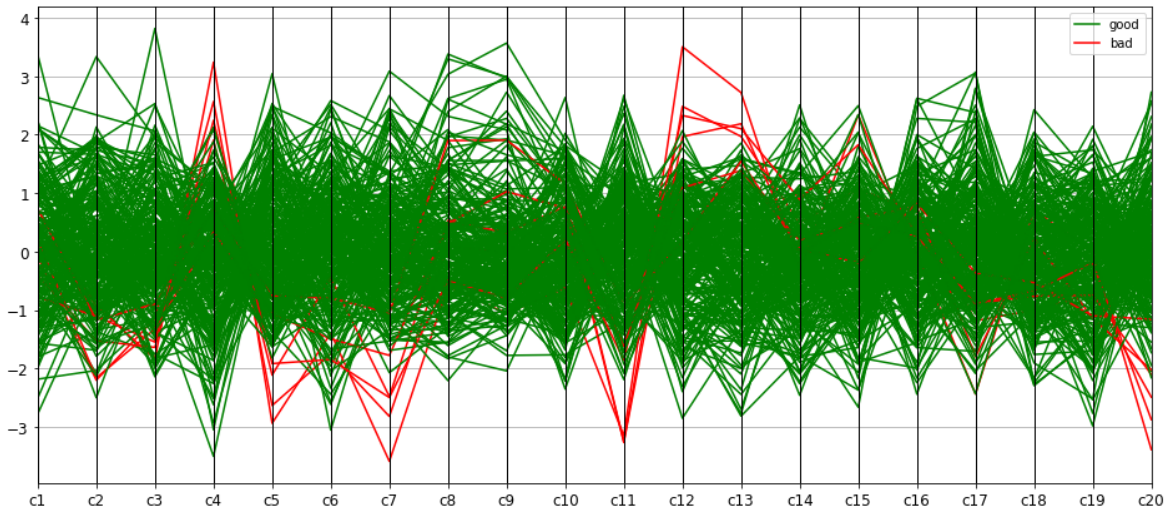
```
#Encoding Values in the Dataset
start_time= time.time()
#df_trial
df_trial = df_train.iloc[:100000,]
#df_trial : good and bad
df_trial_good = df_trial.loc[df_train['label']=='good']
df_trial_bad = df_trial.loc[df_train['label']=='bad']

def create_encoded_array(df):
    arr=np.empty((len(df.index),20))
    for x in df.index:
        arr[x,:]=encoder([df.iloc[x]['content']])
    return arr
arr= create_encoded_array(df_trial)
df_content_encoded = pd.DataFrame(data=arr,columns=['c1','c2','c3','c4','c5','c6','c7'
,
    'c8','c9','c10','c11','c12','c13','c14','c15','c16','c17','c18','c19','c20'])
df_content_encoded['label']=df_trial['label']
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
```

```
***Total Time taken --- 146.7027039527893 seconds ---***
```

In [35]:

```
#Parallel Coordinates Plot:Vector Outputs vs Malicious & Benign Webpages
from sklearn.preprocessing import StandardScaler
from pandas.plotting import parallel_coordinates

start_time= time.time()
# Scaling attribute values to avoid few outiers
cols = ['c1','c2','c3','c4','c5','c6','c7','c8','c9','c10','c11','c12','c13','c14','c1
5','c16','c17','c18','c19','c20']
subset_df = df_content_encoded[cols]
ss = StandardScaler()
scaled_df = ss.fit_transform(subset_df)
scaled_df = pd.DataFrame(scaled_df, columns=cols)
final_df = pd.concat([scaled_df, df_content_encoded['label']], axis=1)
final_df
# plot parallel coordinates
fig= plt.figure(figsize = (16,7))
pc = parallel_coordinates(final_df.iloc[:250,], 'label', color=('green', 'red'))
print("***Total Time taken --- %s seconds ---***" % (time.time() - start_time))
fig.savefig("imgs/Fig37:Parallel Coordinates-Web Content Vectors.png")
```

```
***Total Time taken --- 0.754817008972168 seconds ---***
```



As seen from the Parallel Coordinates plot for all 20 code vectors representing the web content, few code points show distinction between Malicious and Benign webpages. Thus, these code points together may help in segregating the classes.

Analysis of Complete Dataset: Reducing all Attributes to 3 Dimensions Using PCA

For the purpose of 3D visualisation of the complete dataset, multiple attributes of the dataset are reduced to three principal components using the Principal Component Analysis (PCA).

In [36]:

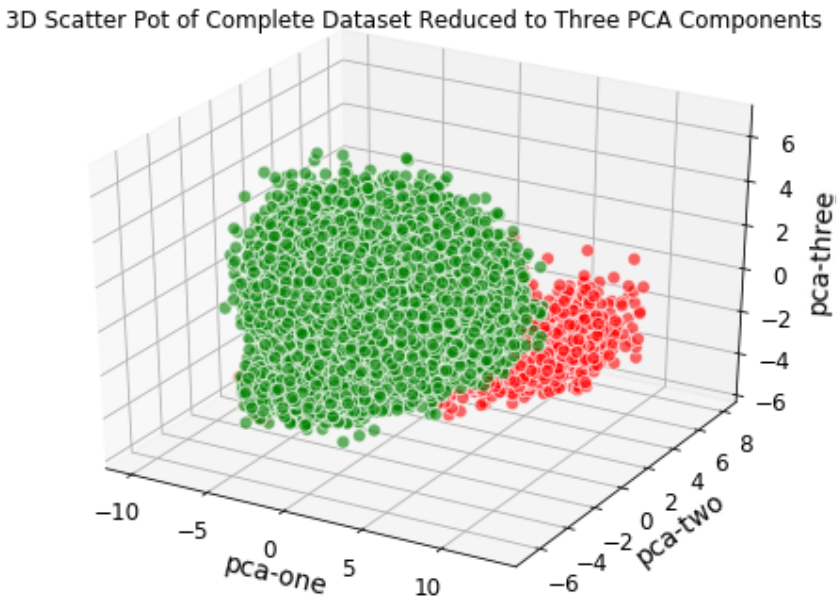
```
#Surface Plot after reducing dimensions using PCA
from sklearn.decomposition import PCA

pca = PCA(n_components=3)
pca_result = pca.fit_transform(final_df[cols].values)
final_df['pca-one'] = pca_result[:,0]
final_df['pca-two'] = pca_result[:,1]
final_df['pca-three'] = pca_result[:,2]
print('Explained variation per principal component: {}'.format(pca.explained_variance_
ratio_))
```

```
Explained variation per principal component: [0.37438488 0.14118358 0.1066
0458]
```

The 3D scatter plot of the principal components deduced above is given below.

In [37]:

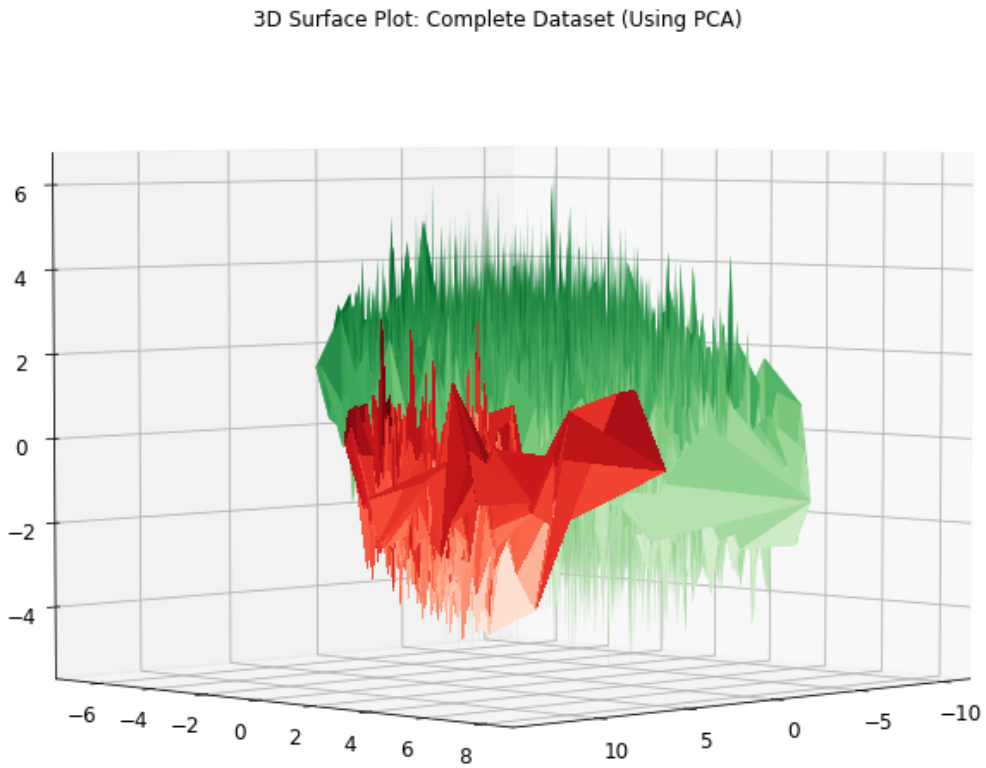
```
# Visualizing 3-D numeric data with Scatter Plots
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
xs = final_df.loc[final_df['label']=='good']['pca-one']
ys = final_df.loc[final_df['label']=='good']['pca-two']
zs = final_df.loc[final_df['label']=='good']['pca-three']
ax.scatter(xs,ys,zs,s=50, alpha=0.6, edgecolors='w',color='green')
xs = final_df.loc[final_df['label']=='bad']['pca-one']
ys = final_df.loc[final_df['label']=='bad']['pca-two']
zs = final_df.loc[final_df['label']=='bad']['pca-three']
ax.scatter(xs, ys, zs, s=50, alpha=0.6, edgecolors='w',color='red')
ax.set_xlabel('pca-one')
ax.set_ylabel('pca-two')
ax.set_zlabel('pca-three')
ax.set_title("3D Scatter Pot of Complete Dataset Reduced to Three PCA Components")
extent =ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
fig.savefig("imgs/Fig38: 3D Scatter-PCA Analysis.png",bbox_inches=extent.expanded(1.6,
1.5))
```



The Surface Plot of the dataset is given below.

In [38]:

```
from mpl_toolkits import mplot3d
import random

fig = plt.figure(figsize=(12,10))
x_good = final_df.loc[final_df['label']=='good']['pca-one']
y_good = final_df.loc[final_df['label']=='good']['pca-two']
z_good = final_df.loc[final_df['label']=='good']['pca-three']
x_bad  = final_df.loc[final_df['label']=='bad']['pca-one']
y_bad  = final_df.loc[final_df['label']=='bad']['pca-two']
z_bad  = final_df.loc[final_df['label']=='bad']['pca-three']
ax.set_xlabel('pca-one')
ax.set_ylabel('pca-two')
ax.set_zlabel('pca-three')
ax = plt.axes(projection='3d')
surf = ax.plot_trisurf(x_bad,y_bad,z_bad, linewidth=0, antialiased=False,cmap='Reds',
edgecolor='none')
surf = ax.plot_trisurf(x_good,y_good,z_good, linewidth=0, antialiased=True,cmap='Green
s', edgecolor='none')
ax.set_title('3D Surface Plot: Complete Dataset (Using PCA)')
ax.view_init(4, 45)
extent =ax.get_window_extent().transformed(figc.dpi_scale_trans.inverted())
fig.savefig("imgs/Fig39: TriSurf Plot-PCA Analysis.png",bbox_inches=extent.expanded(1.
6, 1.5))
plt.show()
```



As seen from the 3D scatter plot and 3D Surface Plot, the dataset can be segregated into its two classes- Malicious(bad) and Benign(good).

Miscellaneous Maintenance Code: Run this for Selected Variables to Clear RAM Space

(Note: Run this selectively only if you are Running Short of Memory)

In [39]:

```
#Clearing Additional load of variables: Creating More RAM Space
import gc

#del df_train_good
#del df_train_bad
#del df_trial
#gc.collect()
```