Open in Colab

Installing the Libraries

In [73]:

```
!pip install pyspellchecker==0.5.6
!pip install fast-autocomplete[levenshtein]
```

```
Requirement already satisfied: pyspellchecker==0.5.6 in /usr/local/lib/python3.7/dist-pac
kages (0.5.6)
Requirement already satisfied: fast-autocomplete[levenshtein] in /usr/local/lib/python3.7
/dist-packages (0.7.1)
Requirement already satisfied: python-Levenshtein>=0.12.0; extra == "levenshtein" in /usr
/local/lib/python3.7/dist-packages (from fast-autocomplete[levenshtein]) (0.12.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-packages (from
python-Levenshtein>=0.12.0; extra == "levenshtein"->fast-autocomplete[levenshtein]) (54.1
.2)
```

Importing the Libraries

In [74]:

```
# Load libraries
import numpy as np
import pandas as pd
import random
import re
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,confusion_matrix,classification_report
from spellchecker import SpellChecker
from fast_autocomplete import AutoComplete
from keras.models import Sequential
from keras.layers import Dense
import tensorflow as tf
import seaborn as sns
import matplotlib.pyplot as plt
```

Reading CSV files

In [80]:

```
df_train = pd.read_csv('exercise_20_train.csv')
df_test = pd.read_csv('exercise_20_test.csv')
```

Data preprocessing

In [81]:

```
# Print Train data
df_train
```

Out[81]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
0	0.963686	6.627185	-45.224008	9.477531	-3.216532	13.216874	9.754747	5.245851	-1.102918	-2.867482	-37.632285

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
1	-1.770062	-23.610459	-0.964003	-31.981497	-10.294599	-10.240251	-1.518888	-1.675208	0.498134	-0.614390	47.652135
2	9.962401	-8.349849	23.248891	-24.196879	8.937480	10.965000	-7.490596	-3.025094	0.595807	0.382732	5.629537
3	-5.780709	-25.261584	1.383115	-11.786929	7.993078	-11.245752	-2.607351	-3.513896	-0.614235	-1.453979	-2.636676
4	1.211541	1.119963	7.512938	21.987312	-5.155392	10.339416	3.045180	-0.619230	-0.928068	0.405024	-16.683612
...	...	...	...	...	...	...	...	...	...	...	...
39995	-1.626076	9.127650	-18.741265	7.387842	3.403415	10.385736	8.824604	1.495547	-1.374972	2.083408	-31.389934
39996	26.420417	-16.714690	-1.641776	19.208386	-11.894191	-9.423328	4.025796	0.748295	-0.760492	1.216863	2.907223
39997	0.677569	-19.773004	-8.317459	4.646619	15.075550	-30.745632	-3.261740	-8.558190	-0.137755	-0.191201	-52.429693
39998	-9.532040	-24.989801	4.117245	-7.697699	25.884646	-8.961295	14.298122	-0.164484	0.387695	1.798387	26.476603
39999	16.695982	-14.407119	-11.654453	-20.180233	-3.187982	4.941689	-6.716241	-2.263061	-2.795973	2.519649	49.440257

40000 rows × 101 columns

In [82]:

```
df_train.describe().T
```

Out[82]:

	count	mean	std	min	25%	50%	75%	max
x0	39988.0	2.020255	9.590599	-36.842503	-4.461433	2.022412	8.389979	44.478690
x1	39990.0	-3.924559	18.768656	-79.156374	-16.591552	-4.061703	8.529110	77.682652
x2	39993.0	1.006619	21.062970	-89.728356	-13.230956	1.184946	15.221205	84.625640
x3	39987.0	-1.378330	29.397779	-126.652341	-21.297149	-1.224625	18.530623	117.004453
x4	39993.0	0.070199	20.243287	-76.412886	-13.580632	0.091600	13.722427	85.934044
...	...	...	...	...	...	...	...	...
x96	39986.0	-0.317345	9.321339	-42.409405	-6.561087	-0.228287	5.939217	38.649613
x97	39991.0	-0.562453	4.050658	-16.287032	-3.293697	-0.548699	2.138787	17.069095
x98	39996.0	0.000484	0.060034	-0.250606	-0.039977	0.000486	0.041186	0.221392
x99	39987.0	0.179715	4.506750	-18.876474	-2.879191	0.171954	3.237456	18.097897
y	40000.0	0.203675	0.402735	0.000000	0.000000	0.000000	0.000000	1.000000

95 rows × 8 columns

In [83]:

```
df_test
```

Out[83]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
0	0.519093	-4.606038	13.707586	-17.990903	12.873394	14.910935	2.915341	-10.110081	1.628317	-0.365064	10.646442
1	-12.357004	13.874141	14.052924	34.129247	34.511107	34.583336	-0.482540	-6.583407	-4.326799	-1.216928	-5.709141

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
2	[illegible]	[illegible]	44.873210	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.172269	[illegible]	[illegible]
3	20.972483	11.548506	-40.924625	-35.296796	-35.253101	-14.601890	5.045075	10.841771	-1.872260	-0.002583	60.212310
4	-9.916044	5.509811	31.749288	-0.803916	-4.005098	20.912490	0.419346	-2.949516	1.057176	-0.338547	25.056651
...	...	...	...	...	...	...	...	...	...	...	...
9995	8.941616	-9.832042	-4.865806	-43.242909	1.309220	-2.234482	-10.261111	-9.300236	-2.755621	2.482869	34.947087
9996	-0.444819	17.358679	2.954173	10.878828	25.144511	-7.924301	10.892586	-7.885435	0.446344	1.078755	5.738145
9997	-18.913055	-1.567725	16.495677	37.931367	20.314431	41.042133	2.687243	7.677554	-0.648772	0.812830	17.083768
9998	17.974482	-2.970413	-49.195814	16.316465	-24.203689	10.595619	2.005677	0.597577	1.024587	1.162438	-53.209016
9999	3.420462	-4.785548	23.126110	-11.496563	13.476099	16.957782	9.298587	13.992507	-0.035260	0.120473	-44.040601

10000 rows × 100 columns

Find total count of "Y" values

In [84]:

```
df_train['y'].value_counts()
```

Out[84]:

```
0    31853
1     8147
Name: y, dtype: int64
```

In [85]:

```
# List all data types in data set
df_train.dtypes.unique()
# Based on result Y column is the only interger type
```

Out[85]:

```
array([dtype('float64'), dtype('O'), dtype('int64')], dtype=object)
```

In [86]:

```
# Find all object columns
df_categorical = df_train.select_dtypes(include='object').columns.values.tolist()
df_categorical
```

Out[86]:

```
['x34', 'x35', 'x41', 'x45', 'x68', 'x93']
```

In [87]:

```
# Find all float columns
df_numerical = df_train.select_dtypes(include='float64').columns.values.tolist()
print(df_numerical)
```

```
['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11', 'x12', 'x13',
'x14', 'x15', 'x16', 'x17', 'x18', 'x19', 'x20', 'x21', 'x22', 'x23', 'x24', 'x25', 'x26
', 'x27', 'x28', 'x29', 'x30', 'x31', 'x32', 'x33', 'x36', 'x37', 'x38', 'x39', 'x40', '
x42', 'x43', 'x44', 'x46', 'x47', 'x48', 'x49', 'x50', 'x51', 'x52', 'x53', 'x54', 'x55'
, 'x56', 'x57', 'x58', 'x59', 'x60', 'x61', 'x62', 'x63', 'x64', 'x65', 'x66', 'x67', 'x
69', 'x70', 'x71', 'x72', 'x73', 'x74', 'x75', 'x76', 'x77', 'x78', 'x79', 'x80', 'x81',
'x82', 'x83', 'x84', 'x85', 'x86', 'x87', 'x88', 'x89', 'x90', 'x91', 'x92', 'x94', 'x95
', 'x96', 'x97', 'x98', 'x99']
```

Handle NaN values, correct spelling mistakes and normalize categorical data

In [88]:

```
# Print unique values
for column_name in  df_categorical:
  print(f'{column_name} : {df_train[column_name].unique()}')
```

```
x34 : ['chrystler' 'volkswagon' 'bmw' 'nissan' 'tesla' 'Toyota' 'Honda'
 'mercades' 'ford' 'chevrolet' nan]
x35 : ['thur' 'thurday' 'wed' 'tuesday' 'wednesday' 'friday' 'fri' 'monday' nan]
x41 : ['$-865.28' '$325.27' '$743.91' ... '$60.77' '$-982.23' '$-904.25']
x45 : ['0.02%' '-0.01%' '0.0%' '0.01%' '-0.0%' '-0.02%' '-0.03%' '0.03%' nan
 '0.04%' '-0.04%']
x68 : ['sept.' 'July' 'Jun' 'Nov' 'Mar' 'May' 'Oct' 'Aug' 'Apr' 'Dev' 'Feb'
 'January' nan]
x93 : ['asia' 'america' 'euorpe' nan]
```

In [89]:

```
# Imputation Categorical features
# Replace missing values with the most common class

df_train[df_categorical] = df_train[df_categorical].apply(lambda x: x.fillna(x.value_cou
nts().index[0]))
df_train[df_categorical].isnull().sum()
```

Out[89]:

```
x34    0
x35    0
x41    0
x45    0
x68    0
x93    0
dtype: int64
```

In [90]:

```
for column_name in  df_categorical:
  print(f'{column_name} : {df_train[column_name].unique()}')
```

```
x34 : ['chrystler' 'volkswagon' 'bmw' 'nissan' 'tesla' 'Toyota' 'Honda'
 'mercades' 'ford' 'chevrolet']
x35 : ['thur' 'thurday' 'wed' 'tuesday' 'wednesday' 'friday' 'fri' 'monday']
x41 : ['$-865.28' '$325.27' '$743.91' ... '$60.77' '$-982.23' '$-904.25']
x45 : ['0.02%' '-0.01%' '0.0%' '0.01%' '-0.0%' '-0.02%' '-0.03%' '0.03%' '0.04%'
 '-0.04%']
x68 : ['sept.' 'July' 'Jun' 'Nov' 'Mar' 'May' 'Oct' 'Aug' 'Apr' 'Dev' 'Feb'
 'January']
x93 : ['asia' 'america' 'euorpe']
```

In [91]:

```
# Build auto-complete vocabulary
auto_makers = {'bmw': {},
               'chrysler': {},
               'volkswagen': {},
               'nissan': {},
               'tesla': {},
               'toyota': {},
               'honda': {},
               'tesla': {},
               'mercedes': {},
               'ford': {},
               'chevrolet': {}
               }

weekdays = {'monday': {},
            'tuesday': {},
            'wednesday': {},
```

```
             'thursday':{},
             'friday':{},
             'sarturday':{},
             'sunday': {}
             }

countries = {'asia': {},
             'america': {},
             'europe': {}
             }

months = {'january': {},
          'february': {},
          'march': {},
          'april': {},
          'may': {},
          'june': {},
          'july': {},
          'august': {},
          'september': {},
          'october': {},
          'november': {},
          'december': {}
          }

words = {}
for d in [auto_makers, weekdays, countries, months]:
  words.update(d)
```

In [92]:

```
# Instantiate SpellChecker class
spell = SpellChecker()
known_words = ['bmw', 'mon', 'tue', 'wed', 'thur', 'fri', 'sat', 'sun', 'jan',
               'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct',
               'nov', 'dec']
# Load known words that does not need to be corrected
spell.word_frequency.load_words(known_words)
# Instantiate AutoComplete class with available vocabulary
autocomplete = AutoComplete(words=words)

for column_name in  ('x34', 'x35', 'x68', 'x93'):
  for value in df_train[column_name].unique():
    # Remove all special characters
    processed_value = re.sub('[^a-z0-9]+', '', value.lower())
    # Correct spelling mistakes
    processed_value = spell.correction(processed_value)
    # Auto-complete word
    processed_value = autocomplete.search(word=processed_value, max_cost=1, size=1)[0][0
]
    # Update value in column
    df_train[column_name] = df_train[column_name].map(lambda x: re.sub(f'^{value}$', pro
cessed_value, x))
```

In [93]:

```
# Remove special characters and convert string to float
for column_name in  ('x41', 'x45'):
  df_train[column_name] = df_train[column_name].map(lambda x: re.sub(r'[$,%]', '', x)).a
stype(float)
```

Handle NaN values and normalize numerical data

In [94]:

```
# Replace Nan (missing values) with mean value by using impute method for numerical data
imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')
df_num_imputed = imputer.fit_transform(df_train[df_numerical])
# Normaliza data bewtween 0 and 1 by using MixMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))
```

```
scaler.fit(df_num_imputed)
df_num_scaled = scaler.transform(df_num_imputed)
df_num_processed = pd.DataFrame(df_num_scaled, columns=df_numerical)
df_num_processed
```

Out[94]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x1
0	0.464900	0.546953	0.255253	0.558695	0.450864	0.586741	0.599612	0.545003	0.370513	0.230554	0.370748	0.25304
1	0.431283	0.354159	0.509104	0.388542	0.407265	0.428470	0.412410	0.404192	0.479079	0.445984	0.650158	0.60314
2	0.575556	0.451460	0.647976	0.420491	0.525728	0.571547	0.313247	0.376728	0.485702	0.541324	0.512483	0.55305
3	0.381964	0.343631	0.522566	0.471423	0.519911	0.421686	0.394335	0.366783	0.403650	0.365706	0.485401	0.42115
4	0.467947	0.511839	0.557723	0.610037	0.438921	0.567326	0.488198	0.425676	0.382370	0.543455	0.439381	0.60256
...	...	...	...	...	...	...	...	...	...	...	...	..
39995	0.433053	0.562896	0.407143	0.550119	0.491640	0.567639	0.584167	0.468702	0.352066	0.703935	0.391200	0.40138
39996	0.777939	0.398126	0.505217	0.598632	0.397412	0.433982	0.504481	0.453499	0.393733	0.621080	0.503564	0.42920
39997	0.461381	0.378626	0.466929	0.538868	0.563537	0.290116	0.383469	0.264155	0.435960	0.486447	0.322269	0.46758
39998	0.335835	0.345364	0.538247	0.488206	0.630117	0.437100	0.675057	0.434928	0.471590	0.676682	0.580782	0.65744
39999	0.658358	0.412839	0.447790	0.436976	0.451040	0.530907	0.326106	0.392232	0.255709	0.745646	0.656016	0.57988

40000 rows × 94 columns

Encode categorical data to binary by using dummy encoder

In [95]:

```
# Encode categorial data to binary by using dummy encoder
df_categorical = df_train.select_dtypes(include='object').columns.values.tolist()
df_categ_processed = pd.get_dummies(df_train[df_categorical], columns = df_categorical)
df_categ_processed
```

Out[95]:

	x34_bmw	x34_chevrolet	x34_chrysler	x34_ford	x34_honda	x34_mercedes	x34_nissan	x34_tesla	x34_toyota	x34_vol
0	0	0	1	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	1	0	0	
4	0	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	
39995	0	0	0	0	0	0	0	0	0	
39996	0	0	0	0	0	0	0	1	0	
39997	0	0	0	0	0	0	0	0	0	
39998	0	0	0	0	0	0	0	0	0	
39999	0	0	0	0	0	0	0	0	0	

40000 rows × 30 columns

Concatenate numerical, categorical and "Y" columns back to Pandas DataFrame

In [96]:

```
# Concatenate back numerical, categorical and y columns
df_processed = pd.concat([df_num_processed, df_categ_processed, df_train['y'] ], axis=1,
sort=False)
df_processed
```

Out[96]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x1
0	0.464900	0.546953	0.255253	0.558695	0.450864	0.586741	0.599612	0.545003	0.370513	0.230554	0.370748	0.25304
1	0.431283	0.354159	0.509104	0.388542	0.407265	0.428470	0.412410	0.404192	0.479079	0.445984	0.650158	0.60314
2	0.575556	0.451460	0.647976	0.420491	0.525728	0.571547	0.313247	0.376728	0.485702	0.541324	0.512483	0.55305
3	0.381964	0.343631	0.522566	0.471423	0.519911	0.421686	0.394335	0.366783	0.403650	0.365706	0.485401	0.42115
4	0.467947	0.511839	0.557723	0.610037	0.438921	0.567326	0.488198	0.425676	0.382370	0.543455	0.439381	0.60256
...	...	...	...	...	...	...	...	...	...	...	...	.
39995	0.433053	0.562896	0.407143	0.550119	0.491640	0.567639	0.584167	0.468702	0.352066	0.703935	0.391200	0.40138
39996	0.777939	0.398126	0.505217	0.598632	0.397412	0.433982	0.504481	0.453499	0.393733	0.621080	0.503564	0.42920
39997	0.461381	0.378626	0.466929	0.538868	0.563537	0.290116	0.383469	0.264155	0.435960	0.486447	0.322269	0.46758
39998	0.335835	0.345364	0.538247	0.488206	0.630117	0.437100	0.675057	0.434928	0.471590	0.676682	0.580782	0.65744
39999	0.658358	0.412839	0.447790	0.436976	0.451040	0.530907	0.326106	0.392232	0.255709	0.745646	0.656016	0.57988

40000 rows × 125 columns

Use Correlation Matrix to find highly correlated features

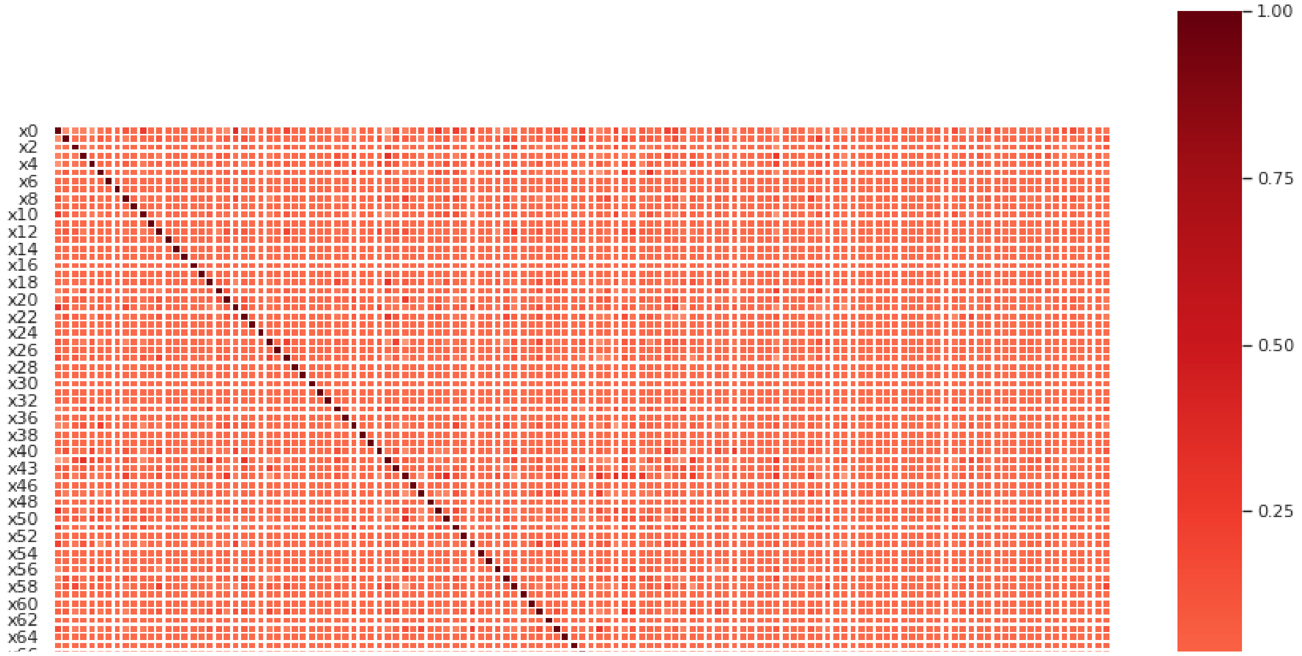
In [97]:

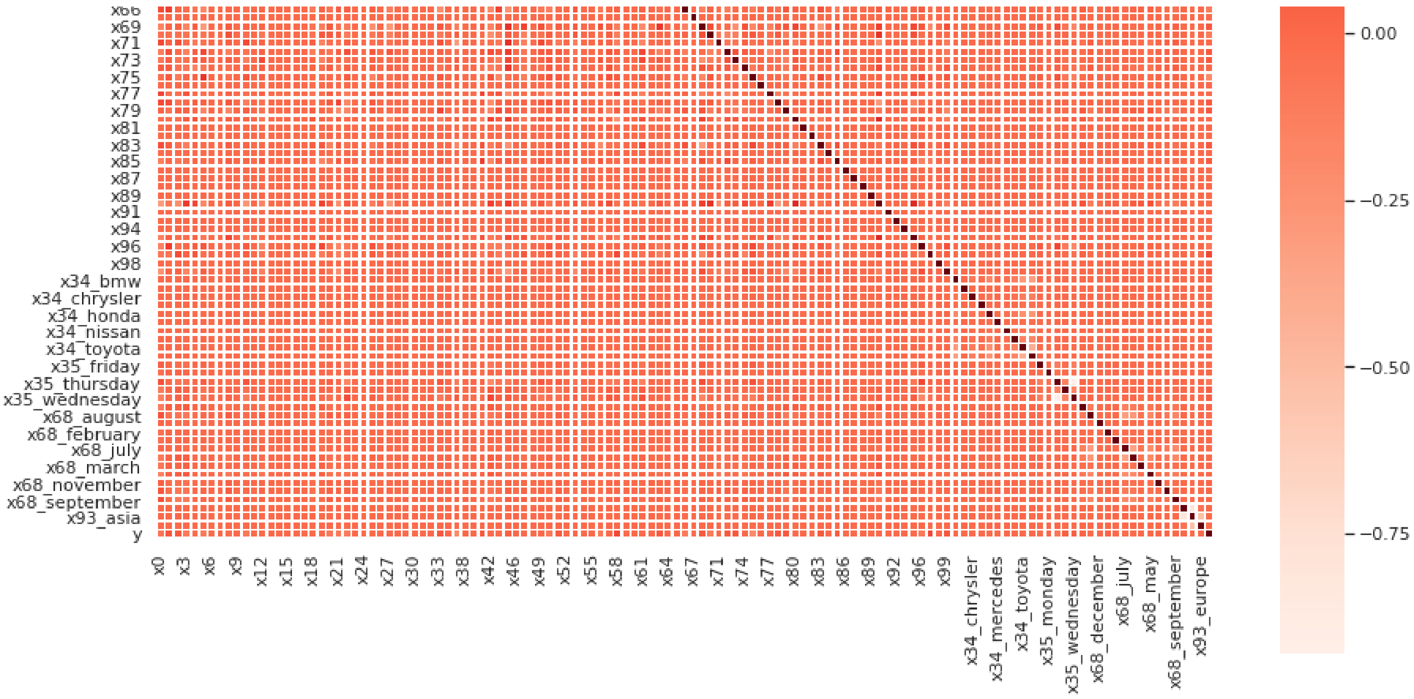
```
#Using Pearson Correlation
plt.figure(figsize=(12,12))
cor = df_processed .corr()
plt.figure(figsize=(15,15))
sns.set(font_scale=1)
sns.heatmap(data=cor,
            center=0,
            cmap=plt.cm.Reds,
            square=True, linewidth=0.5)
```

Out[97]:

<matplotlib.axes._subplots.AxesSubplot at 0x7f06d94fb110>

<Figure size 864x864 with 0 Axes>





In [98]:

```
#Correlation with output variable
cor_target = abs(cor['y'])
#Selecting highly correlated features
relevant_features_score = cor_target[cor_target>0.01]
relevant_features = relevant_features_score.index.to_list()
print(relevant_features)
print(len(relevant_features))
```

```
['x0', 'x1', 'x2', 'x3', 'x5', 'x10', 'x20', 'x21', 'x22', 'x29', 'x33', 'x37', 'x38', 'x
40', 'x44', 'x48', 'x50', 'x51', 'x53', 'x56', 'x58', 'x63', 'x66', 'x69', 'x70', 'x72',
'x73', 'x75', 'x78', 'x79', 'x83', 'x85', 'x96', 'x97', 'x99', 'x35_monday', 'x35_thursda
y', 'x35_tuesday', 'x35_wednesday', 'x68_april', 'x68_august', 'x68_february', 'x68_july'
, 'x68_march', 'x68_may', 'x68_november', 'x68_october', 'y']
48
```

Create Train/Test/Validation datasets

In [99]:

```
# Split the data into training,testing and validation sets using train_test_split
X_train, X_test, y_train, y_test = \
        train_test_split(df_processed[relevant_features].drop(columns=['y']), df_process
ed ['y'] ,test_size=0.3, random_state=42)
X_test, X_val, y_test, y_val = \
        train_test_split(X_test, y_test, test_size=0.50, random_state=42)
```

In [100]:

```
print(f'X_train: {X_train.shape}, y_train: {y_train.shape}',
      f'X_test:  {X_test.shape},  y_train: {y_test.shape}',
      f'X_val:   {X_val.shape},  y_train: {y_val.shape}', sep='\n')
```

```
X_train: (28000, 47), y_train: (28000,)
X_test:  (6000, 47),  y_train: (6000,)
X_val:   (6000, 47),  y_train: (6000,)
```

Model Building

Logistic Regression Model

In [101]:

```
# Build Logistic Regresssion model
logistic_regression_model = LogisticRegression(penalty='l2', solver='sag', max_iter=1000)
# fit the model
logistic_regression_model.fit(X_train, y_train)
```

Out[101]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='sag', tol=0.0001, verbose=0,
                   warm_start=False)
```

Model Evaluation

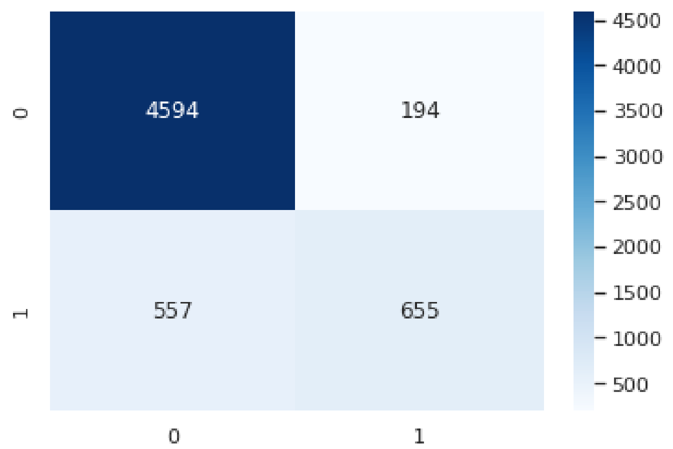
In [102]:

```
y_pred = logistic_regression_model.predict(X_test)
accuracy = accuracy_score(y_test,y_pred)
print(f'Accuracy of Logistic Regression Classifier on Test dataset: {accuracy:.2f}\n')
# Plot Confuzion Matrix
fig, ax =plt.subplots(1,1)
conf_matrix = confusion_matrix(y_test, y_pred)
print(classification_report(y_test, y_pred))
sns.heatmap(conf_matrix, annot=True, cmap='Blues',  fmt='d',  xticklabels=[0,1],yticklab
els=[0,1])
```

```
Accuracy of Logistic Regression Classifier on Test dataset: 0.87

              precision    recall  f1-score   support

           0       0.89      0.96      0.92      4788
           1       0.77      0.54      0.64      1212

    accuracy                           0.87      6000
   macro avg       0.83      0.75      0.78      6000
weighted avg       0.87      0.87      0.87      6000
```

Out[102]:

```
&lt;matplotlib.axes._subplots.AxesSubplot at 0x7f06c10d4250&gt;
```



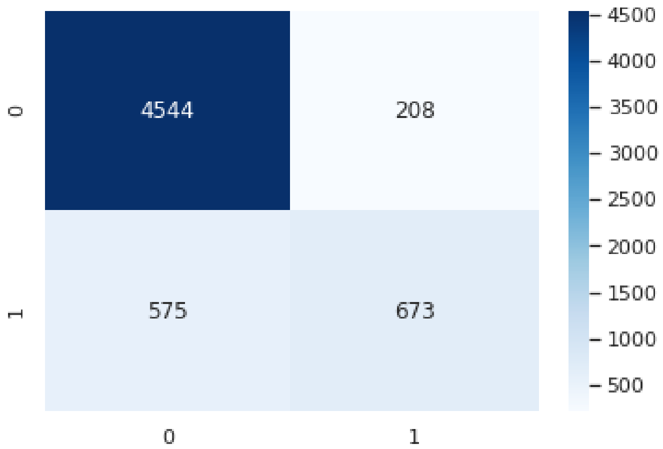
In [103]:

```
y_pred = logistic_regression_model.predict(X_val)
accuracy = accuracy_score(y_val,y_pred)
print(f'Accuracy of Logistic Regression Classifier on Validation dataset: {accuracy:.2f}\
n')
# Plot Confuzion Matrix
fig, ax =plt.subplots(1,1)
conf_matrix = confusion_matrix(y_val, y_pred)
print(classification_report(y_test, y_pred))
sns.heatmap(conf_matrix, annot=True, cmap='Blues',  fmt='d',  xticklabels=[0,1],yticklab
els=[0,1])
```

```
Accuracy of Logistic Regression Classifier on Validation dataset: 0.87

              precision    recall  f1-score   support

           0       0.80      0.85      0.82      4788
           1       0.20      0.15      0.17      1212

    accuracy                           0.71      6000
   macro avg       0.50      0.50      0.50      6000
weighted avg       0.68      0.71      0.69      6000
```

Out[103]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f06c1081a90>
```



Random Forest Model

In [104]:

```
random_forest_model = RandomForestClassifier()
random_forest_model.fit(X_train,y_train)
```

Out[104]:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Model Evaluation

In [105]:

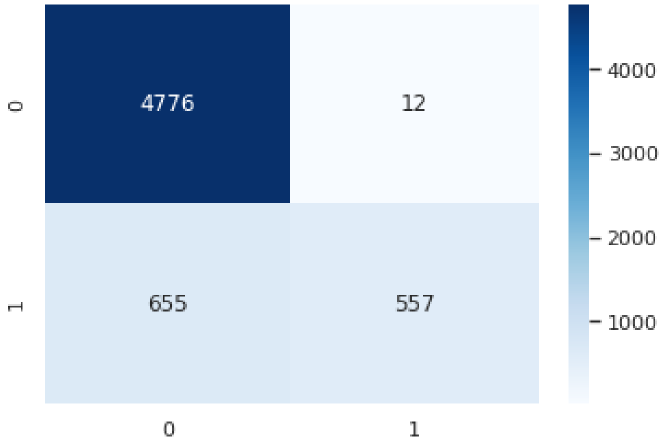
```
y_pred = random_forest_model.predict(X_test)
accuracy = accuracy_score(y_test,y_pred)
print(f'Accuracy of Random Forest Classifier on Test dataset: {accuracy:.2f}\n')
# Plot Confuzion Matrix
fig, ax =plt.subplots(1,1)
conf_matrix = confusion_matrix(y_test, y_pred)
print(classification_report(y_test, y_pred))
sns.heatmap(conf_matrix, annot=True, cmap='Blues',  fmt='d',  xticklabels=[0,1],yticklab
els=[0,1])
```

```
Accuracy of Random Forest Classifier on Test dataset: 0.89

              precision    recall  f1-score   support

           0       0.88      1.00      0.93      4788
           1       0.98      0.46      0.63      1212
```

```
    accuracy                           0.89      6000
   macro avg       0.93      0.73      0.78      6000
weighted avg       0.90      0.89      0.87      6000
```

Out[105]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f06d91ddb10>
```



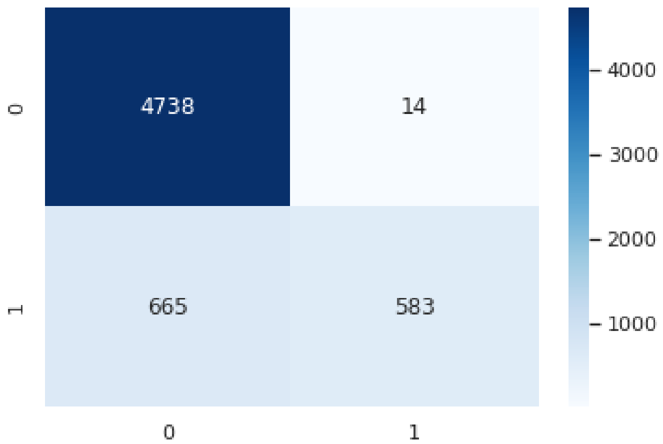
In [108]:

```python
y_pred = random_forest_model.predict(X_val)
accuracy = accuracy_score(y_val,y_pred)
print(f'Accuracy of Random Forest Classifier on Validation dataset: {accuracy:.2f}\n')
# Plot Confuzion Matrix
fig, ax =plt.subplots(1,1)
conf_matrix = confusion_matrix(y_val, y_pred)
print(classification_report(y_val, y_pred))
sns.heatmap(conf_matrix, annot=True, cmap='Blues',  fmt='d',  xticklabels=[0,1],yticklab
els=[0,1])
```

```
Accuracy of Random Forest Classifier on Validation dataset: 0.89

              precision    recall  f1-score   support

           0       0.88      1.00      0.93      4752
           1       0.98      0.47      0.63      1248

    accuracy                           0.89      6000
   macro avg       0.93      0.73      0.78      6000
weighted avg       0.90      0.89      0.87      6000
```

Out[108]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f06d8fff710>
```



Artificial Neural Network (ANN) model

In [109]:

```
def model_with_2_layers():
  """Build ANN model with 2 Dense layers."""
  model = Sequential()
  model.add(Dense(50, input_dim=X_train.shape[1], activation='relu', name='input_layer')
)
  model.add(Dense(25, activation='relu', name='layer_1'))
  model.add(Dense(1, activation='sigmoid', name='output_layer'))

  model.compile(loss='binary_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])
  return model
```

In [113]:

```
ann_model = model_with_2_layers()
ann_model.fit(X_train,
       y_train,
       epochs=70,
       shuffle=True, # shuffle data randomly.
       verbose=0, # this will tell keras to print more detailed info
       validation_data=(X_test, y_test)
       )
```

Out[113]:

```
<tensorflow.python.keras.callbacks.History at 0x7f06d8c58f10>
```

Model Evaluation

In [114]:

```
# Evaluate ANN model on the test data using `evaluate`
results = ann_model.evaluate(X_val, y_val, batch_size=128, verbose=0)
print(f'Accuracy: {results[1]:.2f}',
      f'Loss:     {results[0]:.3f}',
      sep='\n')
```

```
Accuracy: 0.97
Loss:     0.106
```

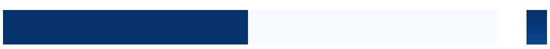
In [115]:

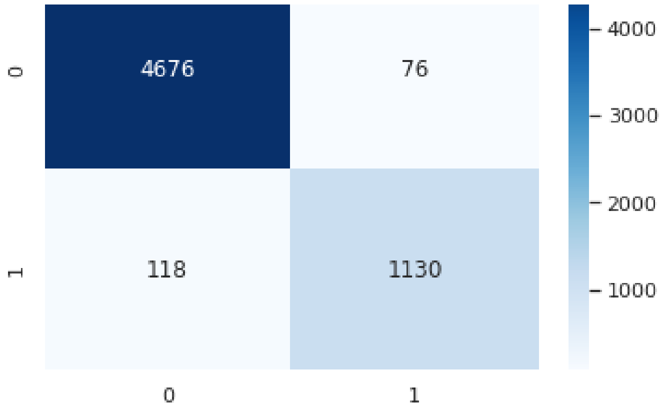
```
y_pred = (ann_model.predict(X_val) > 0.5).astype("int32").flatten()
accuracy = accuracy_score(y_val,y_pred)
print(f'Accuracy of Artificial Neural Network on Validation dataset: {accuracy:.2f}\n')
# Plot Confuzion Matrix
fig, ax =plt.subplots(1,1)
conf_matrix = confusion_matrix(y_val, y_pred)
print(classification_report(y_pred, y_val))
sns.heatmap(conf_matrix, annot=True, cmap='Blues',  fmt='d',  xticklabels=[0,1],yticklab
els=[0,1])
```

```
Accuracy of Artificial Neural Network on Validation dataset: 0.97

              precision    recall  f1-score   support

           0       0.98      0.98      0.98      4794
           1       0.91      0.94      0.92      1206

    accuracy                           0.97      6000
   macro avg       0.94      0.96      0.95      6000
weighted avg       0.97      0.97      0.97      6000
```

Out[115]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f06d8abab10>
```





Preprocessing Prediction Dataset

In [116]:

```
def process_categorical_data(df):
  """
    Correct spellig mistakes, auto-completing words for categorical data
    and removes special charcters and convert categorical data to numerical.
  """
  auto_makers = {'bmw': {},
                 'chrysler': {},
                 'volkswagen': {},
                 'nissan': {},
                 'tesla': {},
                 'toyota': {},
                 'honda': {},
                 'tesla': {},
                 'mercedes': {},
                 'ford': {},
                 'chevrolet': {}
                 }

  weekdays = {'monday': {},
              'tuesday': {},
              'wednesday': {},
              'thursday':{},
              'friday':{},
              'sarturday':{},
              'sunday': {}
              }

  countries = {'asia': {},
               'america': {},
               'europe': {}
              }

  months = {'january': {},
            'february': {},
            'march': {},
            'april': {},
            'may': {},
            'june': {},
            'july': {},
            'august': {},
            'september': {},
            'october': {},
            'november': {},
            'december': {}
            }

  words = {}

  for d in [auto_makers, weekdays, countries, months]:
    words.update(d)
```

```
    spell = SpellChecker()
    known_words = ['bmw', 'mon', 'tue', 'wed', 'thur', 'fri', 'sat', 'sun', 'jan',
                   'feb', 'mar', 'apr', 'may', 'jun', 'jul', 'aug', 'sep', 'oct',
                   'nov', 'dec']
    spell.word_frequency.load_words(known_words)
    autocomplete = AutoComplete(words=words)

    # Get a list of categorical features
    df_categorical = df.select_dtypes(include='object').columns.values.tolist()

    # Imputation Categorical features
    # Replace missing values with the most common class
    df[df_categorical] = df[df_categorical].apply(lambda x: x.fillna(x.value_counts().inde
x[0]))

    for column_name in  ('x34', 'x35', 'x68', 'x93'):
      for value in df[column_name].unique():
        # Remove all special characters
        processed_value = re.sub('[^a-z0-9]+', '', value.lower())
        # Correct spelling mistakes
        processed_value = spell.correction(processed_value)
        # Auto-complete word
        processed_value = autocomplete.search(word=processed_value, max_cost=1, size=1)[0]
[0]
        # Update value in column
        df[column_name] = df[column_name].map(lambda x: re.sub(f'^{value}$', processed_val
ue, x))

    # Remove special characters and convert string to float
    for column_name in  ('x41', 'x45'):
      df[column_name] = df[column_name].map(lambda x: re.sub(r'[$,%]', '', x)).astype(floa
t)


def impute_numerical_data(df):
    """ Use impute method to replace missing values with mean value and
        normalize data between 0 and 1 with MinMax Scaler.
     """
    # Get a list of numerical features
    df_numerical = df.select_dtypes(include='float64').columns.values.tolist()

    imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')
    df_num_imputed = imputer.fit_transform(df[df_numerical])
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaler.fit(df_num_imputed)
    df_num_scaled = scaler.transform(df_num_imputed)

    return pd.DataFrame(df_num_scaled, columns=df_numerical)


def dummy_encoding_categorical(df):
    """ Encode catrgorical data to binary."""
    # Get a list of categorical features
    df_categorical = df.select_dtypes(include='object').columns.values.tolist()
    return pd.get_dummies(df[df_categorical], columns = df_categorical)


def preprocess_data(df):
    """
      Pre-processe data by dealing with missing values and normalize numerical and
      categorical data.
    """
    # Process categorical data in place
    process_categorical_data(df)
    # Impute numerical data with mean value and normalize data between 0 and 1
    numerical_df = impute_numerical_data(df)
    #  print(numerical_df)
    # Categorical data encoding method transforms the categorical variable
    # into a set of binary variables
    categorical_df = dummy_encoding_categorical(df)
    if 'y' in df:
        return pd.concat([numerical_df, categorical_df, df['y'] ], axis=1, sort=False)
```

```python
    return pd.concat([numerical_df, categorical_df], axis=1, sort=False)
```

In [117]:

```python
df_processed_test = preprocess_data(df_test)
df_processed_test
```

Out[117]:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11
0	0.458511	0.515841	0.598980	0.442302	0.580890	0.630122	0.504635	0.267560	0.650089	0.480551	0.513906	0.605543
1	0.294616	0.642884	0.601222	0.661994	0.722634	0.775662	0.443910	0.354744	0.147479	0.397320	0.450618	0.649024
2	0.475259	0.565828	0.218643	0.610623	0.562741	0.563917	0.350060	0.417085	0.498120	0.525605	0.559061	0.594119
3	0.718854	0.626896	0.244279	0.369356	0.265623	0.411780	0.542696	0.785519	0.354641	0.515968	0.705703	0.303692
4	0.325686	0.585383	0.716116	0.514747	0.470322	0.674522	0.460028	0.444579	0.601885	0.483142	0.569667	0.589739
...	...	...	...	...	...	...	...	...	...	...	...	...
9995	0.565718	0.479915	0.478392	0.335862	0.505135	0.503276	0.269152	0.287581	0.280086	0.758810	0.607938	0.416810
9996	0.446242	0.666839	0.529163	0.563991	0.661275	0.461182	0.647200	0.322557	0.550331	0.621620	0.494913	0.329193
9997	0.211167	0.536728	0.617082	0.678020	0.629634	0.823446	0.500558	0.707295	0.457903	0.595638	0.538816	0.459538
9998	0.680694	0.527085	0.190578	0.586911	0.338006	0.598196	0.488378	0.532268	0.599134	0.629796	0.266816	0.437605
9999	0.495441	0.514607	0.660130	0.469676	0.584838	0.645265	0.618713	0.863409	0.509684	0.527991	0.302293	0.595890

10000 rows × 126 columns

Model prediction

In [118]:

```python
# Exclude Y column from relevent features
relevant_features = [x for x in relevant_features if x != 'y']
```

Logist Regression Prediction

In [119]:

```python
y_pred = logistic_regression_model.predict(df_processed_test[relevant_features])
final_prediction = pd.DataFrame(y_pred, columns=['y'])
# Create final csv file
final_prediction.to_csv("result1.csv", index=False, header=False)
```

Random Forest Prediction

In [42]:

```python
y_pred = random_forest_model.predict(df_processed_test[relevant_features])
final_prediction = pd.DataFrame(y_pred, columns=['y'])
# Create final csv file
final_prediction.to_csv("result2.csv", index=False, header=False)
```

ANN Prediction

In [43]:

```python
y_pred = (ann_model.predict(df_processed_test[relevant_features]) > 0.5).astype("int32")
.flatten()
final_prediction = pd.DataFrame(y_pred, columns=['y'])
# Create final csv file
```

```
final_prediction.to_csv("result3.csv", index=False, header=False)
```