cse519_hw2_Li_Kai_113358694

September 23, 2021

0.1 Setup

- Code to download the data directly from the colab notebook.
- If you find it easier to download the data from the kaggle website (and uploading it to your drive), you can skip this section.

[ ]:
```
# First mount your drive before running these cells.
# Create a folder for the this HW and change to that dir
%cd
```

[ ]:
```
!pip install -q kaggle
```

[ ]:
```
from google.colab import files
# Create a new API token under "Account" in the kaggle webpage and download the␣
 ↪json file
# Upload the file by clicking on the browse
files.upload()
```

[ ]:
```
!kaggle competitions download -c microsoft-malware-prediction
```

0.2 Section 1: Library and Data Imports (Q1)

- Import your libraries and read the data into a dataframe. Print the head of the dataframe.

[1]:
```
use_cols = ["MachineIdentifier", "SmartScreen", "AVProductsInstalled",␣
 ↪"AppVersion", "CountryIdentifier", "Census_OSInstallTypeName",␣
 ↪"Wdft_IsGamer",
            "EngineVersion", "AVProductStatesIdentifier", "Census_OSVersion",␣
 ↪"Census_TotalPhysicalRAM", "Census_ActivationChannel",
            "RtpStateBitfield", "Census_ProcessorModelIdentifier",␣
 ↪"Census_PrimaryDiskTotalCapacity",
             "Census_InternalPrimaryDiagonalDisplaySizeInInches",␣
 ↪"Wdft_RegionIdentifier", "LocaleEnglishNameIdentifier",
            "AvSigVersion", "IeVerIdentifier", "IsProtected",␣
 ↪"Census_InternalPrimaryDisplayResolutionVertical",␣
 ↪"Census_PrimaryDiskTypeName",
             "Census_OSWUAutoUpdateOptionsName", "Census_OSEdition",␣
 ↪"Census_GenuineStateName", "Census_ProcessorCoreCount",
```

```
            "Census_OEMNameIdentifier", "Census_MDC2FormFactor",␣
 ↪"Census_FirmwareManufacturerIdentifier", "OsBuildLab",␣
 ↪"Census_OSBuildRevision",
             "Census_OSBuildNumber", "Census_IsPenCapable",␣
 ↪"Census_IsTouchEnabled", "Census_IsAlwaysOnAlwaysConnectedCapable",␣
 ↪"Census_IsSecureBootEnabled",
             "Census_SystemVolumeTotalCapacity", "HasDetections"
            ]
dtypes = {
        'MachineIdentifier':                                    'category',
        'ProductName':                                          'category',
        'EngineVersion':                                        'category',
        'AppVersion':                                           'category',
        'AvSigVersion':                                         'category',
        'IsBeta':                                               'int8',
        'RtpStateBitfield':                                     'float16',
        'IsSxsPassiveMode':                                     'int8',
        'DefaultBrowsersIdentifier':                            'float16',
        'AVProductStatesIdentifier':                            'float32',
        'AVProductsInstalled':                                  'float16',
        'AVProductsEnabled':                                    'float16',
        'HasTpm':                                               'int8',
        'CountryIdentifier':                                    'int16',
        'CityIdentifier':                                       'float32',
        'OrganizationIdentifier':                               'float16',
        'GeoNameIdentifier':                                    'float16',
        'LocaleEnglishNameIdentifier':                          'int8',
        'Platform':                                             'category',
        'Processor':                                            'category',
        'OsVer':                                                'category',
        'OsBuild':                                              'int16',
        'OsSuite':                                              'int16',
        'OsPlatformSubRelease':                                 'category',
        'OsBuildLab':                                           'category',
        'SkuEdition':                                           'category',
        'IsProtected':                                          'float16',
        'AutoSampleOptIn':                                      'int8',
        'PuaMode':                                              'category',
        'SMode':                                                'float16',
        'IeVerIdentifier':                                      'float16',
        'SmartScreen':                                          'category',
        'Firewall':                                             'float16',
        'UacLuaenable':                                         'float32',
        'Census_MDC2FormFactor':                                'category',
        'Census_DeviceFamily':                                  'category',
        'Census_OEMNameIdentifier':                             'float16',
        'Census_OEMModelIdentifier':                            'float32',
```

```
        'Census_ProcessorCoreCount':                                'float16',
        'Census_ProcessorManufacturerIdentifier':                   'float16',
        'Census_ProcessorModelIdentifier':                          'float16',
        'Census_ProcessorClass':                                    'category',
        'Census_PrimaryDiskTotalCapacity':                          'float32',
        'Census_PrimaryDiskTypeName':                               'category',
        'Census_SystemVolumeTotalCapacity':                         'float32',
        'Census_HasOpticalDiskDrive':                               'int8',
        'Census_TotalPhysicalRAM':                                  'float32',
        'Census_ChassisTypeName':                                   'category',
        'Census_InternalPrimaryDiagonalDisplaySizeInInches':        'float16',
        'Census_InternalPrimaryDisplayResolutionHorizontal':        'float16',
        'Census_InternalPrimaryDisplayResolutionVertical':          'float16',
        'Census_PowerPlatformRoleName':                             'category',
        'Census_InternalBatteryType':                               'category',
        'Census_InternalBatteryNumberOfCharges':                    'float32',
        'Census_OSVersion':                                         'category',
        'Census_OSArchitecture':                                    'category',
        'Census_OSBranch':                                          'category',
        'Census_OSBuildNumber':                                     'int16',
        'Census_OSBuildRevision':                                   'int32',
        'Census_OSEdition':                                         'category',
        'Census_OSSkuName':                                         'category',
        'Census_OSInstallTypeName':                                 'category',
        'Census_OSInstallLanguageIdentifier':                       'float16',
        'Census_OSUILocaleIdentifier':                              'int16',
        'Census_OSWUAutoUpdateOptionsName':                         'category',
        'Census_IsPortableOperatingSystem':                         'int8',
        'Census_GenuineStateName':                                  'category',
        'Census_ActivationChannel':                                 'category',
        'Census_IsFlightingInternal':                               'float16',
        'Census_IsFlightsDisabled':                                 'float16',
        'Census_FlightRing':                                        'category',
        'Census_ThresholdOptIn':                                    'float16',
        'Census_FirmwareManufacturerIdentifier':                    'float16',
        'Census_FirmwareVersionIdentifier':                         'float32',
        'Census_IsSecureBootEnabled':                               'int8',
        'Census_IsWIMBootEnabled':                                  'float16',
        'Census_IsVirtualDevice':                                   'float16',
        'Census_IsTouchEnabled':                                    'int8',
        'Census_IsPenCapable':                                      'int8',
        'Census_IsAlwaysOnAlwaysConnectedCapable':                  'float16',
        'Wdft_IsGamer':                                             'float16',
        'Wdft_RegionIdentifier':                                    'float16'
        }
```

[2]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
from sklearn.ensemble import RandomForestClassifier
```

[3]:
```
train_complete = pd.read_csv('microsoft-malware-prediction/train.csv')
train = train_complete[use_cols]
```

```
C:\Users\Gary\AppData\Roaming\Python\Python38\site-
packages\IPython\core\interactiveshell.py:3146: DtypeWarning: Columns (28) have
mixed types.Specify dtype option on import or set low_memory=False.
  has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
```

[4]:
```
train.shape
```

[4]:
```
(8921483, 39)
```

[5]:
```
train.head()
```

[5]:

	MachineIdentifier	SmartScreen	AVProductsInstalled	\
0	0000028988387b115f69f31a3bf04f09	NaN	1.0	
1	000007535c3f730efa9ea0b7ef1bd645	NaN	1.0	
2	000007905a28d863f6d0d597892cd692	RequireAdmin	1.0	
3	00000b11598a75ea8ba1beea8459149f	ExistsNotSet	1.0	
4	000014a5f00daa18e76b81417eeb99fc	RequireAdmin	1.0	

	AppVersion	CountryIdentifier	Census_OSInstallTypeName	Wdft_IsGamer	\
0	4.18.1807.18075	29	UUPUpgrade	0.0	
1	4.13.17134.1	93	IBSClean	0.0	
2	4.18.1807.18075	86	UUPUpgrade	0.0	
3	4.18.1807.18075	88	UUPUpgrade	0.0	
4	4.18.1807.18075	18	Update	0.0	

	EngineVersion	AVProductStatesIdentifier	Census_OSVersion	…	\
0	1.1.15100.1	53447.0	10.0.17134.165	…	
1	1.1.14600.4	53447.0	10.0.17134.1	…	
2	1.1.15100.1	53447.0	10.0.17134.165	…	
3	1.1.15100.1	53447.0	10.0.17134.228	…	
4	1.1.15100.1	53447.0	10.0.17134.191	…	

	Census_FirmwareManufacturerIdentifier	\

```
0                                  628.0
1                                  628.0
2                                  142.0
3                                  355.0
4                                  355.0

                                 OsBuildLab  Census_OSBuildRevision  \
0  17134.1.amd64fre.rs4_release.180410-1804                     165
1  17134.1.amd64fre.rs4_release.180410-1804                       1
2  17134.1.amd64fre.rs4_release.180410-1804                     165
3  17134.1.amd64fre.rs4_release.180410-1804                     228
4  17134.1.amd64fre.rs4_release.180410-1804                     191

   Census_OSBuildNumber  Census_IsPenCapable  Census_IsTouchEnabled  \
0                 17134                    0                      0
1                 17134                    0                      0
2                 17134                    0                      0
3                 17134                    0                      0
4                 17134                    0                      0

   Census_IsAlwaysOnAlwaysConnectedCapable  Census_IsSecureBootEnabled  \
0                                      0.0                           0
1                                      0.0                           0
2                                      0.0                           0
3                                      0.0                           0
4                                      0.0                           0

  Census_SystemVolumeTotalCapacity  HasDetections
0                         299451.0              0
1                         102385.0              0
2                         113907.0              0
3                         227116.0              1
4                         101900.0              1

[5 rows x 39 columns]
```

0.3 Section 2: Measure of Power (Q2a & 2b)

0.3.1 Section 2(a): Definition of Power as a Function of RAM and Processor Core Count

For the first part of task 2, I investigated the features in `use_cols` and considered which features are relevant to computer power. I found that the number of logical cores in the processor and the physical RAM are important factors to measure computer power. Therefore, I defined `ComputerPower` as a linear function of `Census_ProcessorCoreCount` and `Census_TotalPhysicalRAM`. That is, the population model is `ComputerPower` $= \beta_0 + \beta_1 \times$ `Census_ProcessorCoreCount` $+\beta_2 \times$ `Census_TotalPhysicalRAM`. To figure out the appropriate coefficients of the variables (β_0, β_1, and β_2), I used the OuterVision® Power Supply Calculator to help investigate the relationship

between the response and explanatory variables. To simplify the analysis, I assumed the following conditions for all machines: the type of device is Windows desktop; the CPU used for every machine is Intel Core i9-10980XE; the memory standard used to compute the load wattage supplies is double data rate third generation (DDR3); machine utilization time is 16 hours per day. Below are the descriptions for each variable.

`ComputerPower` - Load wattage power consumption in W

`Census_ProcessorCoreCount` - Number of logical cores in the processor

`Census_TotalPhysicalRAM` - Retrieves the physical RAM in MB

To begin, it is essential to look at the value types and the number of observations for each variable.

[6]:
```
df1 = train[['Census_ProcessorCoreCount', 'Census_TotalPhysicalRAM']]
```

[7]:
```
df1.value_counts()
```

[7]:
```
Census_ProcessorCoreCount  Census_TotalPhysicalRAM
4.0                        4096.0                     2670868
                           8192.0                     1619311
2.0                        4096.0                     1314953
                           2048.0                      636083
4.0                        2048.0                      431319
                                                       ...
2.0                        6047.0                           1
                           6051.0                           1
                           6052.0                           1
                           6055.0                           1
192.0                      196608.0                         1
Length: 5808, dtype: int64
```

[8]:
```
df1.isnull().sum()
```

[8]:
```
Census_ProcessorCoreCount    41306
Census_TotalPhysicalRAM      80533
dtype: int64
```

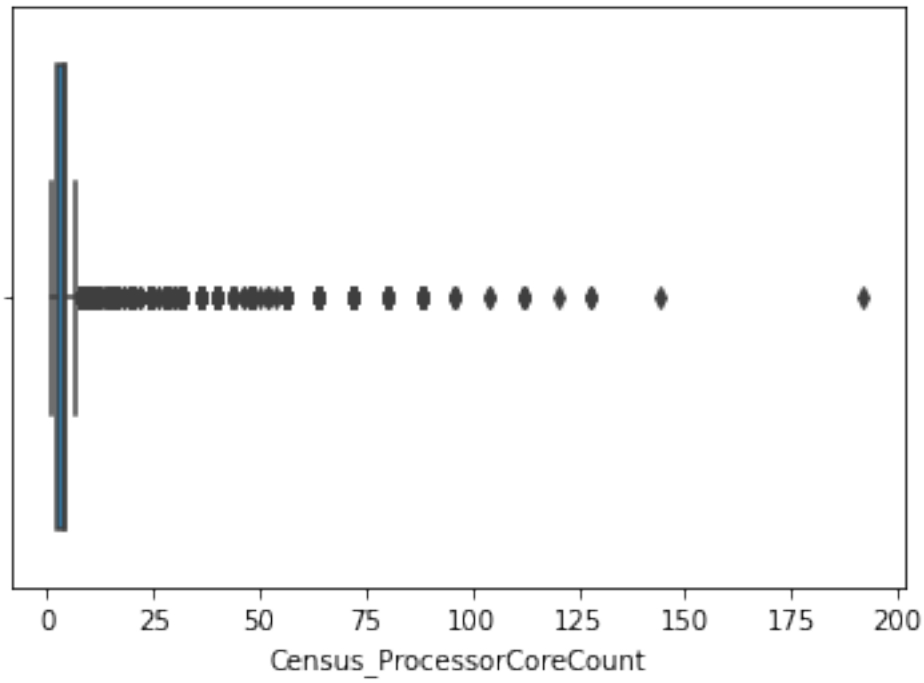
Note that there are 41306 and 80533 missing values in the variables `Census_ProcessorCoreCount` and `Census_TotalPhysicalRAM`, respectively. The total numbers of missing values are about 0.5% and 1.0%, respectively, of the total number of observations. Here, I considered univariate imputation methods to impute the missing values because of the small proportions of missing data. To further select the best imputation method for this task, it is useful to examine the boxplots of the two variables. Similar strategies will be used to decide on the imputation method for future tasks.

[9]:
```
sns.boxplot(df1['Census_ProcessorCoreCount'])
```

```
C:\Users\Gary\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning: Pass the following variable as a keyword arg: x. From version
0.12, the only valid positional argument will be `data`, and passing other
```

```
arguments without an explicit keyword will result in an error or
misinterpretation.
  warnings.warn(
```

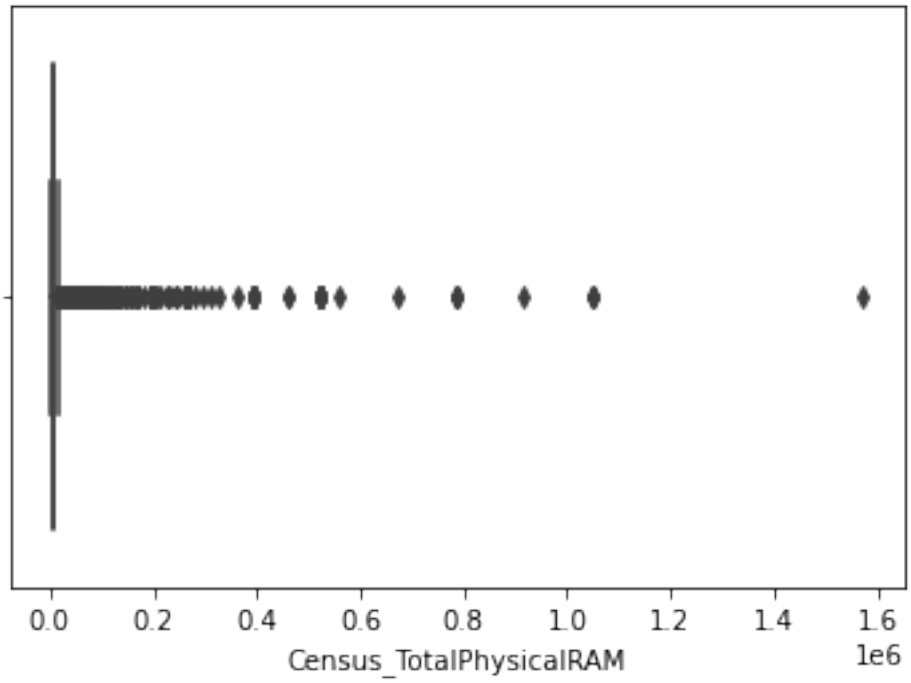
[9]: <AxesSubplot:xlabel='Census_ProcessorCoreCount'>



[10]:
```
sns.boxplot(df1['Census_TotalPhysicalRAM'])
```

```
C:\Users\Gary\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning: Pass the following variable as a keyword arg: x. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  warnings.warn(
```

[10]: <AxesSubplot:xlabel='Census_TotalPhysicalRAM'>



From the boxplots, it is obvious that the data for both variables are right-skewed (positively skewed). Therefore, it may not be appropriate to use mean imputation for the variables `Census_ProcessorCoreCount` and `Census_TotalPhysicalRAM`. Thus, I selected median imputation to fill the missing values.

[11]:
```
df1['Census_ProcessorCoreCount'].fillna(df1['Census_ProcessorCoreCount'].
 ↪median(), inplace = True)
df1['Census_TotalPhysicalRAM'].fillna(df1['Census_TotalPhysicalRAM'].median(),␣
 ↪inplace = True)
```

```
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4463:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().fillna(
```

It is better to verify again that there are no missing values in `Census_ProcessorCoreCount` and `Census_TotalPhysicalRAM`.

[12]:
```
df1.isnull().sum()
```

[12]:
```
Census_ProcessorCoreCount    0
Census_TotalPhysicalRAM      0
dtype: int64
```

`Census_ProcessorCoreCount` and `Census_TotalPhysicalRAM` are considered continuous variables (some may argue that they are ordinal variables. However, it does not matter a lot to consider them as continuous or ordinal for this task). After a careful analysis of the βs with the use of linear regression theory and trial and error, I obtained the parameter estimates: $\beta_0 = 65$, $\beta_1 = 156$, and $\beta_2 = 0.0006959$. Hence, I have the following final model for measuring `ComputerPower`:

$$\texttt{ComputerPower} = 65 + 156 \times \texttt{Census\_ProcessorCoreCount} + 0.0006959 \times \texttt{Census\_TotalPhysicalRAM}$$

Now, we can create a new variable `ComputerPower`, and calculate computer power for different machines.

[13]:
```
df1['ComputerPower'] = 65 + 156 * df1['Census_ProcessorCoreCount'] + 0.0006959␣
↪* df1['Census_TotalPhysicalRAM']
```

```
<ipython-input-13-5da376d9d602>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df1['ComputerPower'] = 65 + 156 * df1['Census_ProcessorCoreCount'] + 0.0006959
* df1['Census_TotalPhysicalRAM']
```

[14]:
```
df1['ComputerPower'].value_counts()
```

[14]:
```
691.850406     2728300
694.700813     1620594
379.850406     1329583
378.425203      636083
690.425203      431826
                …
700.375181           1
1632.125320          1
223.815611           1
1335.268104          1
381.240815           1
Name: ComputerPower, Length: 5827, dtype: int64
```

[15]:
```
q1 = df1['ComputerPower'].quantile(0.25)
q3 = df1['ComputerPower'].quantile(0.75)
iqr = q3 - q1
len(df1['ComputerPower'][(df1['ComputerPower'] > (q3 + 1.5 * iqr)) |␣
↪(df1['ComputerPower'] < (q1 - 1.5 * iqr))]) / len(df1)
```

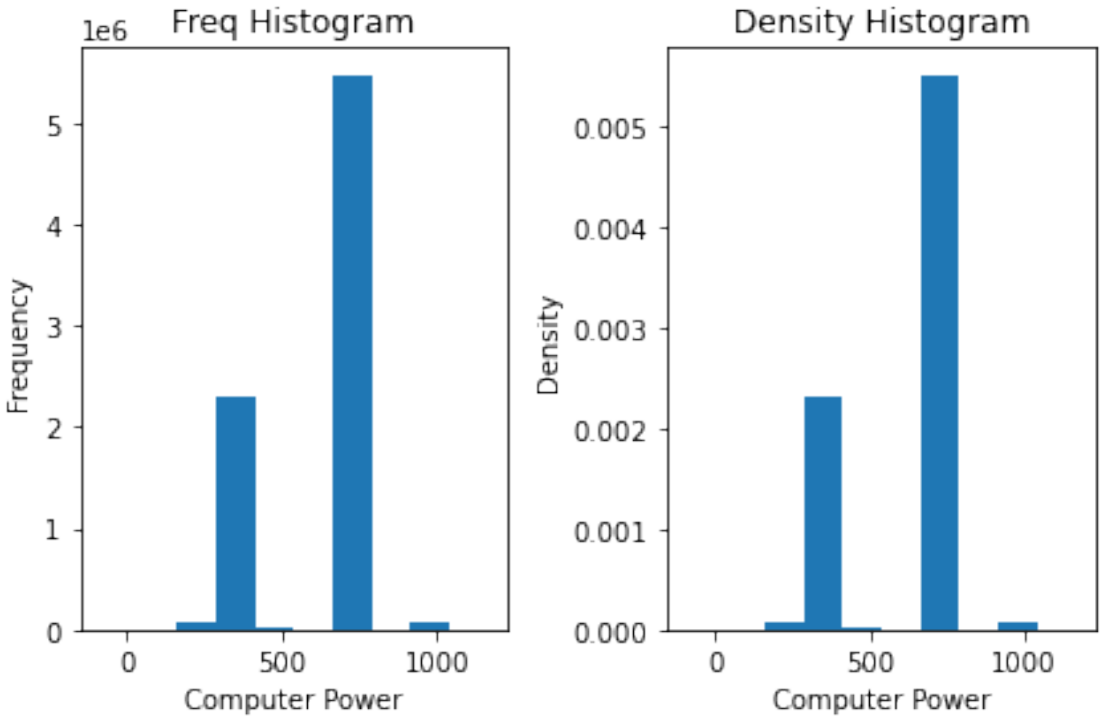
[15]:
```
0.1102800958091833
```

I noticed that the dataset contains a portion of outliers (about 11%). I used the traditional definition - Q1$-1.5\times$IQR and Q3$+1.5\times$IQR - as my classification of outliers, where Q1 and Q3 represent the

25% and 75% quantiles. IQR is the difference between Q3 and Q1. The outliers may affect the distribution of `ComputerPower`.

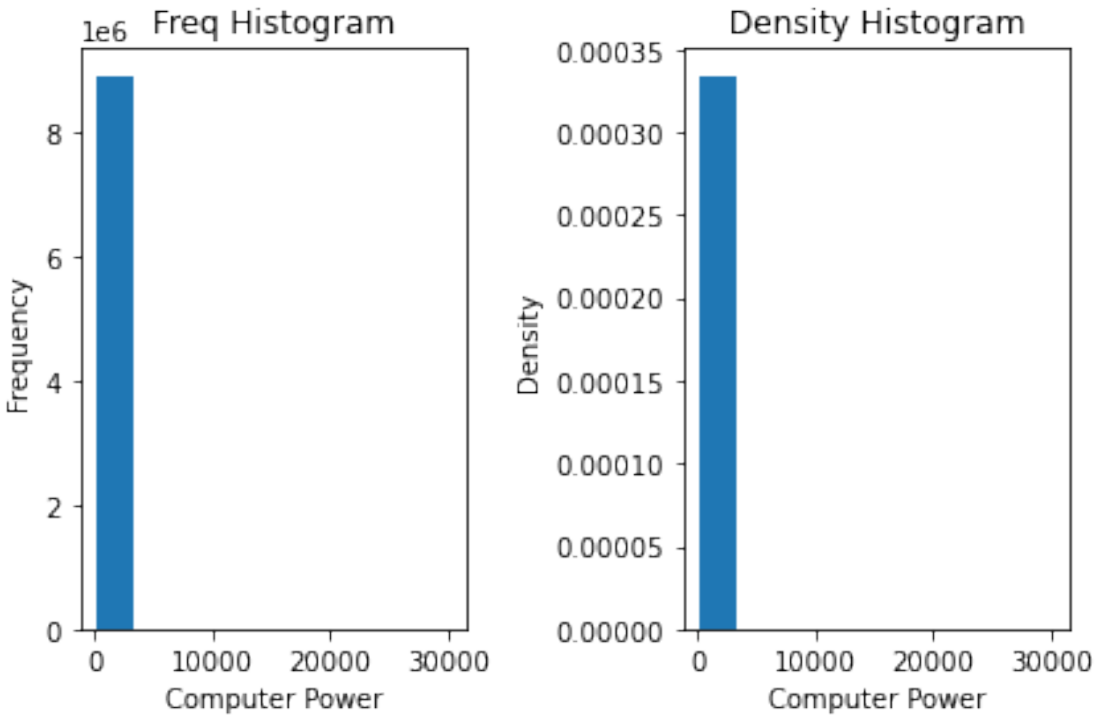
To properly visualize the distribution of computer power, I included two pairs of histograms. One pair is histograms without outliers, and the other pair is histograms with outliers. Each pair contains a frequency and a density histogram.

[16]:
```
plt.subplot(1, 2, 1)
plt.hist(df1['ComputerPower'], range = [q1 - 1.5 * iqr, q3 + 1.5 * iqr])
plt.xlabel('Computer Power')
plt.ylabel('Frequency')
plt.title('Freq Histogram')

plt.subplot(1, 2, 2)
plt.hist(df1['ComputerPower'], range = [q1 - 1.5 * iqr, q3 + 1.5 * iqr],␣
↪density = True)
plt.xlabel('Computer Power')
plt.ylabel('Density')
plt.title('Density Histogram')

plt.tight_layout()
```



[17]:
```
plt.subplot(1, 2, 1)
plt.hist(df1['ComputerPower'])
```

```
plt.xlabel('Computer Power')
plt.ylabel('Frequency')
plt.title('Freq Histogram')

plt.subplot(1, 2, 2)
plt.hist(df1['ComputerPower'], density = True)
plt.xlabel('Computer Power')
plt.ylabel('Density')
plt.title('Density Histogram')

plt.tight_layout()
```



[18]:
```
len(df1['ComputerPower'][(df1['ComputerPower'] < (q1 - 1.5 * iqr))])
```

[18]: 0

[19]:
```
len(df1['ComputerPower'][(df1['ComputerPower'] > (q3 + 1.5 * iqr))])
```

[19]: 983862

By Central Limit Theorem, the distribution with a reasonably large number of observations should be approximately normal. In this case, because I selected only two factors to measure computer power, where some of the machines have the same measure input, the mass of the distribution aggregates into a few bars, as shown in the histograms. Also, note that all outliers are above

Q3 based on the calculation above. Thus, the distribution is skewed to the right. I describe the distribution as a right-skewed bimodal distribution.

0.3.2 Section 2(b): Relationship of Computer Power vs Malware Detection

For the second part of the task, to investigate the relationship between machine power and its malware probability, I studied the variable `HasDetections` by looking into the value types and the number of observations.

`HasDetections` - the ground truth and indicates that Malware was detected on the machine.

[20]:
```
df1['HasDetections'] = train['HasDetections']
```

```
<ipython-input-20-8a4d37ec8336>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df1['HasDetections'] = train['HasDetections']
```

[21]:
```
df1['HasDetections'].value_counts()
```

[21]:
```
0    4462591
1    4458892
Name: HasDetections, dtype: int64
```

[22]:
```
df1['HasDetections'].isnull().sum()
```

[22]:
```
0
```

Note that `HasDetections` is a binary variable with no missing value. We do not need to clean the data for now. To measure the linear relationship between `ComputerPower` and `HasDetections`, I used the Pearson product-moment correlation coefficient.

[23]:
```
df1['ComputerPower'].corr(df1['HasDetections'])
```

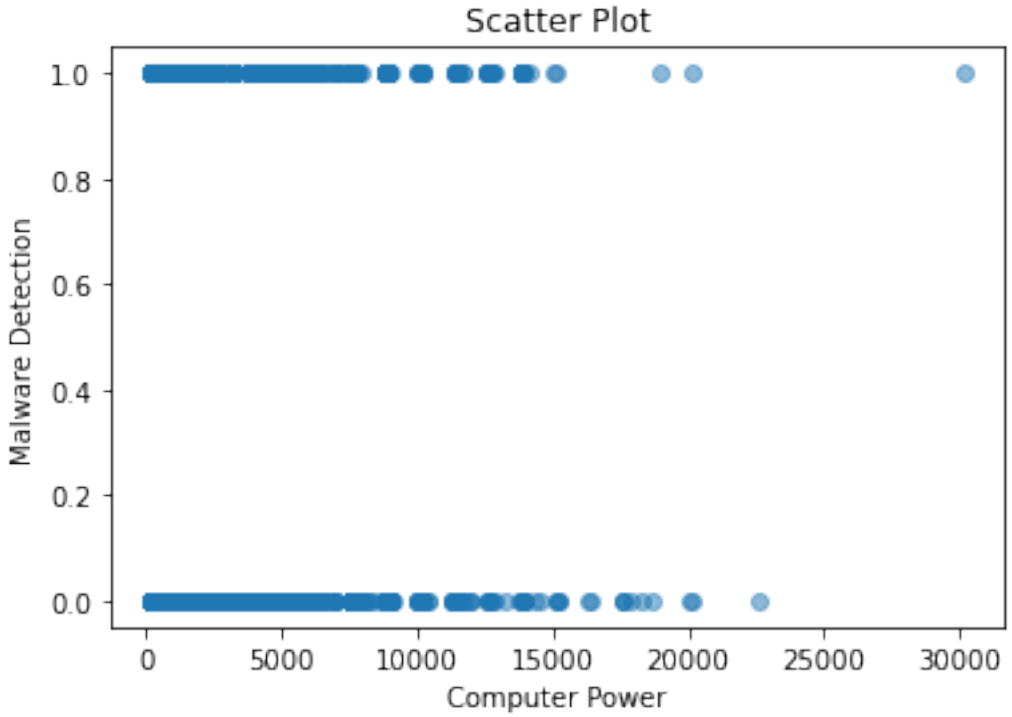
[23]:
```
0.05444230960877764
```

The correlation coefficient is about 0.054, very close to 0. To interpret this result, we can consider the two components of the correlation coefficient: strength and direction. From the strength perspective, note that 0.054 is close to 0. A coefficient of zero represents no linear relationship. Thus, as computer power increases, there is little tendency that the probability of malware will either increase or decrease. Moreover, the direction is positive does not give new information here since the strength is about 0. To be very rigorous, we can conclude that there is a very weak positive correlation between computer power and the probability of malware.

We can obtain a similar result through a scatter plot of the two variables. There is little tendency that as computer power increases will the probability of malware increases or decreases.

[24]:
```
plt.scatter(df1['ComputerPower'], df1['HasDetections'], alpha = 0.5)
plt.xlabel('Computer Power')
plt.ylabel('Malware Detection')
plt.title('Scatter Plot')
```

[24]:
```
Text(0.5, 1.0, 'Scatter Plot')
```



0.4 Section 3: OS version vs Malware detected (Q3)

We begin this task by overviewing the two related variables `Census_OSBuildNumber` and `Census_OSBuildRevision`. Note that the variable `HasDetections` was examined in Section 2(b).

`Census_OSBuildNumber` - OS Build number extracted from the OsVersionFull. Example - OsBuildNumber = 10512 or 10240

`Census_OSBuildRevision` - OS Build revision extracted from the OsVersionFull. Example - OsBuildRevision = 1000 or 16458

[25]:
```
df2 = train[['Census_OSBuildNumber', 'Census_OSBuildRevision']]
```

[26]:
```
df2.value_counts()
```

[26]:
```
Census_OSBuildNumber  Census_OSBuildRevision
17134                 228                      1413627
                      165                       899711
```

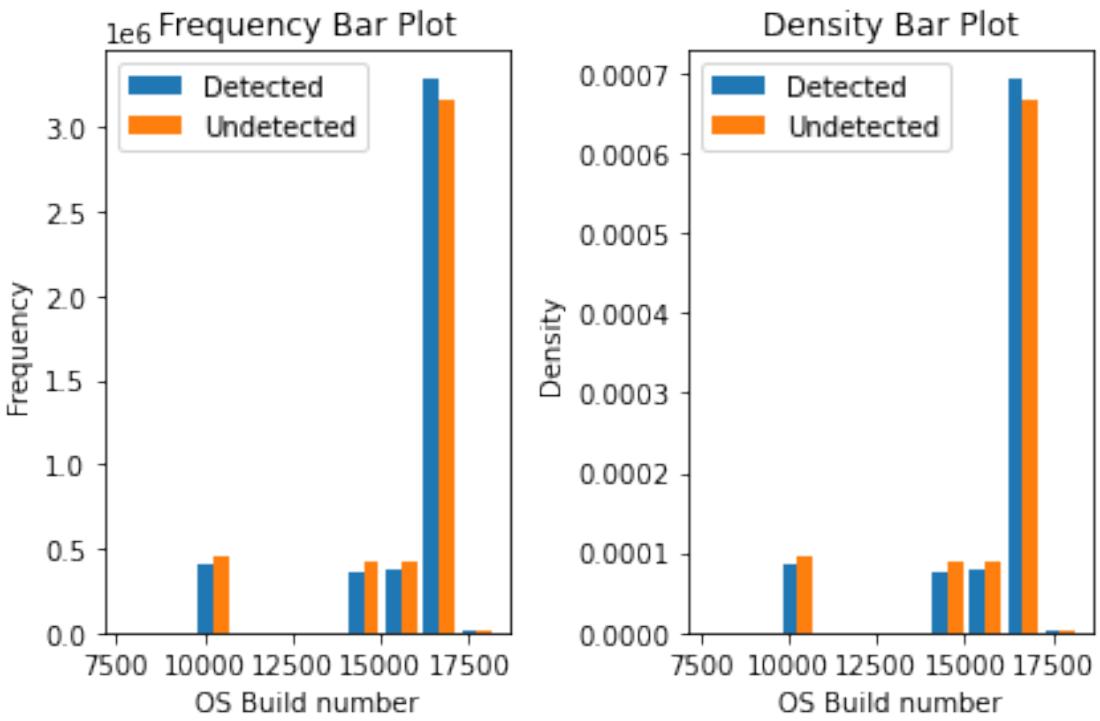
```
16299                  431                        546546
17134                  285                        470280
16299                  547                        346853
                                                  …
16179                  1000                            1
16199                  1000                            1
16215                  1000                            1
16237                  1001                            1
18244                  1000                            1
Length: 474, dtype: int64
```

[27]:
```
df2.isnull().sum()
```

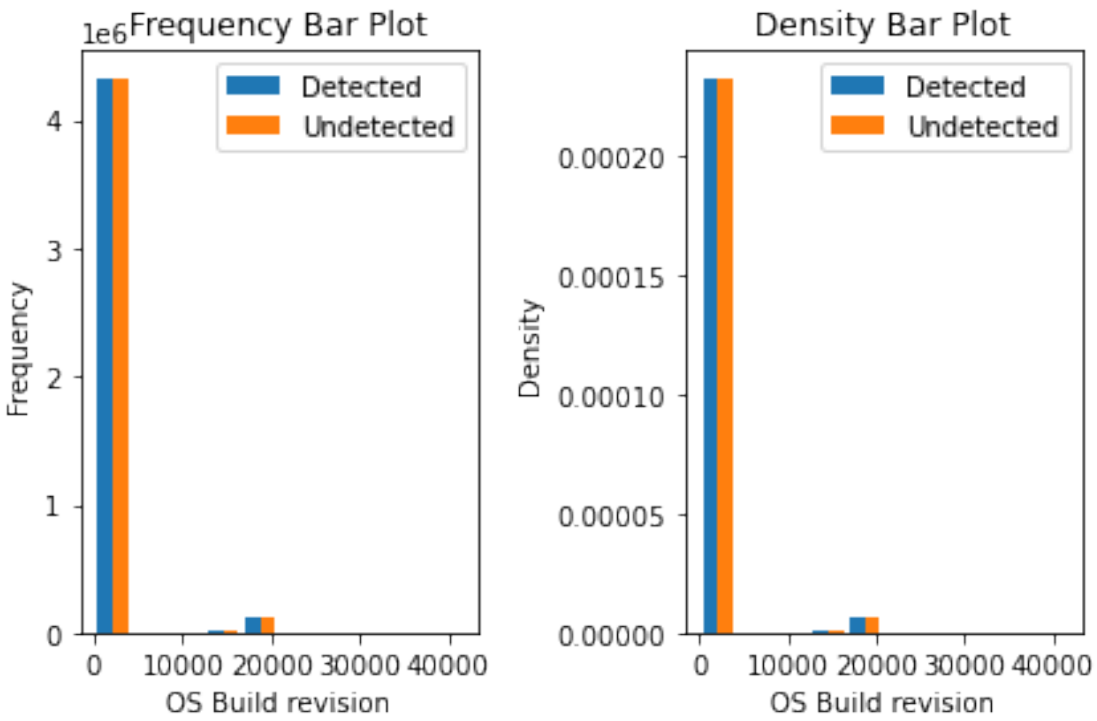
[27]:
```
Census_OSBuildNumber      0
Census_OSBuildRevision    0
dtype: int64
```

It is important to realize that software updates may fix vulnerabilities found in the past. However, it is also very likely that new vulnerabilities may arise after new feature updates are released. To understand whether software updates tend to increase or decrease the chances of malware attack, bar charts of the frequencies and densities of malware detection for every OS Build number and OS Build revision number are created separately for an examination.

[28]:
```
plt.subplot(1, 2, 1)
plt.hist([df2[train['HasDetections'] == 1]['Census_OSBuildNumber'],␣
 ↪df2[train['HasDetections'] == 0]['Census_OSBuildNumber']], label =␣
 ↪['Detected', 'Undetected'])
plt.xlabel('OS Build number')
plt.ylabel('Frequency')
plt.title('Frequency Bar Plot')
plt.legend(loc = 'upper left')

plt.subplot(1, 2, 2)
plt.hist([df2[train['HasDetections'] == 1]['Census_OSBuildNumber'],␣
 ↪df2[train['HasDetections'] == 0]['Census_OSBuildNumber']], label =␣
 ↪['Detected', 'Undetected'], density = True)
plt.xlabel('OS Build number')
plt.ylabel('Density')
plt.title('Density Bar Plot')
plt.legend(loc = 'upper left')

plt.tight_layout()
```



[29]:
```
plt.subplot(1, 2, 1)
plt.hist([df2[train['HasDetections'] == 1]['Census_OSBuildRevision'],␣
 ↪df2[train['HasDetections'] == 0]['Census_OSBuildRevision']], label =␣
 ↪['Detected', 'Undetected'])
plt.xlabel('OS Build revision')
plt.ylabel('Frequency')
plt.title('Frequency Bar Plot')
plt.legend(loc = 'upper right')

plt.subplot(1, 2, 2)
plt.hist([df2[train['HasDetections'] == 1]['Census_OSBuildRevision'],␣
 ↪df2[train['HasDetections'] == 0]['Census_OSBuildRevision']], label =␣
 ↪['Detected', 'Undetected'], density = True)
plt.xlabel('OS Build revision')
plt.ylabel('Density')
plt.title('Density Bar Plot')
plt.legend(loc = 'upper right')

plt.tight_layout()
```



Based on the two pairs of bar charts, I noticed the frequency difference of malware cases for different OS Build revisions to be approximately the same. It means that the revision updates may raise new malware that is vulnerable to about half of the machines and about half not. What is more, the frequency of the detection of malware is higher than the undetected malware for OS Build number greater or equal to 16000. On the other hand, the undetected malware frequency is higher than the detected frequency for OS Build number less than 16000. In my opinion, this suggests that new releases may worsen some of the machines because more proportions of machines were affected by malware compared to OS Build number releases less than 16000. To conclude, because the probability of malware detection did not decrease as new updates were released, I believe that updates created new vulnerabilities for machines.

0.5 Section 4: Effect of Number of AV Products Installed (Q4)

To finish the task, I first investigated the related variable `IsProtected` of antivirus software. I already checked the variable `HasDetections` in Section 2(b).

`IsProtected` - This is a calculated field derived from the Spynet Report's AV Products field. Returns: a. TRUE if there is at least one active and up-to-date antivirus product running on this machine. b. FALSE if there is no active AV product on this machine, or if the AV is active, but is not receiving the latest updates. c. null if there are no Anti Virus Products in the report. Returns: Whether a machine is protected.

[30]:
```
df3 = train[['IsProtected', 'HasDetections']]
```

[31]:
```
df3['IsProtected'].value_counts()
```

[31]:
```
1.0    8402282
0.0     483157
Name: IsProtected, dtype: int64
```

[32]:
```
df3['IsProtected'].isnull().sum()
```

[32]:
```
36044
```

Notice that `IsProtected` variable has three possible values: 1, 0, and NaN, where NaN here does not mean the value is missing, as illustrated in the above description. For this variable, I combined the return cases of b and c because I am interested in whether antivirus software(s) reduces the amount of malware. In other words, the difference between inactive and inexistent AV is not important. Therefore, returning categories b and c should be combined into a single case with a return value of 0.

[33]:
```
df3['IsProtected'].fillna(0.0, inplace = True)
```

```
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4463:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().fillna(
```

[34]:
```
df3['IsProtected'].value_counts()
```

[34]:
```
1.0    8402282
0.0     519201
Name: IsProtected, dtype: int64
```

[35]:
```
df3['IsProtected'].isnull().sum()
```

[35]:
```
0
```

To investigate the relationship between `IsProtected` and `HasDetections`, I used the Pearson product-moment correlation coefficient again to measure their linear relationship.

[36]:
```
df3['IsProtected'].corr(df3['HasDetections'])
```

[36]:
```
0.059080045666215955
```

The correlation coefficient is about 0.059, close to the correlation coefficient of computer power and malware detection in Section 2(b). Therefore, the interpretation is about the same: antivirus software(s) does not necessarily reduce the amount of malware. Furthermore, the number of antivirus products used does not matter because `IsProtected` is a binary variable denoting whether at least one or none (combined with the ones not receiving the latest updates), the actual number of antivirus software used, assuming there is at least one, does not change the correlation interpretation just obtained.

0.6 Section 5: Interesting findings (Q5)

As a beginner in computer hardware, I am interested in studying the specifications of desktops and laptops. One particular specification is display resolution. Display resolution is quoted as width (horizontal) × height (vertical), with the units in pixels. Most of the time, I observed that a large number of horizontal pixels will be paired with another large number of vertical pixels, comparatively speaking. For example, a 1920 × 1080 resolution versus a 1366 × 768 resolution. For this task, I will investigate the linear relationship between the two and see if it is strong enough so that I am convinced of the existence of a general rule.

Similar to the previous tasks, knowing the information about the interested variables is critical in exploring their relationship.

`Census_InternalPrimaryDisplayResolutionHorizontal` - Retrieves the number of pixels in the horizontal direction of the internal display.

`Census_InternalPrimaryDisplayResolutionVertical` - Retrieves the number of pixels in the vertical direction of the internal display

[37]:
```
df4 = train_complete[['Census_InternalPrimaryDisplayResolutionHorizontal',␣
↪'Census_InternalPrimaryDisplayResolutionVertical']]
```

[38]:
```
df4.value_counts()
```

[38]:
```
Census_InternalPrimaryDisplayResolutionHorizontal
Census_InternalPrimaryDisplayResolutionVertical
1366.0                                            768.0
4514227
1920.0                                            1080.0
2134006
1600.0                                            900.0
490012
1024.0                                            768.0
296935
1280.0                                            800.0
261728
                                  …
1518.0                                            918.0
1
                                                  880.0
1
                                                  844.0
1
                                                  832.0
1
1600.0                                            849.0
1
Length: 10067, dtype: int64
```

[39]:
```
df4.isnull().sum()
```

[39]:
```
Census_InternalPrimaryDisplayResolutionHorizontal    46986
Census_InternalPrimaryDisplayResolutionVertical      46986
dtype: int64
```

Note that there are 46986 missing values in each of the variables. I verified that the missing values are paired so that either a machine has data for both variables or none. Since only a small portion of the paired resolution data is missing (about 0.5%), I will drop them for further analysis.

[40]:
```
df4.dropna(how = 'any', inplace = True)
```

```
<ipython-input-40-d3671fa1fa56>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df4.dropna(how = 'any', inplace = True)
```

[41]:
```
df4.isnull().sum()
```

[41]:
```
Census_InternalPrimaryDisplayResolutionHorizontal    0
Census_InternalPrimaryDisplayResolutionVertical      0
dtype: int64
```

[42]:
```
df4['Census_InternalPrimaryDisplayResolutionHorizontal'].
↪corr(df4['Census_InternalPrimaryDisplayResolutionVertical'])
```

[42]:
```
0.9015470699877536
```

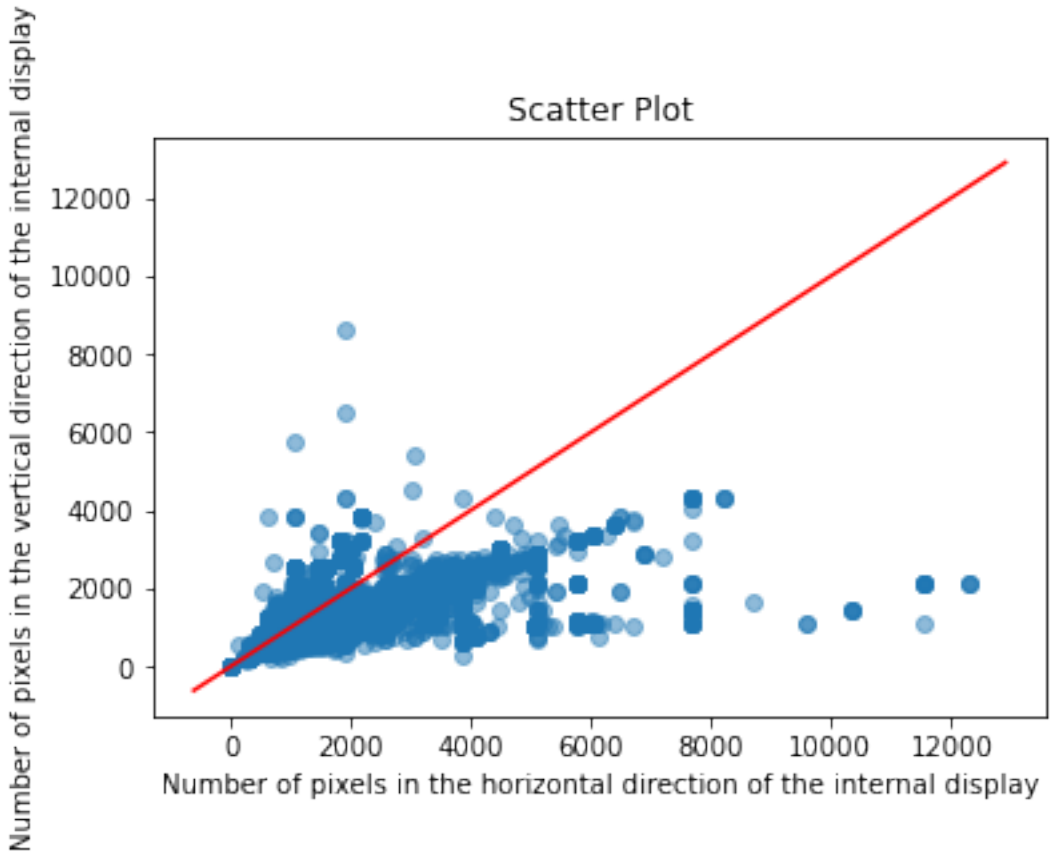
The correlation coefficient is about 0.902, very close to 1. Again, we can consider the two components of the correlation coefficient to interpret its meaning: strength and direction. From the strength perspective, because 0.902 is close to 1, an absolute value of the coefficient close to 1 represents a nearly perfect linear relationship. Moreover, the direction is positive implies that a linear equation with a positive slope can describe the relationship between the number of pixels in the horizontal and vertical direction of the internal display.

I also included a scatter plot of `Census_InternalPrimaryDisplayResolutionHorizontal` and `Census_InternalPrimaryDisplayResolutionVertical` with a straight line of $y = x$ to visualize the relationship intuitively. Except for a limited number of potential outliers or uncommon points, most points match the conclusion made by the correlation interpretation.

[43]:
```
plt.scatter(df4['Census_InternalPrimaryDisplayResolutionHorizontal'],␣
↪df4['Census_InternalPrimaryDisplayResolutionVertical'], alpha = 0.5)
plt.xlabel('Number of pixels in the horizontal direction of the internal␣
↪display')
plt.ylabel('Number of pixels in the vertical direction of the internal display')
plt.title('Scatter Plot')
```

```
axes = plt.gca()
x_vals = np.array(axes.get_xlim())
y_vals = 0 + 1 * x_vals
plt.plot(x_vals, y_vals, 'r-')
```

[43]: [<matplotlib.lines.Line2D at 0x22094a8e040>]



0.7 Section 6: Baseline modelling (Q6)

To build a simple baseline model, Model 0, I selected the feature `AVProductsInstalled` based on my research in malware statistics to predict the probability of malware. The malware detection is denoted as variable `HasDetections` in Section 2(b).

`AVProductsInstalled` - Number of antivirus software installed

Before proceeding to inspect the data for `AVProductsInstalled`, it is important to split the data into a train data and a test data. For this task and the tasks in Section 7, I will train models on 80% of the training data and test it on the remaining 20% chosen at random.

[44]:
```
X, y = train.iloc[:, :-1], train.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,␣
 ↪random_state = 123)
```

For this particular task, I am only interested in the feature `AVProductsInstalled`.

[45]:
```
X_train0 = X_train['AVProductsInstalled']
X_test0 = X_test['AVProductsInstalled']
```

Now, we can examine the data and see if missing values need to be filled. Remark that it is unnecessary to examine the variable separately in the train set and the test set because the split of the dataset is random.

[46]:
```
X['AVProductsInstalled'].value_counts()
```

[46]:
```
1.0    6208893
2.0    2459008
3.0     208103
4.0       8757
5.0        471
6.0         28
0.0          1
7.0          1
Name: AVProductsInstalled, dtype: int64
```

[47]:
```
X['AVProductsInstalled'].isnull().sum()
```

[47]:
```
36221
```

Note that `AVProductsInstalled` contains integer values from 0 to 7. There are about 0.4% of missing data in total for this variable. I used an imputation method here rather than listwise deletion to deal with the missing values. Moreover, because the variable values are mostly 1 or 2, I used mode imputation here to proceed.

[48]:
```
X_train0.fillna(X['AVProductsInstalled'].mode()[0], inplace = True)
X_test0.fillna(X['AVProductsInstalled'].mode()[0], inplace = True)
```

```
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4463:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().fillna(
```

[49]:
```
X_train0.isnull().sum()
```

[49]:
```
0
```

[50]:
```
X_test0.isnull().sum()
```

[50]:
```
0
```

Note that for this baseline model Model 0, I only included one feature to train the logistic regression. Given the documentation of `sklearn.linear_model.LogisticRegression`:

> fit(X, y, sample_weight=None)
>
> **Parameters: X: {array-like, sparse matrix} of shape (n_samples, n_features)**
>
> Training vector, where n_samples is the number of samples and n_features is the number of features.
>
> predict(X)
>
> **Parameters: X: {array-like, sparse matrix} of shape (n_samples, n_features)**
>
> Samples.

The input arrays X for the two functions above should be 2 dimensional. Thus, I used `reshape()` function below to change the dimension of `X_train0` in the train and test datasets.

[51]:
```
model_0 = LogisticRegression()
model_0.fit(X_train0.values.reshape(-1, 1), y_train)
```

[51]:
```
LogisticRegression()
```

[52]:
```
y_predict0 = model_0.predict(X_test0.values.reshape(-1, 1))
(y_test != y_predict0).mean()
```

[52]:
```
0.4331767637338403
```

The error rate is about 43.32% for this model.

[53]:
```
roc_auc_score(y_test, y_predict0)
```

[53]:
```
0.5668461485553381
```

The AUC score of this model is about 0.567.

0.8 Section 7: Feature Cleaning and Additional models (Q7a & 7b)

0.8.1 Section 7(a): Cleaning Features

To build a more sophisticated and useful model, I added more features that I believe will be relevant to the model for training. In particular, I have included variables `RtpStateBitfield`, `AVProductsInstalled` (inspected in Section 6), `IsProtected` (inspected in Section 4), `Census_PrimaryDiskTotalCapacity`, `Census_PrimaryDiskTypeName`, `Census_IsSecureBootEnabled`, `Census_IsAlwaysOnAlwaysConnectedCapable`, and `Wdft_IsGamer`. Also, note that the target variable `HasDetections` was already checked in

Section 2(b). Similar to the procedures done in previous tasks, it is essential to first understand the meaning and the value types of the variables.

`RtpStateBitfield` - RTP state

`Census_PrimaryDiskTotalCapacity` - Amount of disk space on primary disk of the machine in MB

`Census_PrimaryDiskTypeName` - Friendly name of Primary Disk Type - HDD or SSD

`Census_IsSecureBootEnabled` - Indicates if Secure Boot mode is enabled.

`Census_IsAlwaysOnAlwaysConnectedCapable` - Retreives information about whether the battery enables the device to be AlwaysOnAlwaysConnected.

`Wdft_IsGamer` - Indicates whether the device is a gamer device or not based on its hardware combination.

[54]:
```
X_train1 = X_train[['RtpStateBitfield', 'AVProductsInstalled', 'IsProtected',␣
↪'Census_PrimaryDiskTotalCapacity', 'Census_PrimaryDiskTypeName',␣
↪'Census_IsSecureBootEnabled', 'Census_IsAlwaysOnAlwaysConnectedCapable',␣
↪'Wdft_IsGamer']]
X_test1 = X_test[['RtpStateBitfield', 'AVProductsInstalled', 'IsProtected',␣
↪'Census_PrimaryDiskTotalCapacity', 'Census_PrimaryDiskTypeName',␣
↪'Census_IsSecureBootEnabled', 'Census_IsAlwaysOnAlwaysConnectedCapable',␣
↪'Wdft_IsGamer']]
```

Similar to the data analysis procedure in Section 6, we do not have to inspect the variables individually for the train and test set.

[55]:
```
X['RtpStateBitfield'].value_counts()
```

[55]:
```
7.0     8651487
0.0      190701
8.0       21974
5.0       20328
3.0        3029
1.0        1625
35.0         21
Name: RtpStateBitfield, dtype: int64
```

[56]:
```
X['Census_PrimaryDiskTotalCapacity'].value_counts()
```

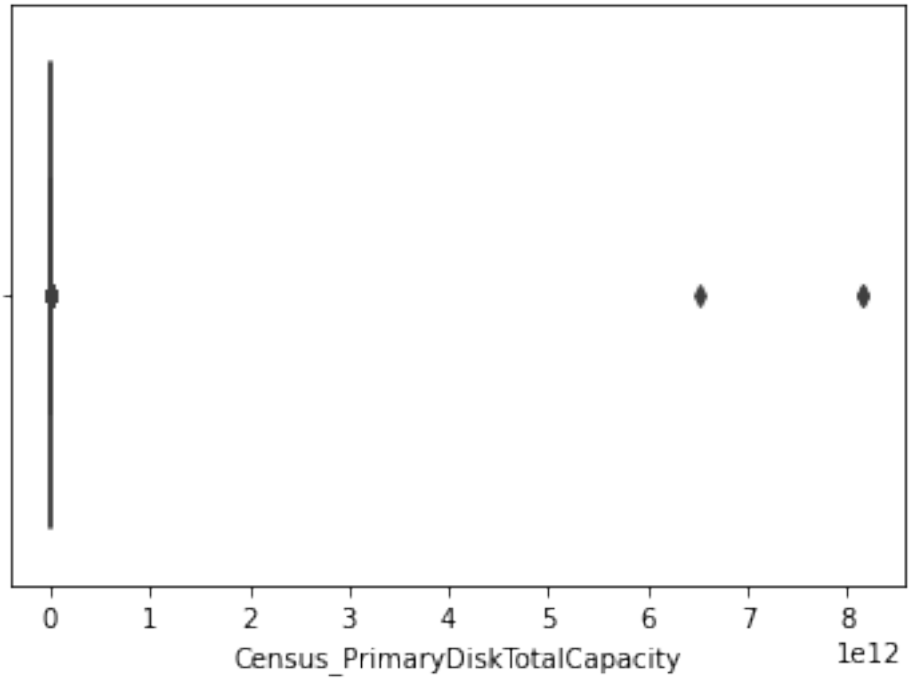
[56]:
```
476940.0    2841530
953869.0    2175780
305245.0     474616
122104.0     469060
244198.0     452284
             …
610424.0          1
480648.0          1
```

```
20518.0           1
117315.0          1
61696.0           1
Name: Census_PrimaryDiskTotalCapacity, Length: 5735, dtype: int64
```

[57]:
```
sns.boxplot(X['Census_PrimaryDiskTotalCapacity'])
```

```
C:\Users\Gary\anaconda3\lib\site-packages\seaborn\_decorators.py:36:
FutureWarning: Pass the following variable as a keyword arg: x. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
  warnings.warn(
```

[57]:
```
<AxesSubplot:xlabel='Census_PrimaryDiskTotalCapacity'>
```



[58]:
```
X['Census_PrimaryDiskTypeName'].value_counts()
```

[58]:
```
HDD            5806804
SSD            2466808
UNKNOWN         358251
Unspecified     276776
Name: Census_PrimaryDiskTypeName, dtype: int64
```

[59]:
```
X['Census_IsSecureBootEnabled'].value_counts()
```

[59]:
```
0    4585438
1    4336045
Name: Census_IsSecureBootEnabled, dtype: int64
```

[60]:
```
X['Census_IsAlwaysOnAlwaysConnectedCapable'].value_counts()
```

[60]:
```
0.0    8341972
1.0     508168
Name: Census_IsAlwaysOnAlwaysConnectedCapable, dtype: int64
```

[61]:
```
X['Wdft_IsGamer'].value_counts()
```

[61]:
```
0.0    6174143
1.0    2443889
Name: Wdft_IsGamer, dtype: int64
```

[62]:
```
X[['RtpStateBitfield', 'AVProductsInstalled', 'IsProtected',␣
 ↪'Census_PrimaryDiskTotalCapacity', 'Census_PrimaryDiskTypeName',␣
 ↪'Census_IsSecureBootEnabled', 'Census_IsAlwaysOnAlwaysConnectedCapable',␣
 ↪'Wdft_IsGamer']].isnull().sum()
```

[62]:
```
RtpStateBitfield                        32318
AVProductsInstalled                     36221
IsProtected                             36044
Census_PrimaryDiskTotalCapacity         53016
Census_PrimaryDiskTypeName              12844
Census_IsSecureBootEnabled                  0
Census_IsAlwaysOnAlwaysConnectedCapable 71343
Wdft_IsGamer                           303451
dtype: int64
```

Given the above variables data types and the number of missing value observations, I used similar analysis procedures as the previous tasks to impute the missing values for each variable. I will impute `RtpStateBitfield`, `Census_PrimaryDiskTotalCapacity`, `Census_PrimaryDiskTypeName`, `Census_IsAlwaysOnAlwaysConnectedCapable`, and `Wdft_IsGamer` using mode, median, random sample (70% HDD, 30% SSD), random sample (94% 0, 6% 1), and random sample (70% 0, 30% 1) imputation, respectively. Note that `AVProductsInstalled` and `IsProtected` can be imputed using their respective techniques discussed in Section 6 and Section 4. There is no missing value for `Census_IsSecureBootEnabled`.

[63]:
```
X_train1['RtpStateBitfield'].fillna(X['RtpStateBitfield'].mode()[0], inplace =␣
 ↪True)
X_test1['RtpStateBitfield'].fillna(X['RtpStateBitfield'].mode()[0], inplace =␣
 ↪True)

X_train1['AVProductsInstalled'].fillna(X['AVProductsInstalled'].mode()[0],␣
 ↪inplace = True)
```

```
X_test1['AVProductsInstalled'].fillna(X['AVProductsInstalled'].mode()[0],␣
 ↪inplace = True)

X_train1['IsProtected'].fillna(0.0, inplace = True)
X_test1['IsProtected'].fillna(0.0, inplace = True)

X_train1['Census_PrimaryDiskTotalCapacity'].
 ↪fillna(X['Census_PrimaryDiskTotalCapacity'].median(), inplace = True)
X_test1['Census_PrimaryDiskTotalCapacity'].
 ↪fillna(X['Census_PrimaryDiskTotalCapacity'].median(), inplace = True)

np.random.seed(123)
X_train1['Census_PrimaryDiskTypeName'].fillna(np.random.choice(('HDD', 'SSD'),␣
 ↪1, p = [0.7, 0.3])[0], inplace = True)
np.random.seed(124)
X_test1['Census_PrimaryDiskTypeName'].fillna(np.random.choice(('HDD', 'SSD'),␣
 ↪1, p = [0.7, 0.3])[0], inplace = True)
np.random.seed(125)
X_train1['Census_PrimaryDiskTypeName'].replace(['UNKNOWN', 'Unspecified'], np.
 ↪random.choice(('HDD', 'SSD'), 1, p = [0.7, 0.3])[0], inplace = True)
np.random.seed(126)
X_test1['Census_PrimaryDiskTypeName'].replace(['UNKNOWN', 'Unspecified'], np.
 ↪random.choice(('HDD', 'SSD'), 1, p = [0.7, 0.3])[0], inplace = True)
X_train1['Census_PrimaryDiskTypeName'].replace(['HDD', 'SSD'], [0, 1], inplace␣
 ↪= True)
X_test1['Census_PrimaryDiskTypeName'].replace(['HDD', 'SSD'], [0, 1], inplace =␣
 ↪True)

np.random.seed(127)
X_train1['Census_IsAlwaysOnAlwaysConnectedCapable'].fillna(np.random.choice((0,␣
 ↪1), 1, p = [0.94, 0.06])[0], inplace = True)
np.random.seed(128)
X_test1['Census_IsAlwaysOnAlwaysConnectedCapable'].fillna(np.random.choice((0,␣
 ↪1), 1, p = [0.94, 0.06])[0], inplace = True)

np.random.seed(129)
X_train1['Wdft_IsGamer'].fillna(np.random.choice((0, 1), 1, p = [0.7, 0.3])[0],␣
 ↪inplace = True)
np.random.seed(130)
X_test1['Wdft_IsGamer'].fillna(np.random.choice((0, 1), 1, p = [0.7, 0.3])[0],␣
 ↪inplace = True)
```

```
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4463:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().fillna(
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4509:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().replace(
```

After the data cleaning process is done, I verified the data types are correctly transformed, and no missing values are present. Here, I only listed the value counts for `Census_PrimaryDiskTypeName` because I did some encoding to this variable, whereas only imputations were done for the rest of the variables.

[64]:
```
X_train1['Census_PrimaryDiskTypeName'].value_counts()
```

[64]:
```
0    5163884
1    1973302
Name: Census_PrimaryDiskTypeName, dtype: int64
```

[65]:
```
X_test1['Census_PrimaryDiskTypeName'].value_counts()
```

[65]:
```
0    1290791
1     493506
Name: Census_PrimaryDiskTypeName, dtype: int64
```

[66]:
```
X_train1.isnull().sum()
```

[66]:
```
RtpStateBitfield                        0
AVProductsInstalled                     0
IsProtected                             0
Census_PrimaryDiskTotalCapacity         0
Census_PrimaryDiskTypeName              0
Census_IsSecureBootEnabled              0
Census_IsAlwaysOnAlwaysConnectedCapable 0
Wdft_IsGamer                            0
dtype: int64
```

[67]:
```
X_test1.isnull().sum()
```

[67]:
```
RtpStateBitfield                        0
AVProductsInstalled                     0
IsProtected                             0
Census_PrimaryDiskTotalCapacity         0
Census_PrimaryDiskTypeName              0
Census_IsSecureBootEnabled              0
```

```
Census_IsAlwaysOnAlwaysConnectedCapable    0
Wdft_IsGamer                               0
dtype: int64
```

Usually, it is critical to normalize the variables so that the ranges and distributions of the variables are comparable. However, after a careful inspection of the value types and ranges of the features, I did not perform the normalization process due to the following reasons. First, of the eight features, only three variables are considered continuous, and the rest are binary variables with only 0 and 1 values. It is unnecessary to perform normalization on the dummy variables. Among the three continuous variables, `RtpStateBitfield` and `AVProductsInstalled` contain data values mostly small integers from 0 to 8. Performing such normalization may not benefit much. Finally, normalizing variable `Census_PrimaryDiskTotalCapacity` can potentially raise computing overflow errors because the normalized values are extremely small (e.g. 1.0e-07), which provides a worse matrix for the model training procedure. Therefore, the model training performs better without normalizing the variables.

0.8.2 Section 7(b): Final Model Creation

Section 7(b)(i): Logistic Regression Before training the final model, feature selection is preferred to reduce the number of input variables. The predictive model can have a lower error rate if some redundant features are not included in the training process. Here, I used a univariate feature selection method called `SelectKBest` to remove all features except the k highest scoring features. In particular, the scoring function used is the sample's ANOVA F-value.

[68]:
```
test = SelectKBest(score_func = f_classif, k = 3)
fit = test.fit(X_train1, y_train)
print(fit.scores_)
```

```
[1.22219333e+04 1.61044267e+05 2.50733990e+04 2.06201921e-02
 7.39617775e+01 2.32395658e+01 2.82209683e+04 1.93266437e+04]
```

For this particular task, I chose to have the highest three scoring features ($k = 3$) as the final features for training. The three highest features are `AVProductsInstalled`, `IsProtected`, and `Census_IsAlwaysOnAlwaysConnectedCapable`. Hence, we can proceed to do the logistic regression training. Name the model Model 1.

[69]:
```
X_train1 = X_train1[['AVProductsInstalled', 'IsProtected',␣
↪'Census_IsAlwaysOnAlwaysConnectedCapable']]
X_test1 = X_test1[['AVProductsInstalled', 'IsProtected',␣
↪'Census_IsAlwaysOnAlwaysConnectedCapable']]
```

[70]:
```
model_1 = LogisticRegression()
model_1.fit(X_train1, y_train)
```

[70]:
```
LogisticRegression()
```

[71]:
```
y_predict1 = model_1.predict(X_test1)
(y_test != y_predict1).mean()
```

[71]:
```
0.4171872731949894
```

The error rate is about 41.72% for the logistic regression model.

Section 7(b)(ii): Random Forest For random forest training, I used the same three features for training, with the number of trees in the forest set to be 100 by default. The training process is similar to the logistic regression training. Name the model Model 2.

[72]:
```
model_2 = RandomForestClassifier()
model_2.fit(X_train1, y_train)
```

[72]:
```
RandomForestClassifier()
```

[73]:
```
y_predict2 = model_2.predict(X_test1)
(y_test != y_predict2).mean()
```

[73]:
```
0.4171872731949894
```

The error rate is about 41.72% for the random forest model.

0.8.3 Summary

Here is a table summarizing the error rates for the three models built in Section 6 and Section 7(b).

Model	Error Rate
Model 0	43.32%
Model 1	41.72%
Model 2	41.72%

To compare the performance of the models, I will start with the difference in terms of the error rate. Model 0 only uses `AVProductsInstalled` as the feature for training. Model 1 and 2 use three features for training: `AVProductsInstalled`, `IsProtected`, and `Census_IsAlwaysOnAlwaysConnectedCapable`. Therefore, adding more relevant features to the model decreases the error rate of prediction. This is intuitive because there are more useful resources for training so that it is very possible to have a better prediction.

Furthermore, observing that no difference in error rate between Model 1 and Model 2, I conclude that there is no difference in prediction error rate using logistic regression algorithm and random forest algorithm. I conjecture the reason is that logistic regression predicts similar to the random forest for large datasets.

0.9 Section 8: Screenshots (Q8)

Last but not least, to predict the data in test.csv using the three models, we first import the test.csv dataset.

[74]:
```
test = pd.read_csv('microsoft-malware-prediction/test.csv')
```

```
C:\Users\Gary\AppData\Roaming\Python\Python38\site-
packages\IPython\core\interactiveshell.py:3146: DtypeWarning: Columns (28) have
mixed types.Specify dtype option on import or set low_memory=False.
  has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
```

Model 0 prediction:

[75]:
```
X_test_0 = test['AVProductsInstalled']
X_test_0.fillna(test['AVProductsInstalled'].mode()[0], inplace = True)
y_predict_test_0 = model_0.predict(X_test_0.values.reshape(-1, 1))
result = test['MachineIdentifier']
result_0 = pd.concat([result, pd.DataFrame(y_predict_test_0, columns =␣
↪['HasDetections'])], axis = 1)
result_0.to_csv('model_0.csv', index = False)
```

Model 1 prediction:

[76]:
```
X_test_1 = test[['AVProductsInstalled', 'IsProtected',␣
↪'Census_IsAlwaysOnAlwaysConnectedCapable']]
X_test_1['AVProductsInstalled'].fillna(test['AVProductsInstalled'].mode()[0],␣
↪inplace = True)
X_test_1['IsProtected'].fillna(0.0, inplace = True)
X_test_1['Census_IsAlwaysOnAlwaysConnectedCapable'].fillna(np.random.choice((0,␣
↪1), 1, p = [0.94, 0.06])[0], inplace = True)
y_predict_test_1 = model_1.predict(X_test_1)
result_1 = pd.concat([result, pd.DataFrame(y_predict_test_1, columns =␣
↪['HasDetections'])], axis = 1)
result_1.to_csv('model_1.csv', index = False)
```

```
C:\Users\Gary\anaconda3\lib\site-packages\pandas\core\series.py:4463:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().fillna(
```

Model 2 prediction:

[77]:
```
y_predict_test_2 = model_2.predict(X_test_1)
result_2 = pd.concat([result, pd.DataFrame(y_predict_test_2, columns =␣
↪['HasDetections'])], axis = 1)
result_2.to_csv('model_2.csv', index = False)
```

Public Score: 0.57462

Private Score: 0.52528

Kaggle profile link: https://www.kaggle.com/garylikai

Screenshot(s):