Final-Assignment

July 10, 2024

Nastaran Qalandari - Karen Panahi - Maede Majidi - Maryam Fendereski

Table of contents

1 Importing Required Libraries

```
[ ]: import warnings
     import pandas as pd
     import seaborn as sns
     import matplotlib.pyplot as plt
     from sklearn.feature_selection import mutual_info_classif
     from sklearn.model_selection import train_test_split
     from sklearn.model_selection import GridSearchCV
     from sklearn.metrics import classification_report
     import time
     from sklearn.linear_model import LogisticRegression
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.naive_bayes import GaussianNB
     from sklearn.tree import DecisionTreeClassifier
     from sklearn.ensemble import AdaBoostClassifier
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.svm import LinearSVC
     from sklearn.svm import SVC
```

```
[ ]: warnings.filterwarnings('ignore')
```

2 Loading Dataset

```
[ ]: from ucimlrepo import fetch_ucirepo

     spambase = fetch_ucirepo(id=94)

     X = spambase.data.features
     y = spambase.data.targets

     print(spambase.metadata)
     print(spambase.variables)
```


```
{'uci_id': 94, 'name': 'Spambase', 'repository_url':
'https://archive.ics.uci.edu/dataset/94/spambase', 'data_url':
'https://archive.ics.uci.edu/static/public/94/data.csv', 'abstract':
'Classifying Email as Spam or Non-Spam', 'area': 'Computer Science', 'tasks':
['Classification'], 'characteristics': ['Multivariate'], 'num_instances': 4601,
'num_features': 57, 'feature_types': ['Integer', 'Real'], 'demographics': [],
'target_col': ['Class'], 'index_col': None, 'has_missing_values': 'no',
'missing_values_symbol': None, 'year_of_dataset_creation': 1999, 'last_updated':
'Mon Aug 28 2023', 'dataset_doi': '10.24432/C53G6X', 'creators': ['Mark
Hopkins', 'Erik Reeber', 'George Forman', 'Jaap Suermondt'], 'intro_paper':
None, 'additional_info': {'summary': 'The "spam" concept is diverse:
advertisements for products/web sites, make money fast schemes, chain letters,
pornography…\n\nThe classification task for this dataset is to determine
```

```
whether a given email is spam or not.\n\t\nOur collection of spam e-mails came
from our postmaster and individuals who had filed spam.  Our collection of non-
spam e-mails came from filed work and personal e-mails, and hence the word
\'george\' and the area code \'650\' are indicators of non-spam.  These are
useful when constructing a personalized spam filter.  One would either have to
blind such non-spam indicators or get a very wide collection of non-spam to
generate a general purpose spam filter.\n\nFor background on spam: Cranor,
Lorrie F., LaMacchia, Brian A.  Spam!, Communications of the ACM, 41(8):74-83,
1998.\n\nTypical performance is around ~7% misclassification error. False
positives (marking good mail as spam) are very undesirable.If we insist on zero
false positives in the training/testing set, 20-25% of the spam passed through
the filter. See also Hewlett-Packard Internal-only Technical Report. External
version forthcoming. ', 'purpose': None, 'funded_by': None,
'instances_represent': 'Emails', 'recommended_data_splits': None,
'sensitive_data': None, 'preprocessing_description': None, 'variable_info': 'The
last column of \'spambase.data\' denotes whether the e-mail was considered spam
(1) or not (0), i.e. unsolicited commercial e-mail.  Most of the attributes
indicate whether a particular word or character was frequently occuring in the
e-mail.  The run-length attributes (55-57) measure the length of sequences of
consecutive capital letters.  For the statistical measures of each attribute,
see the end of this file.  Here are the definitions of the attributes:\r\n\r\n48
continuous real [0,100] attributes of type word_freq_WORD \r\n= percentage of
words in the e-mail that match WORD, i.e. 100 * (number of times the WORD
appears in the e-mail) / total number of words in e-mail.  A "word" in this case
is any string of alphanumeric characters bounded by non-alphanumeric characters
or end-of-string.\r\n\r\n6 continuous real [0,100] attributes of type
char_freq_CHAR] \r\n= percentage of characters in the e-mail that match CHAR,
i.e. 100 * (number of CHAR occurences) / total characters in e-mail\r\n\r\n1
continuous real [1,…] attribute of type capital_run_length_average \r\n=
average length of uninterrupted sequences of capital letters\r\n\r\n1 continuous
integer [1,…] attribute of type capital_run_length_longest \r\n= length of
longest uninterrupted sequence of capital letters\r\n\r\n1 continuous integer
[1,…] attribute of type capital_run_length_total \r\n= sum of length of
uninterrupted sequences of capital letters \r\n= total number of capital letters
in the e-mail\r\n\r\n1 nominal {0,1} class attribute of type spam\r\n= denotes
whether the e-mail was considered spam (1) or not (0), i.e. unsolicited
commercial e-mail.  \r\n', 'citation': None}}
```


```
                          name     role        type demographic  \
0               word_freq_make  Feature  Continuous        None
1            word_freq_address  Feature  Continuous        None
2                word_freq_all  Feature  Continuous        None
3                 word_freq_3d  Feature  Continuous        None
4                word_freq_our  Feature  Continuous        None
5               word_freq_over  Feature  Continuous        None
6             word_freq_remove  Feature  Continuous        None
7           word_freq_internet  Feature  Continuous        None
8              word_freq_order  Feature  Continuous        None
9               word_freq_mail  Feature  Continuous        None
```

10	word_freq_receive	Feature	Continuous	None
11	word_freq_will	Feature	Continuous	None
12	word_freq_people	Feature	Continuous	None
13	word_freq_report	Feature	Continuous	None
14	word_freq_addresses	Feature	Continuous	None
15	word_freq_free	Feature	Continuous	None
16	word_freq_business	Feature	Continuous	None
17	word_freq_email	Feature	Continuous	None
18	word_freq_you	Feature	Continuous	None
19	word_freq_credit	Feature	Continuous	None
20	word_freq_your	Feature	Continuous	None
21	word_freq_font	Feature	Continuous	None
22	word_freq_000	Feature	Continuous	None
23	word_freq_money	Feature	Continuous	None
24	word_freq_hp	Feature	Continuous	None
25	word_freq_hpl	Feature	Continuous	None
26	word_freq_george	Feature	Continuous	None
27	word_freq_650	Feature	Continuous	None
28	word_freq_lab	Feature	Continuous	None
29	word_freq_labs	Feature	Continuous	None
30	word_freq_telnet	Feature	Continuous	None
31	word_freq_857	Feature	Continuous	None
32	word_freq_data	Feature	Continuous	None
33	word_freq_415	Feature	Continuous	None
34	word_freq_85	Feature	Continuous	None
35	word_freq_technology	Feature	Continuous	None
36	word_freq_1999	Feature	Continuous	None
37	word_freq_parts	Feature	Continuous	None
38	word_freq_pm	Feature	Continuous	None
39	word_freq_direct	Feature	Continuous	None
40	word_freq_cs	Feature	Continuous	None
41	word_freq_meeting	Feature	Continuous	None
42	word_freq_original	Feature	Continuous	None
43	word_freq_project	Feature	Continuous	None
44	word_freq_re	Feature	Continuous	None
45	word_freq_edu	Feature	Continuous	None
46	word_freq_table	Feature	Continuous	None
47	word_freq_conference	Feature	Continuous	None
48	char_freq_;	Feature	Continuous	None
49	char_freq_(	Feature	Continuous	None
50	char_freq_[	Feature	Continuous	None
51	char_freq_!	Feature	Continuous	None
52	char_freq_$	Feature	Continuous	None
53	char_freq_#	Feature	Continuous	None
54	capital_run_length_average	Feature	Continuous	None
55	capital_run_length_longest	Feature	Continuous	None
56	capital_run_length_total	Feature	Continuous	None
57	Class	Target	Binary	None

	description	units	missing_values
0	None	None	no
1	None	None	no
2	None	None	no
3	None	None	no
4	None	None	no
5	None	None	no
6	None	None	no
7	None	None	no
8	None	None	no
9	None	None	no
10	None	None	no
11	None	None	no
12	None	None	no
13	None	None	no
14	None	None	no
15	None	None	no
16	None	None	no
17	None	None	no
18	None	None	no
19	None	None	no
20	None	None	no
21	None	None	no
22	None	None	no
23	None	None	no
24	None	None	no
25	None	None	no
26	None	None	no
27	None	None	no
28	None	None	no
29	None	None	no
30	None	None	no
31	None	None	no
32	None	None	no
33	None	None	no
34	None	None	no
35	None	None	no
36	None	None	no
37	None	None	no
38	None	None	no
39	None	None	no
40	None	None	no
41	None	None	no
42	None	None	no
43	None	None	no
44	None	None	no
45	None	None	no

```
46                        None  None             no
47                        None  None             no
48                        None  None             no
49                        None  None             no
50                        None  None             no
51                        None  None             no
52                        None  None             no
53                        None  None             no
54                        None  None             no
55                        None  None             no
56                        None  None             no
57  spam (1) or not spam (0)  None             no
```

2.1 Creating the Dataframe

[ ]:
```
df = pd.DataFrame(X, columns=spambase.feature_names_short)
df['target'] = y
```

3 Getting Familiar with the Dataset

[ ]:
```
X
```

[ ]:
```
      word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
0               0.00               0.64           0.64           0.0
1               0.21               0.28           0.50           0.0
2               0.06               0.00           0.71           0.0
3               0.00               0.00           0.00           0.0
4               0.00               0.00           0.00           0.0
…                …                  …              …             …
4596            0.31               0.00           0.62           0.0
4597            0.00               0.00           0.00           0.0
4598            0.30               0.00           0.30           0.0
4599            0.96               0.00           0.00           0.0
4600            0.00               0.00           0.65           0.0

      word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
0              0.32            0.00              0.00                0.00
1              0.14            0.28              0.21                0.07
2              1.23            0.19              0.19                0.12
3              0.63            0.00              0.31                0.63
4              0.63            0.00              0.31                0.63
…               …               …                 …                   …
4596           0.00            0.31              0.00                0.00
4597           0.00            0.00              0.00                0.00
4598           0.00            0.00              0.00                0.00
4599           0.32            0.00              0.00                0.00
4600           0.00            0.00              0.00                0.00
```

```
      word_freq_order  word_freq_mail  …  word_freq_conference  char_freq_;  \
0                0.00            0.00  …                   0.0        0.000
1                0.00            0.94  …                   0.0        0.000
2                0.64            0.25  …                   0.0        0.010
3                0.31            0.63  …                   0.0        0.000
4                0.31            0.63  …                   0.0        0.000
…                 …               …  …                    …          …
4596             0.00            0.00  …                   0.0        0.000
4597             0.00            0.00  …                   0.0        0.000
4598             0.00            0.00  …                   0.0        0.102
4599             0.00            0.00  …                   0.0        0.000
4600             0.00            0.00  …                   0.0        0.000

      char_freq_(  char_freq_[  char_freq_!  char_freq_$  char_freq_#  \
0           0.000          0.0        0.778        0.000        0.000
1           0.132          0.0        0.372        0.180        0.048
2           0.143          0.0        0.276        0.184        0.010
3           0.137          0.0        0.137        0.000        0.000
4           0.135          0.0        0.135        0.000        0.000
…             …          …          …          …          …
4596        0.232          0.0        0.000        0.000        0.000
4597        0.000          0.0        0.353        0.000        0.000
4598        0.718          0.0        0.000        0.000        0.000
4599        0.057          0.0        0.000        0.000        0.000
4600        0.000          0.0        0.125        0.000        0.000

      capital_run_length_average  capital_run_length_longest  \
0                          3.756                          61
1                          5.114                         101
2                          9.821                         485
3                          3.537                          40
4                          3.537                          40
…                          …                         …
4596                       1.142                           3
4597                       1.555                           4
4598                       1.404                           6
4599                       1.147                           5
4600                       1.250                           5

      capital_run_length_total
0                          278
1                         1028
2                         2259
3                          191
4                          191
…                        …
```

```
4596                       88
4597                       14
4598                      118
4599                       78
4600                       40

[4601 rows x 57 columns]
```

[ ]:
```
X.columns
```

[ ]:
```
Index(['word_freq_make', 'word_freq_address', 'word_freq_all', 'word_freq_3d',
       'word_freq_our', 'word_freq_over', 'word_freq_remove',
       'word_freq_internet', 'word_freq_order', 'word_freq_mail',
       'word_freq_receive', 'word_freq_will', 'word_freq_people',
       'word_freq_report', 'word_freq_addresses', 'word_freq_free',
       'word_freq_business', 'word_freq_email', 'word_freq_you',
       'word_freq_credit', 'word_freq_your', 'word_freq_font', 'word_freq_000',
       'word_freq_money', 'word_freq_hp', 'word_freq_hpl', 'word_freq_george',
       'word_freq_650', 'word_freq_lab', 'word_freq_labs', 'word_freq_telnet',
       'word_freq_857', 'word_freq_data', 'word_freq_415', 'word_freq_85',
       'word_freq_technology', 'word_freq_1999', 'word_freq_parts',
       'word_freq_pm', 'word_freq_direct', 'word_freq_cs', 'word_freq_meeting',
       'word_freq_original', 'word_freq_project', 'word_freq_re',
       'word_freq_edu', 'word_freq_table', 'word_freq_conference',
       'char_freq_;', 'char_freq_(', 'char_freq_[', 'char_freq_!',
       'char_freq_$', 'char_freq_#', 'capital_run_length_average',
       'capital_run_length_longest', 'capital_run_length_total'],
      dtype='object')
```

[ ]:
```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 57 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   word_freq_make              4601 non-null   float64
 1   word_freq_address           4601 non-null   float64
 2   word_freq_all               4601 non-null   float64
 3   word_freq_3d                4601 non-null   float64
 4   word_freq_our               4601 non-null   float64
 5   word_freq_over              4601 non-null   float64
 6   word_freq_remove            4601 non-null   float64
 7   word_freq_internet          4601 non-null   float64
 8   word_freq_order             4601 non-null   float64
 9   word_freq_mail              4601 non-null   float64
 10  word_freq_receive           4601 non-null   float64
 11  word_freq_will              4601 non-null   float64
```

```
 12  word_freq_people            4601 non-null   float64
 13  word_freq_report            4601 non-null   float64
 14  word_freq_addresses         4601 non-null   float64
 15  word_freq_free              4601 non-null   float64
 16  word_freq_business          4601 non-null   float64
 17  word_freq_email             4601 non-null   float64
 18  word_freq_you               4601 non-null   float64
 19  word_freq_credit            4601 non-null   float64
 20  word_freq_your              4601 non-null   float64
 21  word_freq_font              4601 non-null   float64
 22  word_freq_000               4601 non-null   float64
 23  word_freq_money             4601 non-null   float64
 24  word_freq_hp                4601 non-null   float64
 25  word_freq_hpl               4601 non-null   float64
 26  word_freq_george            4601 non-null   float64
 27  word_freq_650               4601 non-null   float64
 28  word_freq_lab               4601 non-null   float64
 29  word_freq_labs              4601 non-null   float64
 30  word_freq_telnet            4601 non-null   float64
 31  word_freq_857               4601 non-null   float64
 32  word_freq_data              4601 non-null   float64
 33  word_freq_415               4601 non-null   float64
 34  word_freq_85                4601 non-null   float64
 35  word_freq_technology        4601 non-null   float64
 36  word_freq_1999              4601 non-null   float64
 37  word_freq_parts             4601 non-null   float64
 38  word_freq_pm                4601 non-null   float64
 39  word_freq_direct            4601 non-null   float64
 40  word_freq_cs                4601 non-null   float64
 41  word_freq_meeting           4601 non-null   float64
 42  word_freq_original          4601 non-null   float64
 43  word_freq_project           4601 non-null   float64
 44  word_freq_re                4601 non-null   float64
 45  word_freq_edu               4601 non-null   float64
 46  word_freq_table             4601 non-null   float64
 47  word_freq_conference        4601 non-null   float64
 48  char_freq_;                 4601 non-null   float64
 49  char_freq_(                 4601 non-null   float64
 50  char_freq_[                 4601 non-null   float64
 51  char_freq_!                 4601 non-null   float64
 52  char_freq_$                 4601 non-null   float64
 53  char_freq_#                 4601 non-null   float64
 54  capital_run_length_average  4601 non-null   float64
 55  capital_run_length_longest  4601 non-null   int64
 56  capital_run_length_total    4601 non-null   int64
dtypes: float64(55), int64(2)
memory usage: 2.0 MB
```

```
[ ]: X.describe()
```

```
[ ]:        word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
count     4601.000000        4601.000000    4601.000000   4601.000000
mean         0.104553           0.213015       0.280656      0.065425
std          0.305358           1.290575       0.504143      1.395151
min          0.000000           0.000000       0.000000      0.000000
25%          0.000000           0.000000       0.000000      0.000000
50%          0.000000           0.000000       0.000000      0.000000
75%          0.000000           0.000000       0.420000      0.000000
max          4.540000          14.280000       5.100000     42.810000

       word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
count    4601.000000     4601.000000       4601.000000         4601.000000
mean        0.312223        0.095901          0.114208            0.105295
std         0.672513        0.273824          0.391441            0.401071
min         0.000000        0.000000          0.000000            0.000000
25%         0.000000        0.000000          0.000000            0.000000
50%         0.000000        0.000000          0.000000            0.000000
75%         0.380000        0.000000          0.000000            0.000000
max        10.000000        5.880000          7.270000           11.110000

       word_freq_order  word_freq_mail  …  word_freq_conference  \
count      4601.000000     4601.000000  …           4601.000000
mean          0.090067        0.239413  …              0.031869
std           0.278616        0.644755  …              0.285735
min           0.000000        0.000000  …              0.000000
25%           0.000000        0.000000  …              0.000000
50%           0.000000        0.000000  …              0.000000
75%           0.000000        0.160000  …              0.000000
max           5.260000       18.180000  …             10.000000

       char_freq_;  char_freq_(  char_freq_[  char_freq_!  char_freq_$  \
count  4601.000000  4601.000000  4601.000000  4601.000000  4601.000000
mean      0.038575     0.139030     0.016976     0.269071     0.075811
std       0.243471     0.270355     0.109394     0.815672     0.245882
min       0.000000     0.000000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.000000
50%       0.000000     0.065000     0.000000     0.000000     0.000000
75%       0.000000     0.188000     0.000000     0.315000     0.052000
max       4.385000     9.752000     4.081000    32.478000     6.003000

       char_freq_#  capital_run_length_average  capital_run_length_longest  \
count  4601.000000                 4601.000000                 4601.000000
mean      0.044238                    5.191515                   52.172789
std       0.429342                   31.729449                  194.891310
min       0.000000                    1.000000                    1.000000
```

```
25%      0.000000                   1.588000                    6.000000
50%      0.000000                   2.276000                   15.000000
75%      0.000000                   3.706000                   43.000000
max     19.829000                1102.500000                 9989.000000

       capital_run_length_total
count               4601.000000
mean                 283.289285
std                  606.347851
min                    1.000000
25%                   35.000000
50%                   95.000000
75%                  266.000000
max                15841.000000

[8 rows x 57 columns]
```

[ ]:
```
y
```

[ ]:
```
      Class
0         1
1         1
2         1
3         1
4         1
…       …
4596      0
4597      0
4598      0
4599      0
4600      0

[4601 rows x 1 columns]
```

[ ]:
```
y.describe()
```

[ ]:
```
             Class
count  4601.000000
mean      0.394045
std       0.488698
min       0.000000
25%       0.000000
50%       0.000000
75%       1.000000
max       1.000000
```

[ ]:
```
y.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4601 entries, 0 to 4600
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Class   4601 non-null   int64
dtypes: int64(1)
memory usage: 36.1 KB
```

3.1 Question 1

1.1 How many features are there in the dataset?

As you have observed, the variable X, which represents the feature vector, consists of 57 columns. This indicates that the dataset contains 57 features

1.2 Could you provide a brief explanation of each feature?

First of all, please note that all features in this dataset are numerical. Based on the names of each column and their data types, the columns that contain the keyword "freq" represent the frequency of a word or character in the text. However, there are three columns that do not have the "freq" keyword:

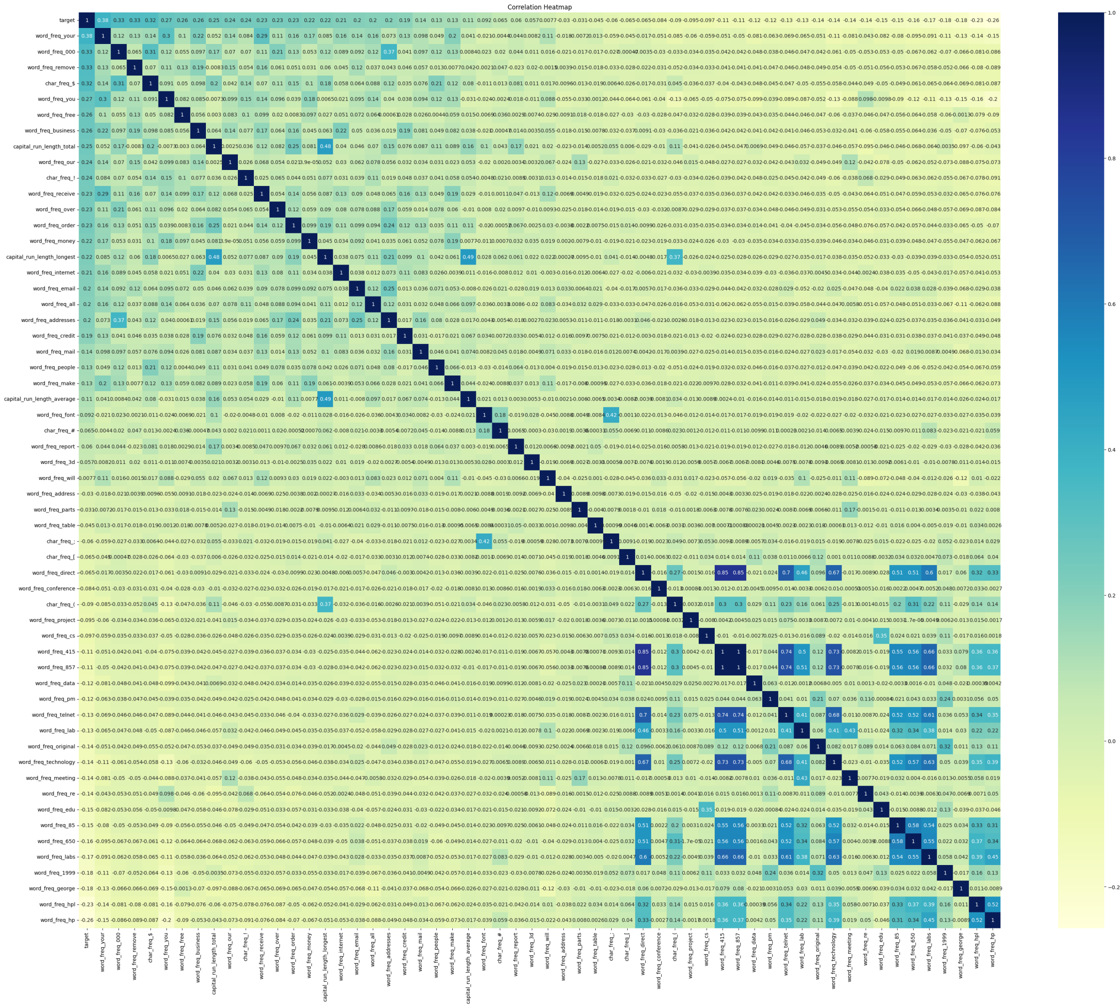
1. capital_run_length_average: This column indicates the average length of phrases with capital letters in the text of the email.
2. capital_run_length_longest: This column indicates the longest consecutive sequence of capital letters in the text of the email.
3. capital_run_length_total: This column indicates the overall frequency of capital letters in the text of the email.

1.3 What is the relationship between the features and the target variable?

To answer this question, let's see the correlation heatmap between input features and the target variable:

[ ]:
```
corr_with_target = df.corr()['target'].sort_values(ascending=False)
```

[ ]:
```
plt.figure(figsize=(40, 32))
sns.heatmap(df[corr_with_target.index].corr(), cmap="YlGnBu", annot=True)
plt.title('Correlation Heatmap')
plt.show()
```

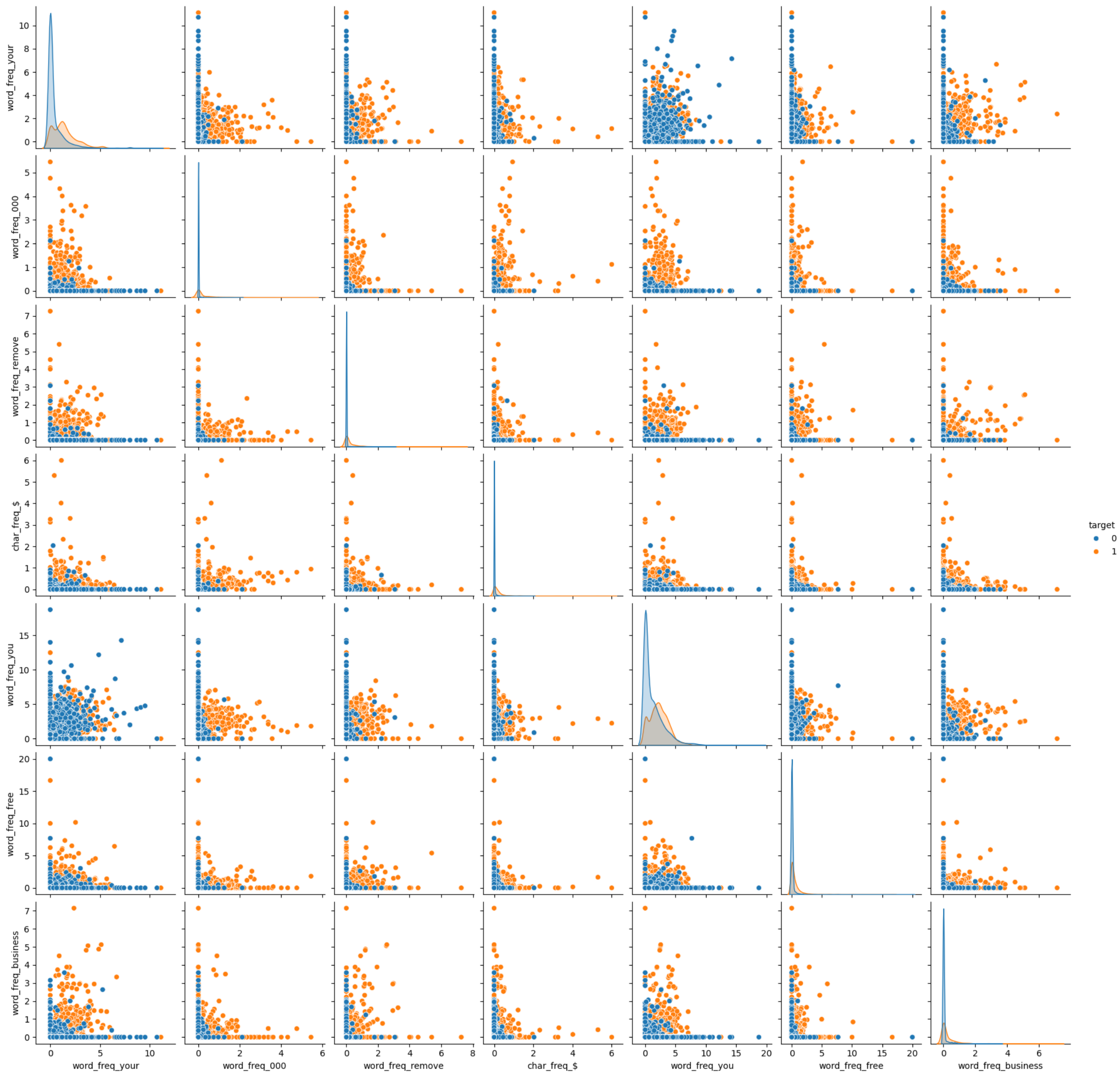


Furthermore, let's plot a pairplot of the 7 input features that have the highest correlation with the target variable. This will allow us to visualize their relationship with the target variable in a more effective manner.

```
[ ]: top_corr_features = corr_with_target.index[1:8]
     top_corr_features
```

```
[ ]: Index(['word_freq_your', 'word_freq_000', 'word_freq_remove', 'char_freq_$',
            'word_freq_you', 'word_freq_free', 'word_freq_business'],
           dtype='object')
```

```
[ ]: sns.pairplot(df, vars=top_corr_features, hue='target')
     plt.show()
```



1.4 Are all of the features informative and useful in predicting the target variable?

Not all features in a dataset are always informative or useful for predicting the target variable. Some features might be redundant or irrelevant, and can be removed without incurring much loss of information. Reducing the number of features can help improve the performance of a machine learning model by alleviating issues such as overfitting and high computational cost. We will perform preprocessing steps to identify the most suitable features and ensure they are properly cleaned for the models that will be trained on the dataset.

4 Preprocessing

4.1 Features Importance

One way to find importance of each feature is to calculate the mutual information of each feature with the target variable that can be done using "mutual_info_classif" method in python. This method computes the information gain of each feature using the following formula:

$H(X) - H(X|Y)$

That H(X) is entropy of X

[ ]:
```python
discrete_features = [False for _ in list(X.columns)]

index = [col for col in list(X.columns)]

feature_importance = pd.DataFrame(columns = ['fi'], index = index)

feature_importance['fi'] = mutual_info_classif(X=X, y=y,␣
↪discrete_features=discrete_features, random_state=1401)
feature_importance = feature_importance.sort_values(by='fi', ascending=False)

feature_importance
```

[ ]:

	fi
char_freq_!	0.207199
capital_run_length_longest	0.186514
char_freq_$	0.179647
capital_run_length_average	0.172395
word_freq_your	0.160483
word_freq_remove	0.155613
capital_run_length_total	0.138657
word_freq_free	0.134731
word_freq_money	0.116666
word_freq_hp	0.116089
word_freq_000	0.110283
word_freq_you	0.104641
word_freq_our	0.099015
word_freq_george	0.087874
word_freq_business	0.085198
word_freq_all	0.080496
word_freq_hpl	0.078179
word_freq_receive	0.076259
word_freq_mail	0.071755
word_freq_address	0.068259
word_freq_over	0.063330
word_freq_internet	0.059936
word_freq_email	0.057100
word_freq_credit	0.057071
word_freq_will	0.053527
word_freq_order	0.053378
char_freq_(	0.051365
word_freq_edu	0.050793
word_freq_re	0.048533

```
word_freq_addresses         0.045612
char_freq_#                 0.043857
word_freq_lab               0.041429
word_freq_people            0.041222
word_freq_make              0.040439
word_freq_85                0.037944
word_freq_labs              0.037589
word_freq_1999              0.035242
word_freq_telnet            0.031501
word_freq_meeting           0.029336
word_freq_original          0.028302
word_freq_650               0.028238
word_freq_technology        0.028061
word_freq_pm                0.027902
word_freq_857               0.021855
word_freq_cs                0.021353
word_freq_conference        0.021194
word_freq_data              0.021141
word_freq_415               0.020587
word_freq_project           0.019880
char_freq_;                 0.015641
word_freq_report            0.014027
word_freq_font              0.010290
word_freq_parts             0.009443
word_freq_direct            0.008153
word_freq_3d                0.007104
word_freq_table             0.001882
char_freq_[                 0.000000
```

4.2 Checking for Missing Values

[ ]:
```
X.isna().any()
```

[ ]:
```
word_freq_make                False
word_freq_address             False
word_freq_all                 False
word_freq_3d                  False
word_freq_our                 False
word_freq_over                False
word_freq_remove              False
word_freq_internet            False
word_freq_order               False
word_freq_mail                False
word_freq_receive             False
word_freq_will                False
word_freq_people              False
word_freq_report              False
```

```
word_freq_addresses          False
word_freq_free               False
word_freq_business           False
word_freq_email              False
word_freq_you                False
word_freq_credit             False
word_freq_your               False
word_freq_font               False
word_freq_000                False
word_freq_money              False
word_freq_hp                 False
word_freq_hpl                False
word_freq_george             False
word_freq_650                False
word_freq_lab                False
word_freq_labs               False
word_freq_telnet             False
word_freq_857                False
word_freq_data               False
word_freq_415                False
word_freq_85                 False
word_freq_technology         False
word_freq_1999               False
word_freq_parts              False
word_freq_pm                 False
word_freq_direct             False
word_freq_cs                 False
word_freq_meeting            False
word_freq_original           False
word_freq_project            False
word_freq_re                 False
word_freq_edu                False
word_freq_table              False
word_freq_conference         False
char_freq_;                  False
char_freq_(                  False
char_freq_[                  False
char_freq_!                  False
char_freq_$                  False
char_freq_#                  False
capital_run_length_average   False
capital_run_length_longest   False
capital_run_length_total     False
dtype: bool
```

As, you can observe, there's no missing value in this dataset.

4.3 Checking for Outliers

```
[ ]: def detect_outliers_iqr(df, col):
         Q1 = df[col].quantile(0.25)
         Q3 = df[col].quantile(0.75)
         IQR = Q3 - Q1
         upper_limit = Q3 + 1.5 * IQR
         lower_limit = Q1 - 1.5 * IQR
         outliers = df[(df[col] < lower_limit) | (df[col] > upper_limit)]
         return outliers
```

```
[ ]: outliers_count = {}
```

```
[ ]: for col in X.columns:
         outliers_count[col] = len(detect_outliers_iqr(X, col))
```

```
[ ]: sorted_outliers_count = {k: v for k, v in sorted(outliers_count.items(),␣
      ↪key=lambda item: item[1])}
```

Now, you can observe the columns with their correspond outliers count:

```
[ ]: for col, count in sorted_outliers_count.items():
         print(f"{col}: {count} outliers")
```

```
word_freq_3d: 47 outliers
word_freq_table: 63 outliers
word_freq_you: 75 outliers
word_freq_parts: 83 outliers
word_freq_font: 117 outliers
word_freq_cs: 148 outliers
word_freq_conference: 203 outliers
word_freq_857: 205 outliers
word_freq_415: 215 outliers
word_freq_your: 229 outliers
word_freq_will: 270 outliers
word_freq_telnet: 293 outliers
char_freq_(: 296 outliers
word_freq_project: 327 outliers
word_freq_addresses: 336 outliers
word_freq_all: 338 outliers
word_freq_meeting: 341 outliers
word_freq_report: 357 outliers
capital_run_length_average: 363 outliers
word_freq_lab: 372 outliers
word_freq_original: 375 outliers
word_freq_pm: 384 outliers
word_freq_data: 405 outliers
char_freq_!: 411 outliers
word_freq_credit: 424 outliers
```

```
word_freq_direct: 453 outliers
word_freq_650: 463 outliers
capital_run_length_longest: 463 outliers
word_freq_labs: 469 outliers
word_freq_85: 485 outliers
word_freq_our: 501 outliers
word_freq_edu: 517 outliers
char_freq_[: 529 outliers
capital_run_length_total: 550 outliers
word_freq_technology: 599 outliers
word_freq_000: 679 outliers
word_freq_receive: 709 outliers
word_freq_money: 735 outliers
char_freq_#: 750 outliers
word_freq_order: 773 outliers
word_freq_george: 780 outliers
char_freq_;: 790 outliers
word_freq_remove: 807 outliers
word_freq_hpl: 811 outliers
char_freq_$: 811 outliers
word_freq_internet: 824 outliers
word_freq_1999: 829 outliers
word_freq_mail: 852 outliers
word_freq_people: 852 outliers
word_freq_address: 898 outliers
word_freq_free: 957 outliers
word_freq_business: 963 outliers
word_freq_over: 999 outliers
word_freq_re: 1001 outliers
word_freq_email: 1038 outliers
word_freq_make: 1053 outliers
word_freq_hp: 1090 outliers
```

Now, we will proceed to select the top 10 best features for further steps. The criterion for selecting these best features is based on their importance. Additionally, we will address any outliers present in the chosen best features.

[ ]:
```
top_12_features = feature_importance.head(12).index
best_X = X[top_12_features]
```

[ ]:
```
best_X
```

[ ]:
```
   char_freq_!  capital_run_length_longest  char_freq_$  \
0        0.778                          61        0.000
1        0.372                         101        0.180
2        0.276                         485        0.184
3        0.137                          40        0.000
4        0.135                          40        0.000
```

```
…          …                        …          …
4596       0.000                          3          0.000
4597       0.353                          4          0.000
4598       0.000                          6          0.000
4599       0.000                          5          0.000
4600       0.125                          5          0.000

      capital_run_length_average  word_freq_your  word_freq_remove  \
0                          3.756            0.96              0.00
1                          5.114            1.59              0.21
2                          9.821            0.51              0.19
3                          3.537            0.31              0.31
4                          3.537            0.31              0.31
…                          …             …                …
4596                       1.142            0.00              0.00
4597                       1.555            2.00              0.00
4598                       1.404            0.30              0.00
4599                       1.147            0.32              0.00
4600                       1.250            0.65              0.00

      capital_run_length_total  word_freq_free  word_freq_money  word_freq_hp  \
0                          278            0.32             0.00           0.0
1                         1028            0.14             0.43           0.0
2                         2259            0.06             0.06           0.0
3                          191            0.31             0.00           0.0
4                          191            0.31             0.00           0.0
…                        …             …                …           …
4596                        88            0.00             0.00           0.0
4597                        14            0.00             0.00           0.0
4598                       118            0.00             0.00           0.0
4599                        78            0.00             0.00           0.0
4600                        40            0.00             0.00           0.0

      word_freq_000  word_freq_you
0              0.00           1.93
1              0.43           3.47
2              1.16           1.36
3              0.00           3.18
4              0.00           3.18
…             …            …
4596           0.00           0.62
4597           0.00           6.00
4598           0.00           1.50
4599           0.00           1.93
4600           0.00           4.60

[4601 rows x 12 columns]
```

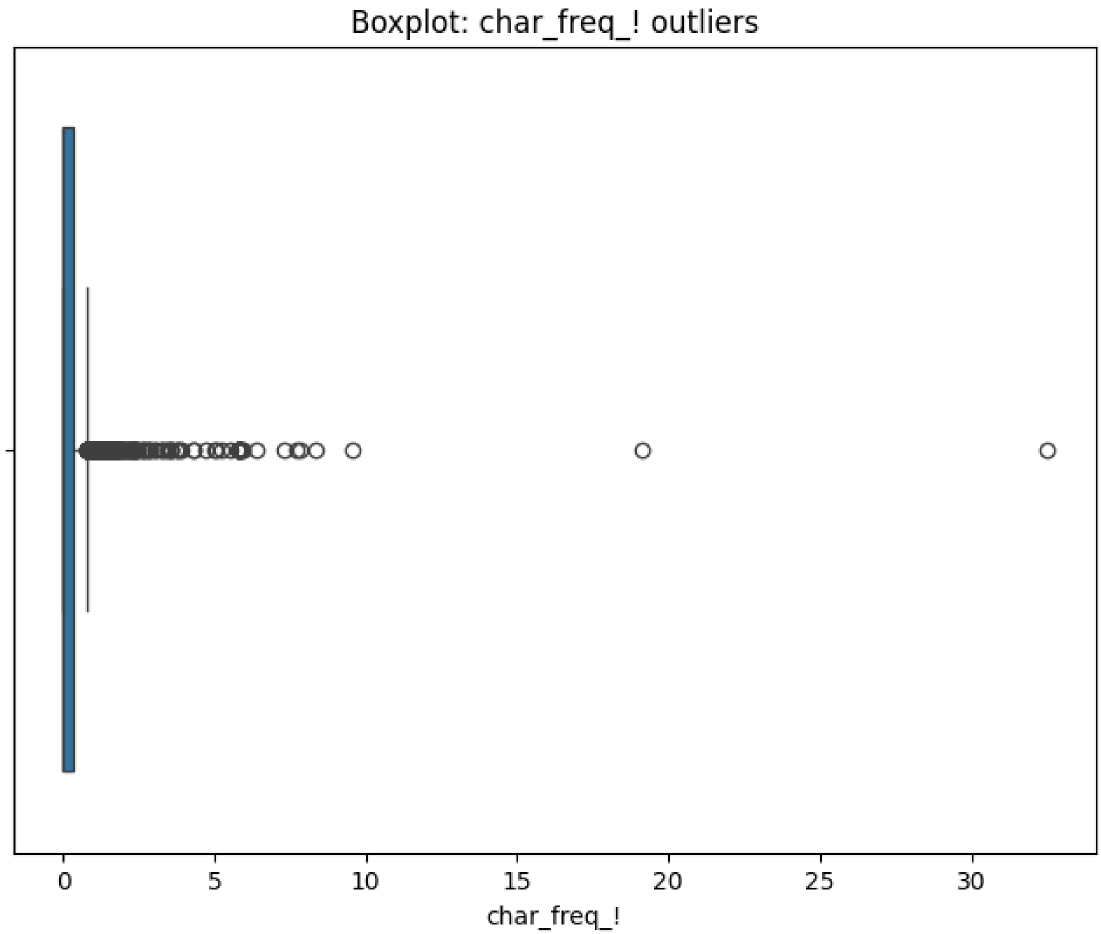
Let's see the outliers count in our best features:

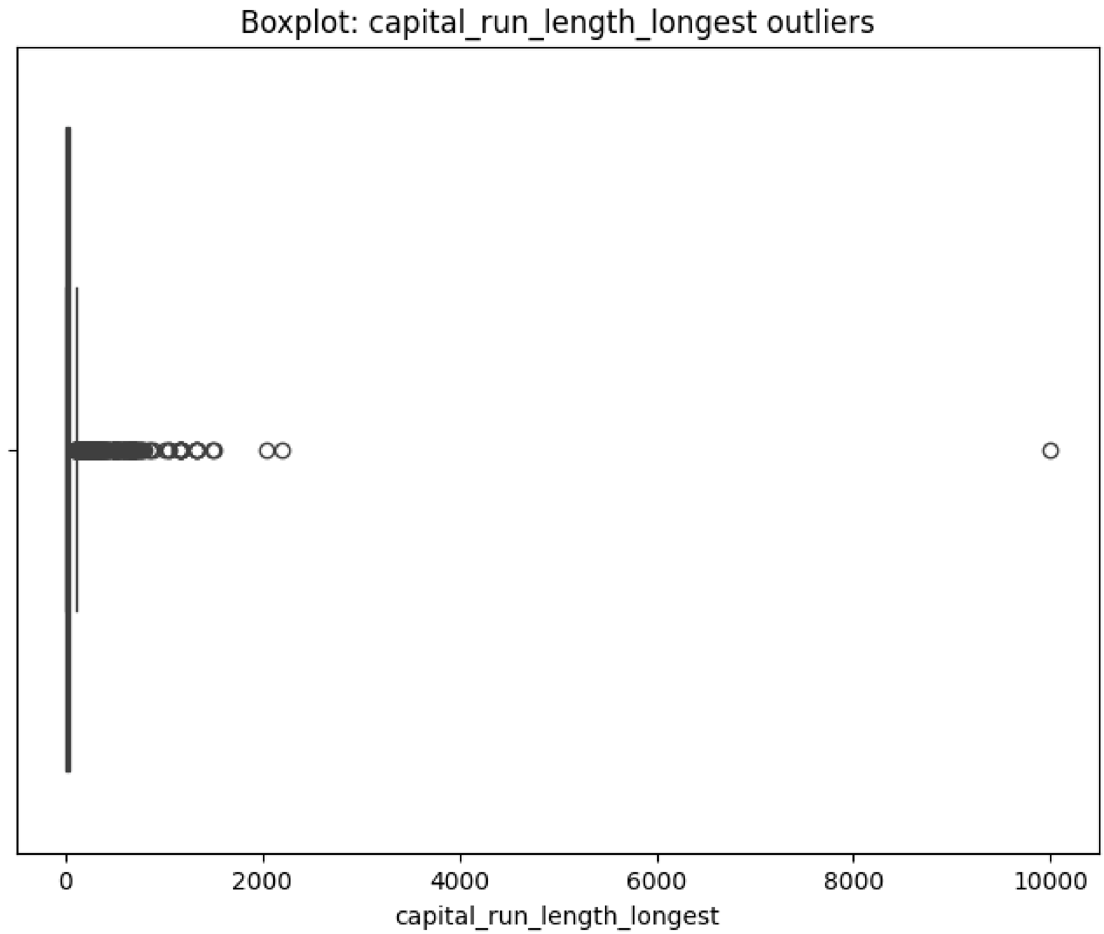
```
[ ]: for col in best_X.columns:
         print(f'{col} outliers count: {outliers_count[col]}')
```

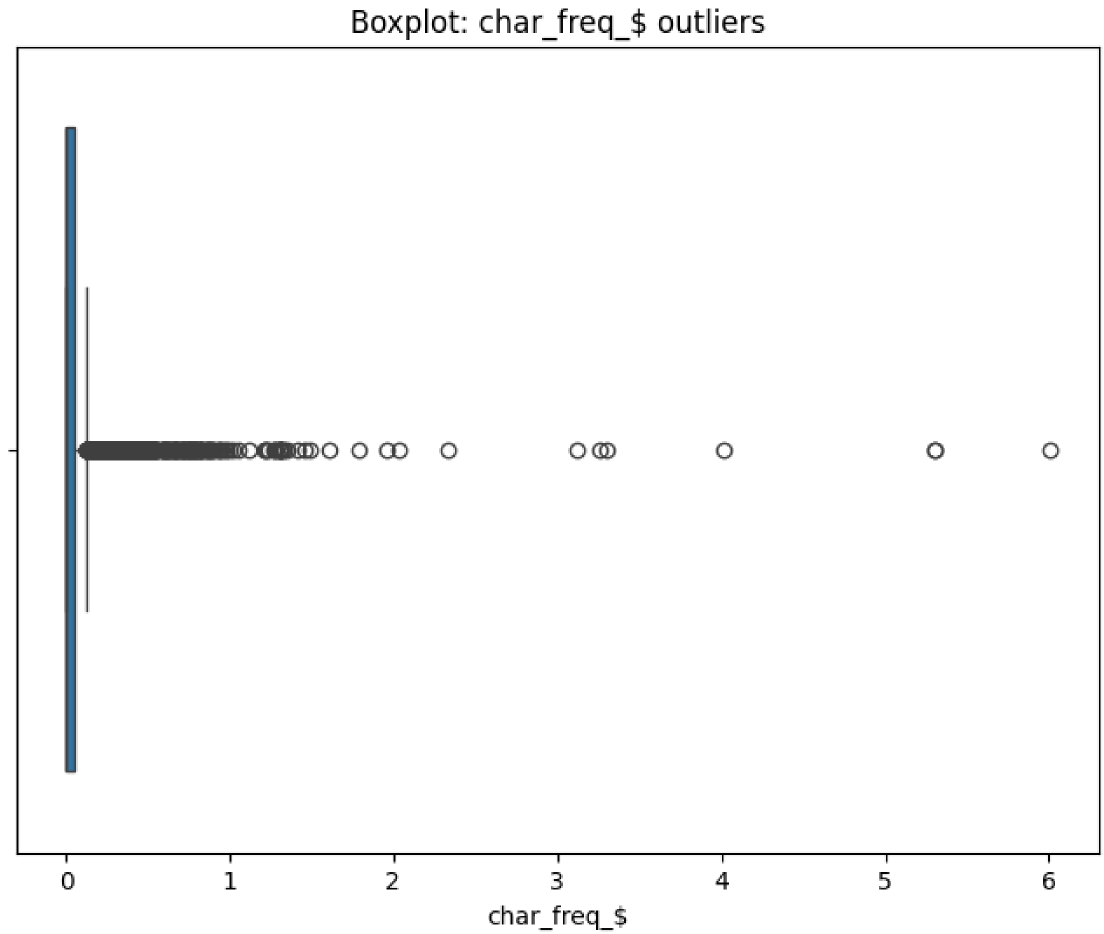
```
char_freq_! outliers count: 411
capital_run_length_longest outliers count: 463
char_freq_$ outliers count: 811
capital_run_length_average outliers count: 363
word_freq_your outliers count: 229
word_freq_remove outliers count: 807
capital_run_length_total outliers count: 550
word_freq_free outliers count: 957
word_freq_money outliers count: 735
word_freq_hp outliers count: 1090
word_freq_000 outliers count: 679
word_freq_you outliers count: 75
```

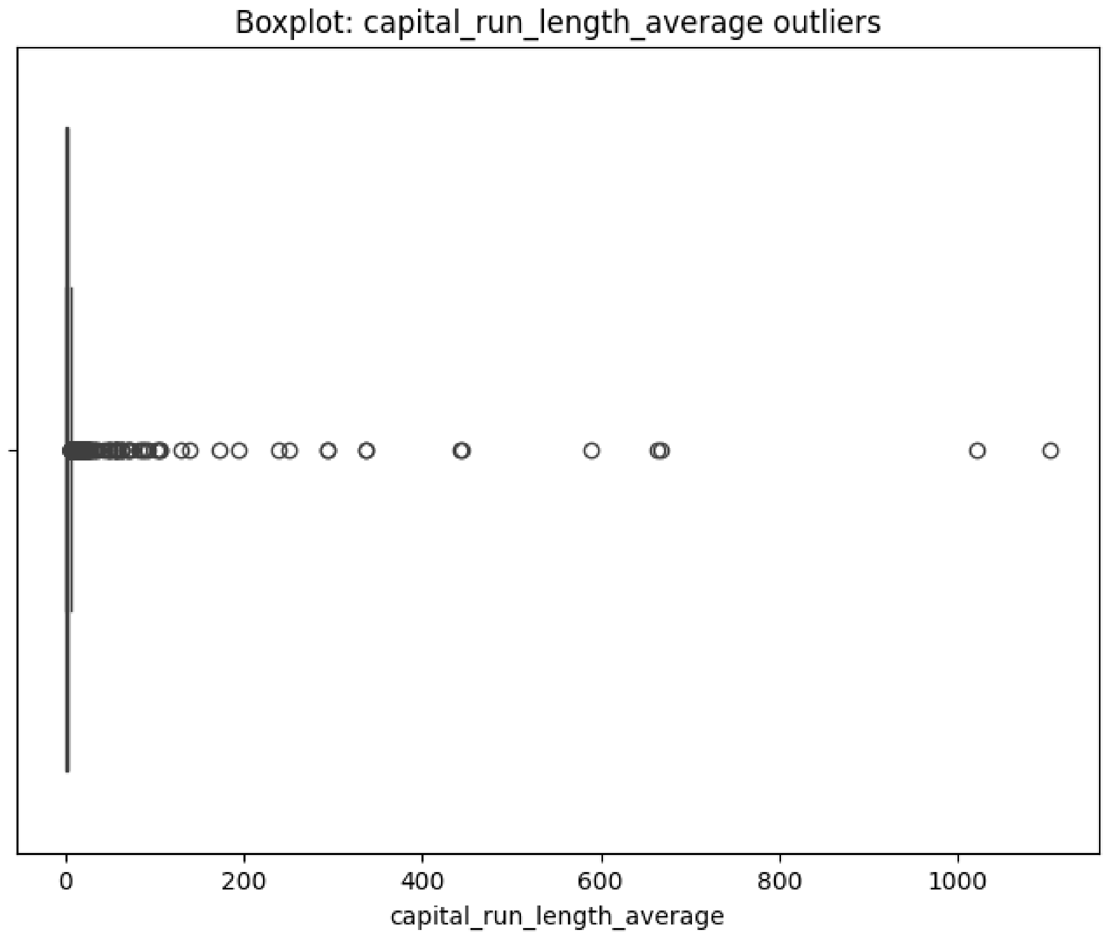
Now, let's create box plots for each column to enhance the visualization of outliers.

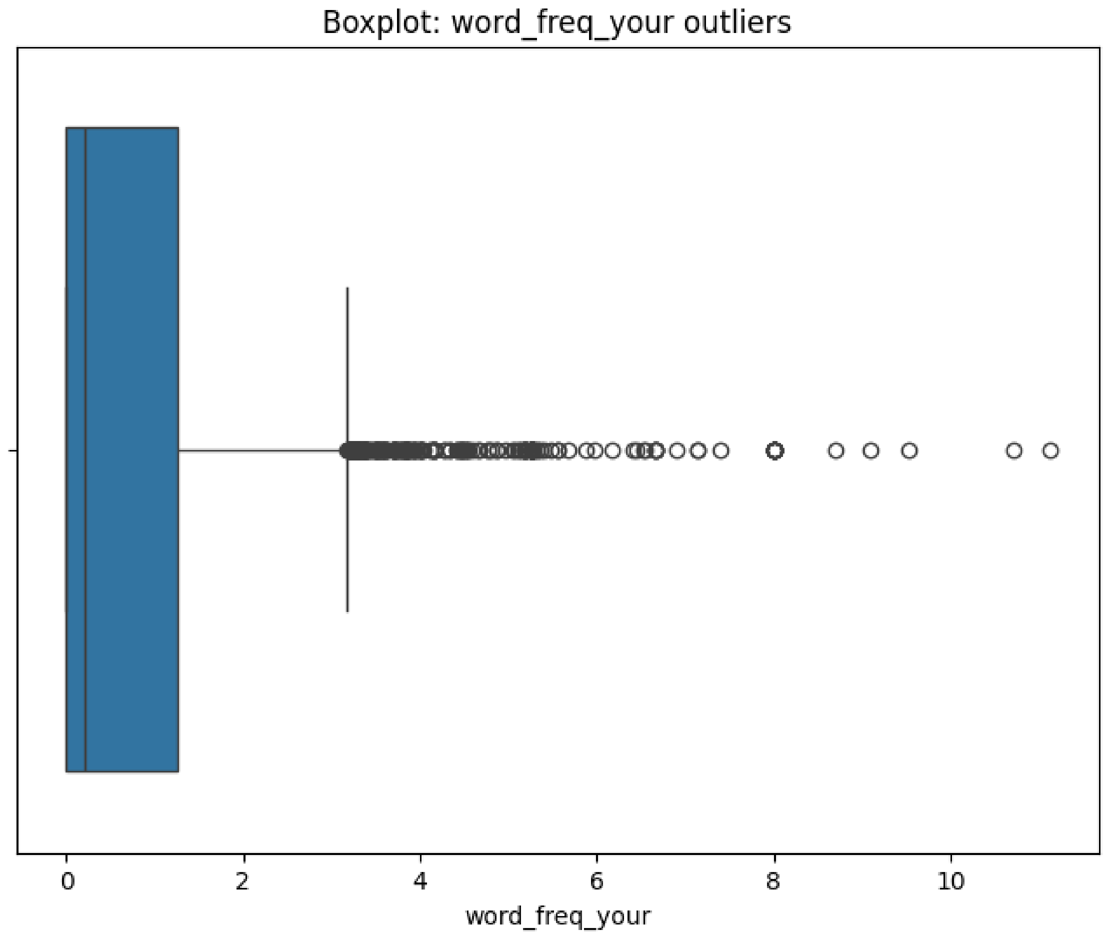
```
[ ]: for col in best_X.columns:

         plt.figure(figsize=(8, 6))
         sns.boxplot(x=best_X[col])
         plt.title(f'Boxplot: {col} outliers')
         plt.show()

         # Check for negative values in the column outliers
         negative_outliers = outliers_count[col] < 0
         if negative_outliers:
             print(f'Negative outliers exist in column: {col}')
```

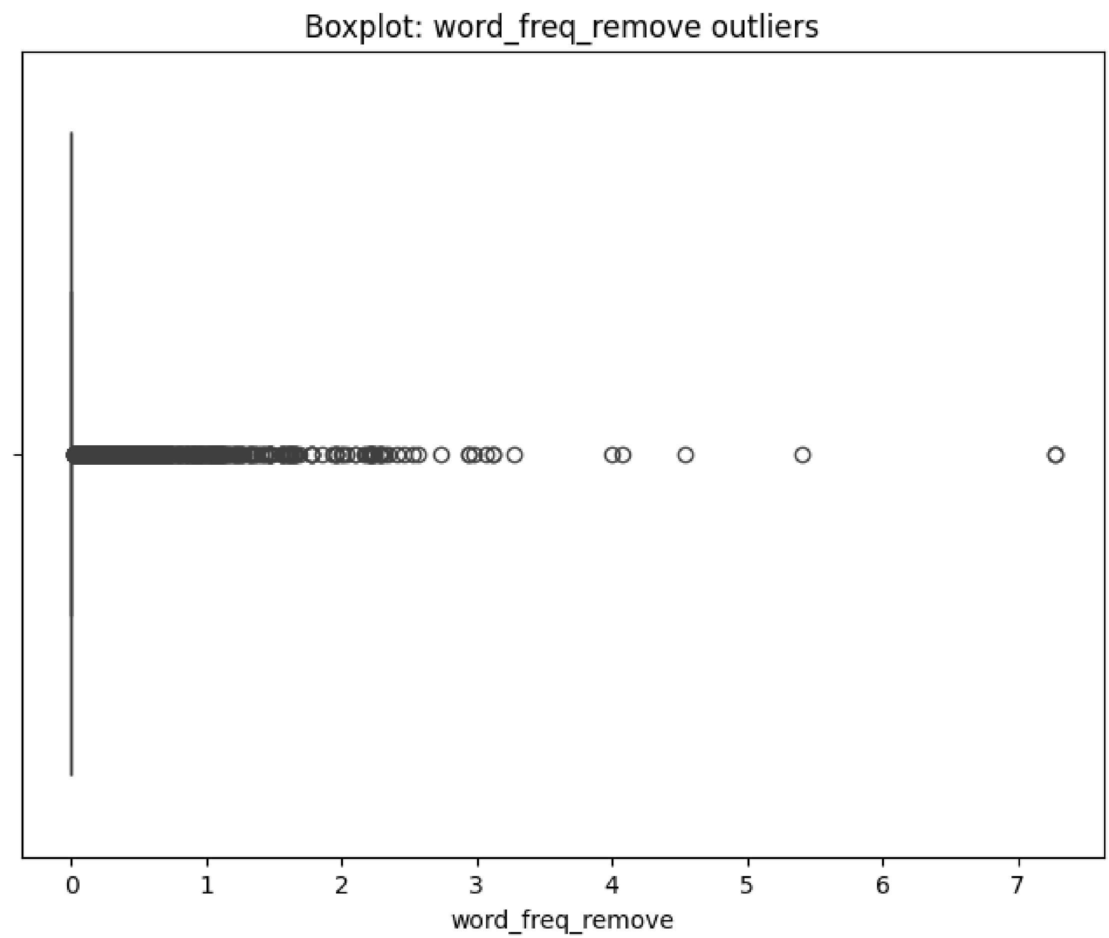

Boxplot: char_freq_! outliers
0
5
10
15
20
25
30
char_freq_!

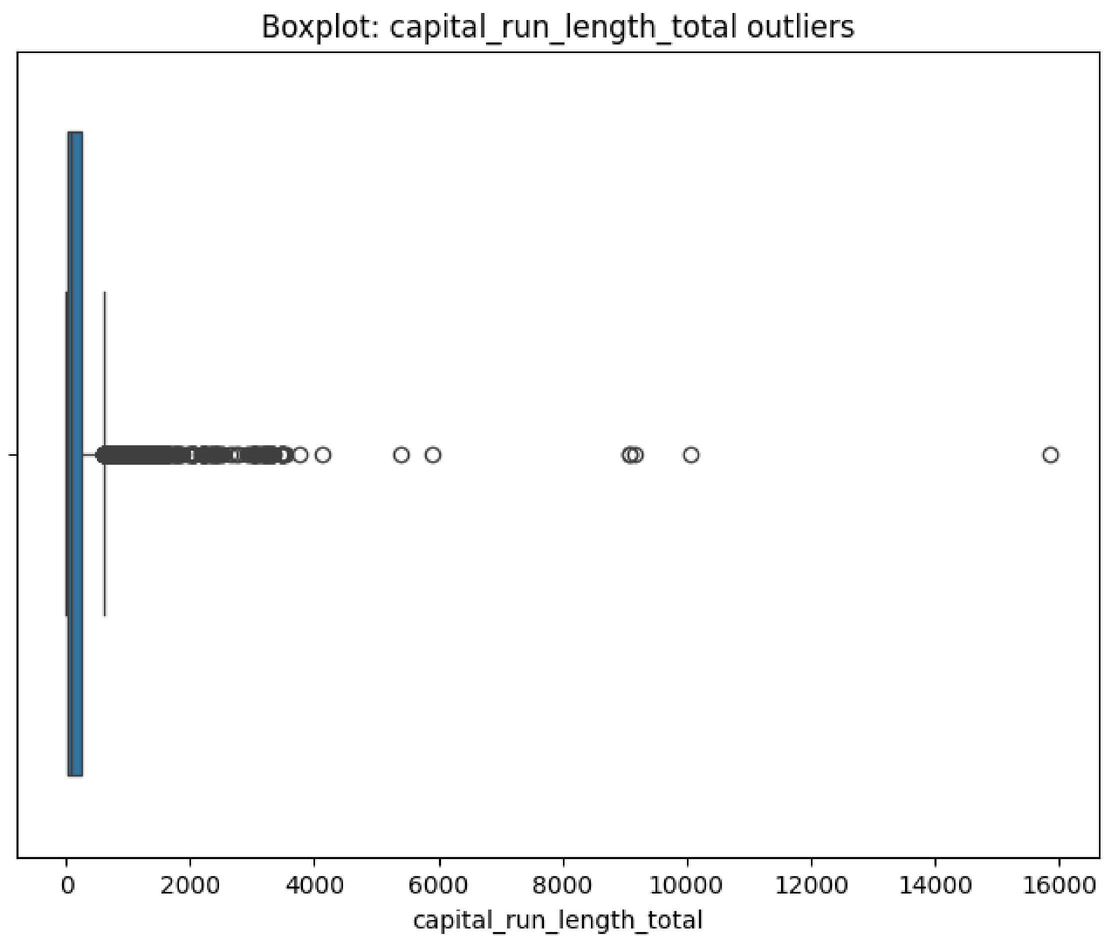

Boxplot: capital_run_length_longest outliers
0
2000
4000
6000
8000
10000
capital_run_length_longest

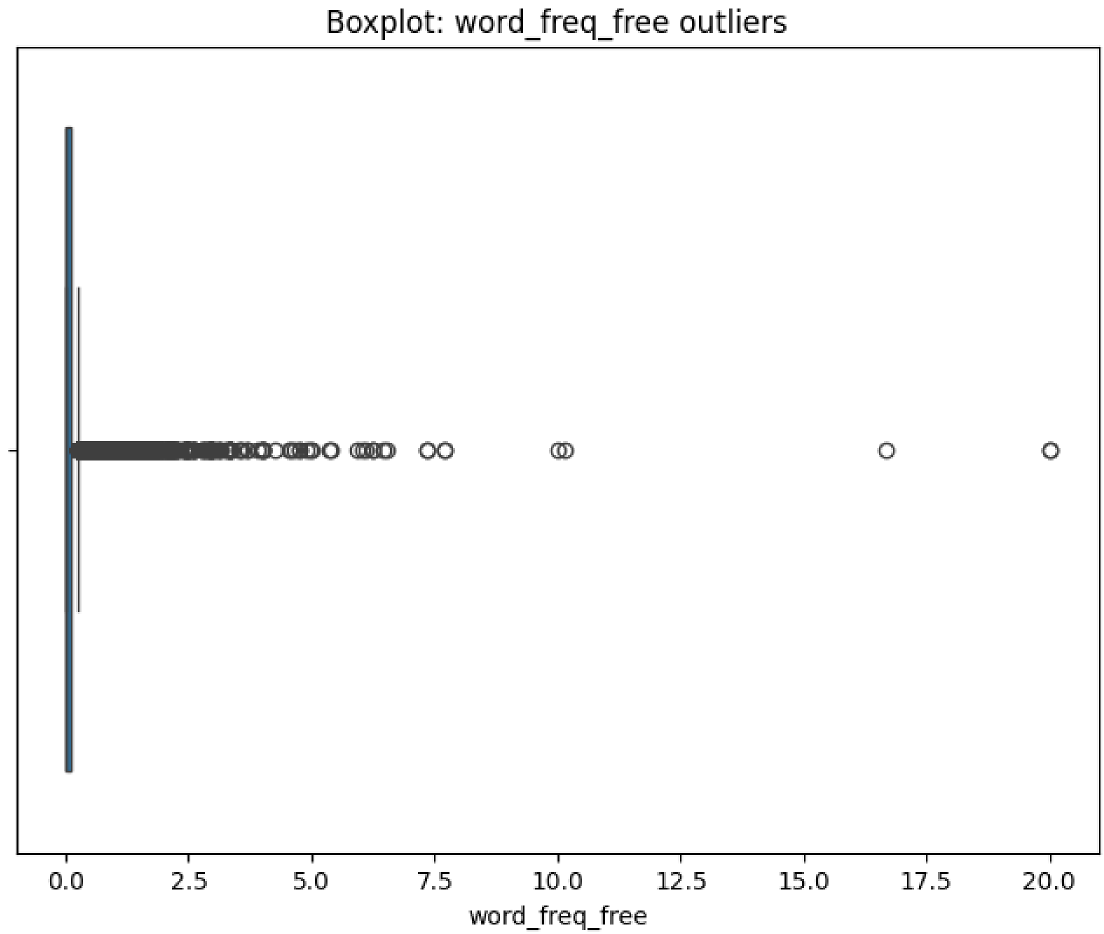

Boxplot: char_freq_$ outliers
0
1
2
3
4
5
6
char_freq_$

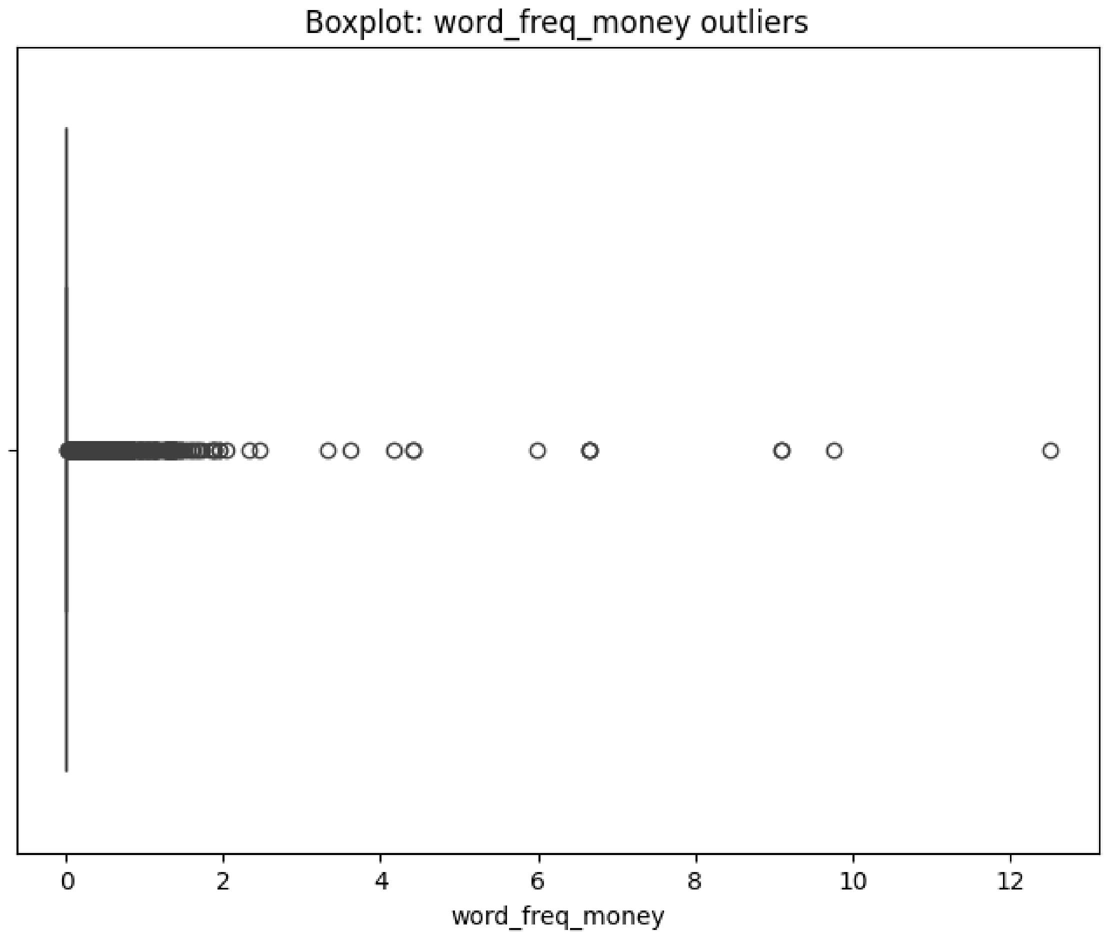

Boxplot: capital_run_length_average outliers
0
200
400
600
800
1000
capital_run_length_average

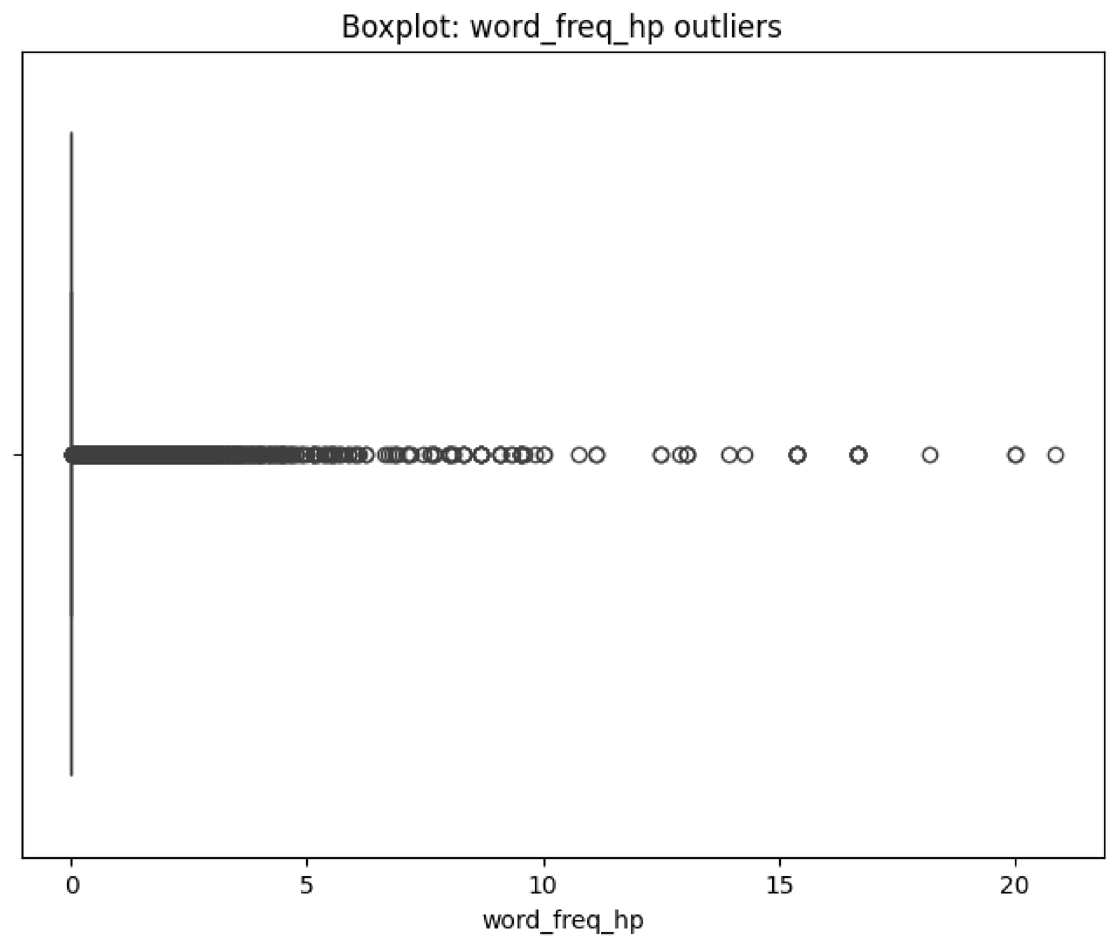

Boxplot: word_freq_your outliers
0
2
4
6
8
10
word_freq_your

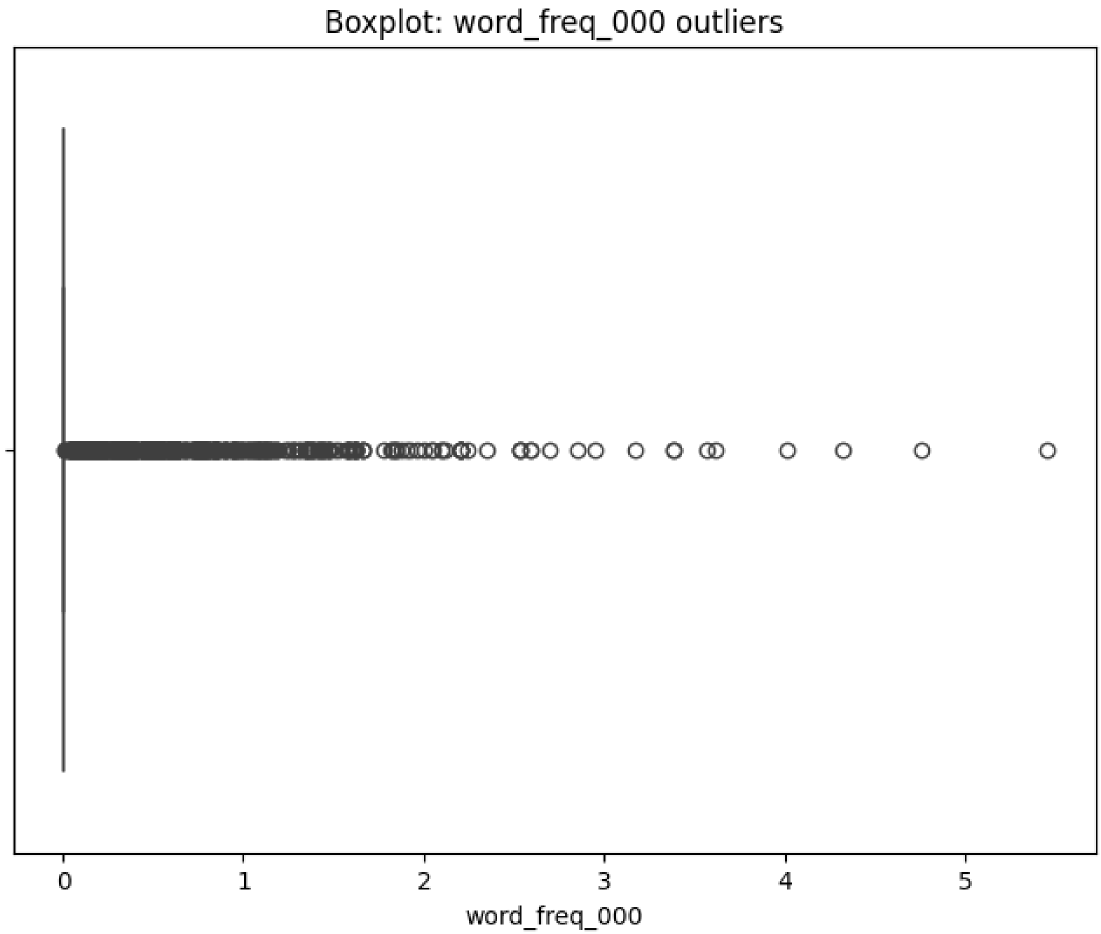

Boxplot: word_freq_remove outliers
0
1
2
3
4
5
6
7
word_freq_remove

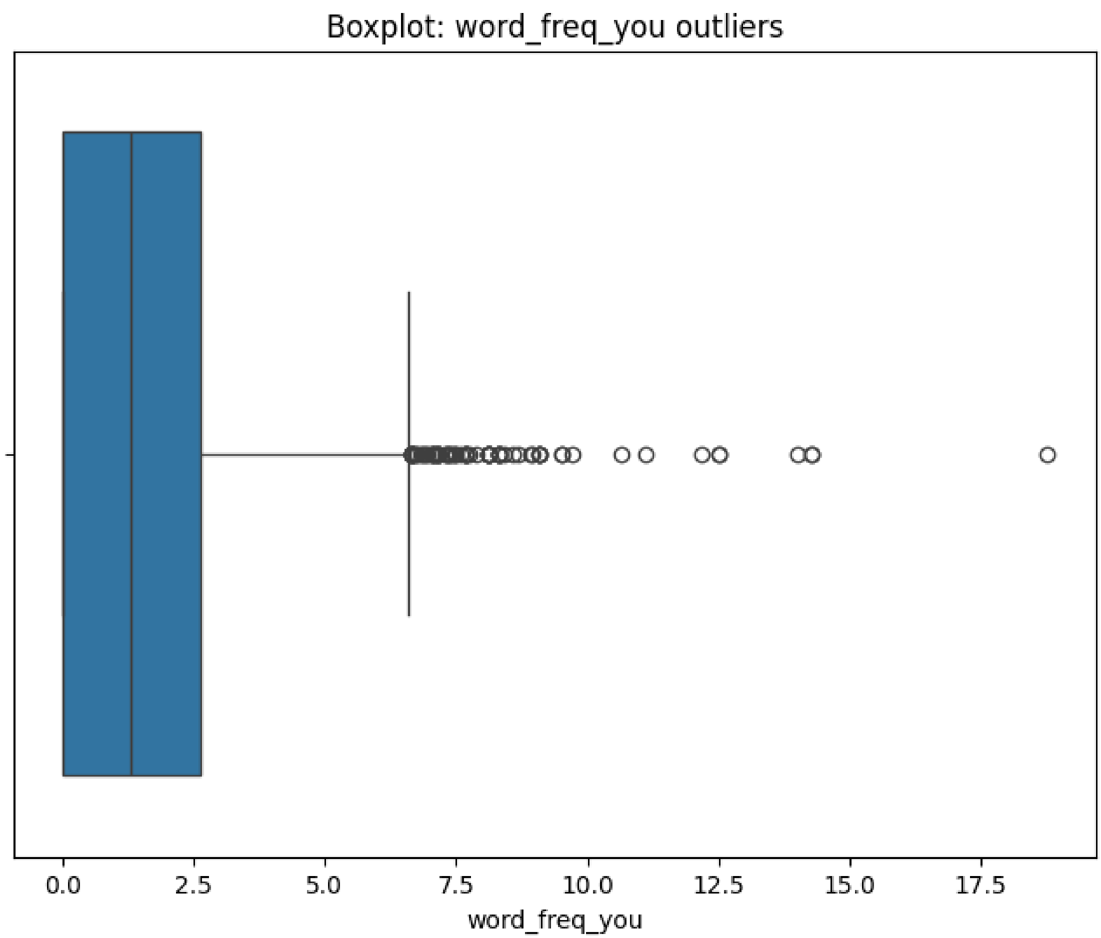

Boxplot: capital_run_length_total outliers
0
2000
4000
6000
8000
10000
12000
14000
16000
capital_run_length_total


Boxplot: word_freq_free outliers
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
word_freq_free


Boxplot: word_freq_money outliers
0
2
4
6
8
10
12
word_freq_money


Boxplot: word_freq_hp outliers
0
5
10
15
20
word_freq_hp


Boxplot: word_freq_000 outliers
0
1
2
3
4
5
word_freq_000



Good news! The outliers in the dataset are not of a casual nature, and they do not have negative values. Although they have a significant impact on our models, they will not result in erroneous predictions. Therefore, there is no need to remove them.

5 Question 2

2.1 Split the dataset into train and test sets and reserve 30 percent of the data for the test set.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
       ↪random_state=42)
```

2.2 Train multiple ML models including Logistic Regression, KNN, Naive Bayes, Decision Tree, Adaboost, Random Forest, linear, and non-linear SVM. Furthermore, tune the hyperparameters for each model. Moreover, report the classification results and the time consumed for training each model

5.1 Training Models

5.1.1 Utils

```
[ ]: def tune_hyperparameters(model, param_grid, X_train, y_train):
         grid_search = GridSearchCV(estimator=model, param_grid=param_grid,␣
      ↪scoring='accuracy', cv=5)
         grid_search.fit(X_train, y_train)
         best_params = grid_search.best_params_

         return best_params

     def train_model(model, X_train, y_train, best_params):
         if best_params:
             model.set_params(**best_params)

         start_time = time.time()
         model.fit(X_train, y_train)
         end_time = time.time()
         training_time = end_time - start_time

         return model, training_time

     def report_results(model_name, best_params, model, X_train, y_train,␣
      ↪training_time):
         train_predictions = model.predict(X_train)
         train_report = classification_report(y_train, train_predictions)

         print(f"Model Name: {model_name}")

         if best_params:
             print(f"\nBest Hyperparameters: {best_params}")

         print(f"\nThe time consumed for training: {training_time} seconds")
         print("\nClassification Report for Train Data:")
         print(train_report)
```

5.1.2 Logistic Regression

```
[ ]: logistic_model = LogisticRegression()
     logistic_param_grid = {'C': [0.1, 1, 10],
                            'penalty': ['l1', 'l2']}

     logestic_best_params = tune_hyperparameters(logistic_model,␣
      ↪logistic_param_grid, X_train, y_train)
     trained_model, training_time = train_model(logistic_model, X_train, y_train,␣
      ↪logestic_best_params)
```

[ ]:
```
report_results("Logistic Regression", logestic_best_params, trained_model,␣
↪X_train, y_train, training_time)
```

```
Model Name: Logistic Regression

Best Hyperparameters: {'C': 10, 'penalty': 'l2'}

The time consumed for training: 0.03457331657409668 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.92      0.94      0.93      1984
           1       0.90      0.87      0.89      1236

    accuracy                           0.91      3220
   macro avg       0.91      0.91      0.91      3220
weighted avg       0.91      0.91      0.91      3220
```

5.1.3 KNN

[ ]:
```
knn_model = KNeighborsClassifier()
knn_param_grid = {'n_neighbors': [3, 5, 7],
                  'weights': ['uniform', 'distance']}

knn_best_params = tune_hyperparameters(knn_model, knn_param_grid, X_train,␣
↪y_train)
knn_trained_model, knn_training_time = train_model(knn_model, X_train, y_train,␣
↪knn_best_params)
```

[ ]:
```
report_results("KNN", knn_best_params, knn_trained_model, X_train, y_train,␣
↪knn_training_time)
```

```
Model Name: KNN

Best Hyperparameters: {'n_neighbors': 5, 'weights': 'distance'}

The time consumed for training: 0.0021038055419921875 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1984
           1       1.00      1.00      1.00      1236

    accuracy                           1.00      3220
   macro avg       1.00      1.00      1.00      3220
```

```
weighted avg       1.00      1.00      1.00      3220
```

5.1.4 Naive Bayes

Note that since Naive Bayes models do not require hyperparameter tuning, there is no need for the tune_hyperparameters method

[ ]:
```
naive_bayes_model = GaussianNB()

naive_bayes_trained_model, naive_bayes_training_time =␣
 ↪train_model(naive_bayes_model, X_train, y_train, [])
```

[ ]:
```
report_results("Naive Bayes", [], naive_bayes_trained_model, X_train, y_train,␣
 ↪naive_bayes_training_time)
```

```
Model Name: Naive Bayes

The time consumed for training: 0.010539531707763672 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.96      0.73      0.83      1984
           1       0.68      0.95      0.80      1236

    accuracy                           0.81      3220
   macro avg       0.82      0.84      0.81      3220
weighted avg       0.86      0.81      0.82      3220
```

5.1.5 Decision Tree

[ ]:
```
decision_tree_model = DecisionTreeClassifier()
decision_tree_param_grid = {'max_depth': [None, 5, 10, 15],
                            'min_samples_split': [2, 5, 10]}

decision_tree_best_params = tune_hyperparameters(decision_tree_model,␣
 ↪decision_tree_param_grid, X_train, y_train)
trained_decision_tree_model, decision_tree_training_time =␣
 ↪train_model(decision_tree_model, X_train, y_train, decision_tree_best_params)
```

[ ]:
```
report_results("Decision Tree", decision_tree_best_params,␣
 ↪trained_decision_tree_model, X_train, y_train, decision_tree_training_time)
```

```
Model Name: Decision Tree

Best Hyperparameters: {'max_depth': 15, 'min_samples_split': 2}
```

```
The time consumed for training: 0.045362234115600586 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      1984
           1       0.99      0.97      0.98      1236

    accuracy                           0.99      3220
   macro avg       0.99      0.98      0.98      3220
weighted avg       0.99      0.99      0.99      3220
```

5.1.6 Adaboost

[ ]:
```python
adaboost_model = AdaBoostClassifier()
adaboost_param_grid = {'n_estimators': [50, 100, 200],
                       'learning_rate': [0.1, 0.5, 1.0]}

adaboost_best_params = tune_hyperparameters(adaboost_model,␣
 ↪adaboost_param_grid, X_train, y_train)
trained_adaboost_model, adaboost_training_time = train_model(adaboost_model,␣
 ↪X_train, y_train, adaboost_best_params)
```

[ ]:
```python
report_results("Adaboost", adaboost_best_params, trained_adaboost_model,␣
 ↪X_train, y_train, adaboost_training_time)
```

```
Model Name: Adaboost

Best Hyperparameters: {'learning_rate': 0.5, 'n_estimators': 200}

The time consumed for training: 1.380237340927124 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1984
           1       0.96      0.95      0.95      1236

    accuracy                           0.96      3220
   macro avg       0.96      0.96      0.96      3220
weighted avg       0.96      0.96      0.96      3220
```

5.1.7 Random Forest

[ ]:
```
random_forest_model = RandomForestClassifier()
random_forest_param_grid = {'n_estimators': [100, 200, 500],
                            'max_depth': [None, 5, 10, 15]}

random_forest_best_params = tune_hyperparameters(random_forest_model,␣
 ↪random_forest_param_grid, X_train, y_train)
trained_random_forest_model, random_forest_training_time =␣
 ↪train_model(random_forest_model, X_train, y_train, random_forest_best_params)
```

[ ]:
```
report_results("Random Forest", random_forest_best_params,␣
 ↪trained_random_forest_model, X_train, y_train, random_forest_training_time)
```

```
Model Name: Random Forest

Best Hyperparameters: {'max_depth': None, 'n_estimators': 500}

The time consumed for training: 2.8785240650177 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1984
           1       1.00      1.00      1.00      1236

    accuracy                           1.00      3220
   macro avg       1.00      1.00      1.00      3220
weighted avg       1.00      1.00      1.00      3220
```

5.1.8 Linear SVM

[ ]:
```
linear_svm_model = LinearSVC()
linear_svm_param_grid = {'C': [0.1, 1, 10]}

linear_svm_best_params = tune_hyperparameters(linear_svm_model,␣
 ↪linear_svm_param_grid, X_train, y_train)
trained_linear_svm_model, linear_svm_training_time =␣
 ↪train_model(linear_svm_model, X_train, y_train, linear_svm_best_params)
```

[ ]:
```
report_results("Linear SVM", linear_svm_best_params, trained_linear_svm_model,␣
 ↪X_train, y_train, linear_svm_training_time)
```

```
Model Name: Linear SVM

Best Hyperparameters: {'C': 1}

The time consumed for training: 0.04999566078186035 seconds
```

```
Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.93      0.96      0.94      1984
           1       0.93      0.88      0.90      1236

    accuracy                           0.93      3220
   macro avg       0.93      0.92      0.92      3220
weighted avg       0.93      0.93      0.93      3220
```

5.1.9 Non-linear SVM

[ ]:
```
nonlinear_svm_model = SVC()
nonlinear_svm_param_grid = {'C': [0.1, 1, 10],
                            'kernel': ['rbf', 'poly', 'sigmoid']}

nonlinear_svm_best_params = tune_hyperparameters(nonlinear_svm_model,␣
 ↪nonlinear_svm_param_grid, X_train, y_train)
trained_nonlinear_svm_model, nonlinear_svm_training_time =␣
 ↪train_model(nonlinear_svm_model, X_train, y_train, nonlinear_svm_best_params)
```

[ ]:
```
report_results("Non-linear SVM", nonlinear_svm_best_params,␣
 ↪trained_nonlinear_svm_model, X_train, y_train, nonlinear_svm_training_time)
```

```
Model Name: Non-linear SVM

Best Hyperparameters: {'C': 10, 'kernel': 'rbf'}

The time consumed for training: 0.3791682720184326 seconds

Classification Report for Train Data:
              precision    recall  f1-score   support

           0       0.73      0.90      0.81      1984
           1       0.76      0.47      0.58      1236

    accuracy                           0.74      3220
   macro avg       0.75      0.69      0.70      3220
weighted avg       0.74      0.74      0.72      3220
```

2.3 What are training errors? Explain 2 of them at least.

During the training of a machine learning model, training errors refer to the discrepancies or mistakes made by the model during the learning process. Here are explanations of two common types of training errors:

1. Bias Error: Bias error, also known as underfitting, occurs when a model is too simplistic to

capture the underlying patterns or complexity of the data. It often leads to high errors on both the training and testing data. Bias error can arise when the model is too simple or lacks the necessary features to accurately represent the data. It can result in a model that is unable to learn important relationships and produces inaccurate predictions.

2. Variance Error: Variance error, also known as overfitting, occurs when a model becomes too complex or too closely fits the training data, leading to poor generalization on unseen data. While a model with low bias aims to minimize training errors, it can suffer from high variance if the model becomes too sensitive to noise or random fluctuations in the training data. Overfitting typically results in low errors on the training data but performs poorly on new, unseen data.

6 Question 3

3.1 Apply the trained models on test dataset and report the classification results as well as the time consumed for test

6.1 Testing the Models

6.1.1 Utils

[ ]:
```
def report_test_results(model_name, model, X_test, y_test):
    start_time = time.time()
    test_predictions = model.predict(X_test)
    end_time = time.time()

    testing_time = end_time - start_time

    test_report = classification_report(y_test, test_predictions)

    print(f"Model Name: {model_name}")
    print(f"\nThe time consumed for testing: {testing_time} seconds")
    print("\nClassification Report for Test Data:")
    print(test_report)
```

6.1.2 Logistic Regression

[ ]:
```
report_test_results("Logistic Regression", logistic_model, X_test, y_test)
```

```
Model Name: Logistic Regression

The time consumed for testing: 0.0015742778778076172 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.92      0.94      0.93       804
           1       0.91      0.89      0.90       577
```

```
    accuracy                           0.92      1381
   macro avg       0.91      0.91      0.91      1381
weighted avg       0.92      0.92      0.92      1381
```

6.1.3 KNN

[ ]:
```
report_test_results("KNN", knn_model, X_test, y_test)
```

```
Model Name: KNN

The time consumed for testing: 0.01650404930114746 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.82      0.85      0.84       804
           1       0.78      0.74      0.76       577

    accuracy                           0.81      1381
   macro avg       0.80      0.80      0.80      1381
weighted avg       0.81      0.81      0.81      1381
```

6.1.4 Naive Bayes

[ ]:
```
report_test_results("Naive Bayes", naive_bayes_model, X_test, y_test)
```

```
Model Name: Naive Bayes

The time consumed for testing: 0.006520986557006836 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.95      0.74      0.83       804
           1       0.72      0.95      0.82       577

    accuracy                           0.82      1381
   macro avg       0.84      0.84      0.82      1381
weighted avg       0.86      0.82      0.83      1381
```

6.1.5 Decision Tree

[ ]:
```
report_test_results("Decision Tree", decision_tree_model, X_test, y_test)
```

```
Model Name: Decision Tree
```

```
The time consumed for testing: 0.0027549266815185547 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.92      0.93      0.93       804
           1       0.90      0.89      0.90       577

    accuracy                           0.91      1381
   macro avg       0.91      0.91      0.91      1381
weighted avg       0.91      0.91      0.91      1381
```

6.1.6 Adaboost

```
[ ]: report_test_results("Adaboost", adaboost_model, X_test, y_test)
```

```
Model Name: Adaboost

The time consumed for testing: 0.06086611747741699 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.95      0.96      0.96       804
           1       0.95      0.93      0.94       577

    accuracy                           0.95      1381
   macro avg       0.95      0.95      0.95      1381
weighted avg       0.95      0.95      0.95      1381
```

6.1.7 Random Forest

```
[ ]: report_test_results("Random Forest", random_forest_model, X_test, y_test)
```

```
Model Name: Random Forest

The time consumed for testing: 0.0995640754699707 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       804
           1       0.96      0.93      0.95       577

    accuracy                           0.96      1381
   macro avg       0.96      0.95      0.95      1381
```

```
weighted avg       0.96      0.96      0.96      1381
```

6.1.8 Linear SVM

```
[ ]: report_test_results("Linear SVM", linear_svm_model, X_test, y_test)
```

```
Model Name: Linear SVM

The time consumed for testing: 0.003495454788208008 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.92      0.96      0.94       804
           1       0.94      0.88      0.91       577

    accuracy                           0.92      1381
   macro avg       0.93      0.92      0.92      1381
weighted avg       0.93      0.92      0.92      1381
```

6.1.9 Non-linear SVM

```
[ ]: report_test_results("Non-linear SVM", nonlinear_svm_model, X_test, y_test)
```

```
Model Name: Non-linear SVM

The time consumed for testing: 0.2033226490020752 seconds

Classification Report for Test Data:
              precision    recall  f1-score   support

           0       0.71      0.91      0.79       804
           1       0.79      0.48      0.59       577

    accuracy                           0.73      1381
   macro avg       0.75      0.69      0.69      1381
weighted avg       0.74      0.73      0.71      1381
```

3.2 Which model had the best performance and why?

As you can observe the Random Forest model had the best performance while the Non-linear SVM and KNN models were the weakest. The performance of machine learning models can vary based on the characteristics of the dataset and the inherent properties of the models themselves. Here are some justifications for why the Random Forest model might have outperformed the non-linear SVM and KNN models in our scenario:

1. Handling Non-linear Relationships: Non-linear SVM models are known for their ability to

handle non-linear relationships. However, in cases where the decision boundaries are highly complex and have non-linear patterns, the Random Forest model can often provide better performance. Random Forests operate by creating an ensemble of decision trees and combining their predictions through voting or averaging. This ensemble approach can capture intricate non-linear relationships more effectively than individual non-linear SVM models.

2. Robustness Against Noisy Data: KNN models are sensitive to noise and outliers in the data. If our dataset contained a significant amount of noise or outliers, it could have negatively impacted the performance of the KNN model. Random Forests, on the other hand, are generally more robust to noisy data since each decision tree in the ensemble is trained on a subset of the data and is less likely to be affected by outliers.

3. Feature Importance and Dimensionality: Random Forests have the advantage of providing feature importance measures. This can help in identifying the most relevant features for the classification task. In contrast, non-linear SVM models do not inherently provide feature importance measures. Additionally, if the dataset has a high dimensionality, Random Forest models can handle a large number of features more efficiently compared to non-linear SVM models.

3.3 Can eliminating non-informative features improve the results?

Eliminating non-informative features of a dataset can potentially improve the results of training a machine learning model. Here's why:

1. Reduce Overfitting: Non-informative features do not contribute meaningful information to the model's learning process. Including them in the model can lead to overfitting, where the model becomes too complex and starts interpreting noise or irrelevant patterns in the data. By removing non-informative features, the model becomes more focused on the relevant patterns and reduces the chances of overfitting.

2. Improve Model Efficiency: Including non-informative features increases the dimensionality of the dataset. High-dimensional datasets can make the model more computationally expensive and increase the risk of the curse of dimensionality. By eliminating non-informative features, the dataset's dimensionality is reduced, leading to faster training times and potentially better model performance.

3. Enhance Interpretability: Non-informative features can introduce noise and make it more challenging to interpret the model's results. The presence of irrelevant features can obscure the true relationships between the input features and the target variable. By removing non-informative features, the model becomes more interpretable, enabling easier identification and understanding of the meaningful relationships.