Prepare python environment

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

%matplotlib inline
```

```
random_state=5 # use this to control randomness across runs e.g., dataset partitioning
```

Preparing the Glass Dataset (2 points)

We will use glass dataset from UCI machine learning repository. Details for this data can be found here. The objective of the dataset is to identify the class of glass based on the following features:

1. RI: refractive index
2. Na: Sodium
3. Mg: Magnesium
4. Al: Aluminum
5. Si: Silica
6. K: Potassium
7. Ca: Calcium
8. Ba: Barium
9. Fe: Iron
10. Type of glass (Target label)

The classes of glass are:

1. building_windows_float_processed
2. building_windows_non_float_processed
3. vehicle_windows_float_processed
4. containers
5. tableware
6. headlamps

Identification of glass from its content can be used for forensic analysis.

Loading the dataset

```
# Download and load the dataset
import os
if not os.path.exists('glass.csv'):
    !wget https://raw.githubusercontent.com/JHA-Lab/ece364/main/dataset/glass.csv
data = pd.read_csv('glass.csv')
# Display the first five instances in the dataset
data.head(5)
```

```
--2021-10-07 03:22:54--  https://raw.githubusercontent.com/JHA-Lab/ece364/main/dataset/g
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:44
HTTP request sent, awaiting response... 200 OK
Length: 9838 (9.6K) [text/plain]
Saving to: ‘glass.csv’

glass.csv           100%[===================>]   9.61K  --.-KB/s    in 0s

2021-10-07 03:22:54 (77.7 MB/s) - ‘glass.csv’ saved [9838/9838]
```

	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
0	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.0	1
1	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.0	1
2	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.0	1
3	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.0	1
4	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.0	1

```
# display some stats
data.describe()
```

RI Na Mg Al Si K Ca

Look at some statistics of the data using the `describe` function in pandas.

mean 1.518365 13.407850 2.684533 1.444907 72.650935 0.497056 8.956963

```
data.describe()
```

	RI	Na	Mg	Al	Si	K	Ca
count	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000	214.000000
mean	1.518365	13.407850	2.684533	1.444907	72.650935	0.497056	8.956963
std	0.003037	0.816604	1.442408	0.499270	0.774546	0.652192	1.423153
min	1.511150	10.730000	0.000000	0.290000	69.810000	0.000000	5.430000
25%	1.516523	12.907500	2.115000	1.190000	72.280000	0.122500	8.240000
50%	1.517680	13.300000	3.480000	1.360000	72.790000	0.555000	8.600000
75%	1.519157	13.825000	3.600000	1.630000	73.087500	0.610000	9.172500
max	1.533930	17.380000	4.490000	3.500000	75.410000	6.210000	16.190000

```
# Check type of data in each column
data.info()
```

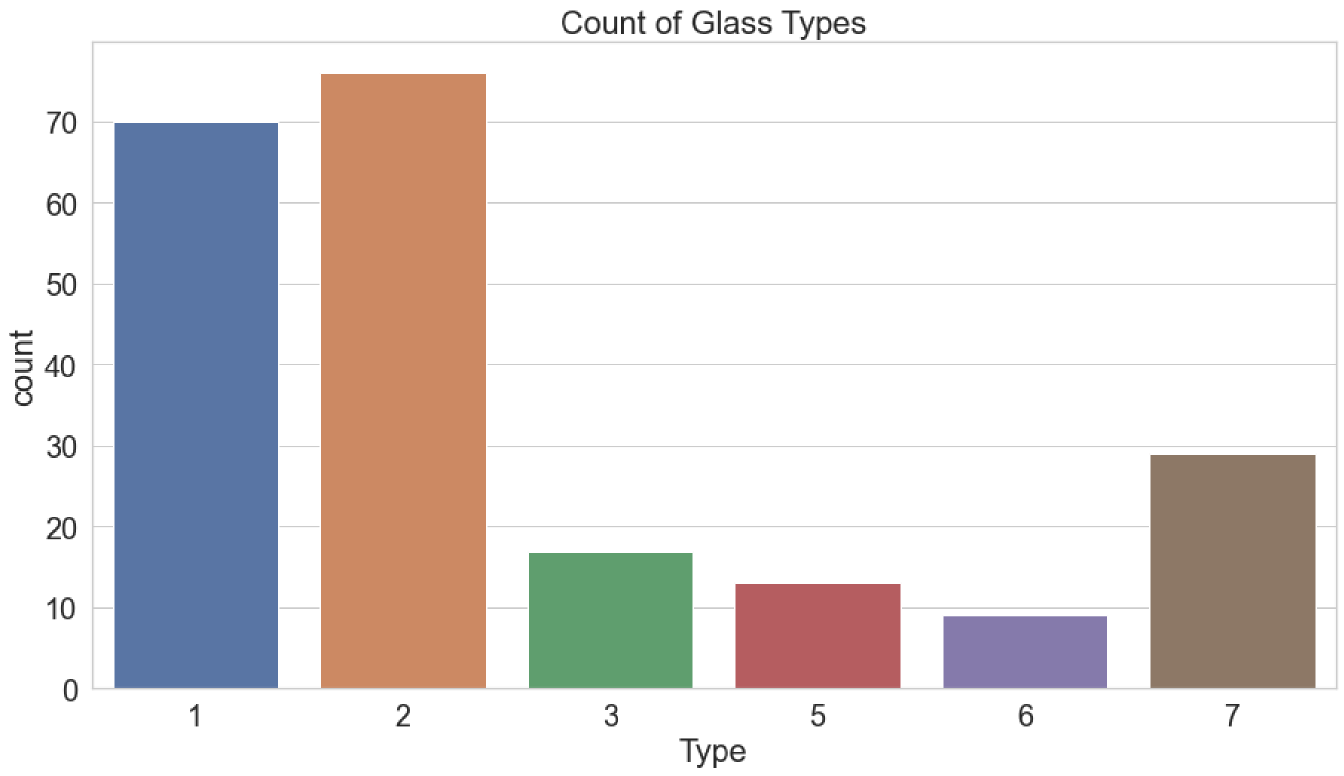
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 214 entries, 0 to 213
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   RI      214 non-null    float64
 1   Na      214 non-null    float64
 2   Mg      214 non-null    float64
 3   Al      214 non-null    float64
 4   Si      214 non-null    float64
 5   K       214 non-null    float64
 6   Ca      214 non-null    float64
 7   Ba      214 non-null    float64
 8   Fe      214 non-null    float64
 9   Type    214 non-null    int64  
dtypes: float64(9), int64(1)
memory usage: 16.8 KB
```

Visualize the Data

Check how many classes of each type of glass are there in the data. This has been done for you.

```
sns.set(style="whitegrid", font_scale=1.8)
plt.subplots(figsize = (15,8))
sns.countplot(x='Type',data=data).set_title('Count of Glass Types')
```

```
    Text(0.5, 1.0, 'Count of Glass Types')
```



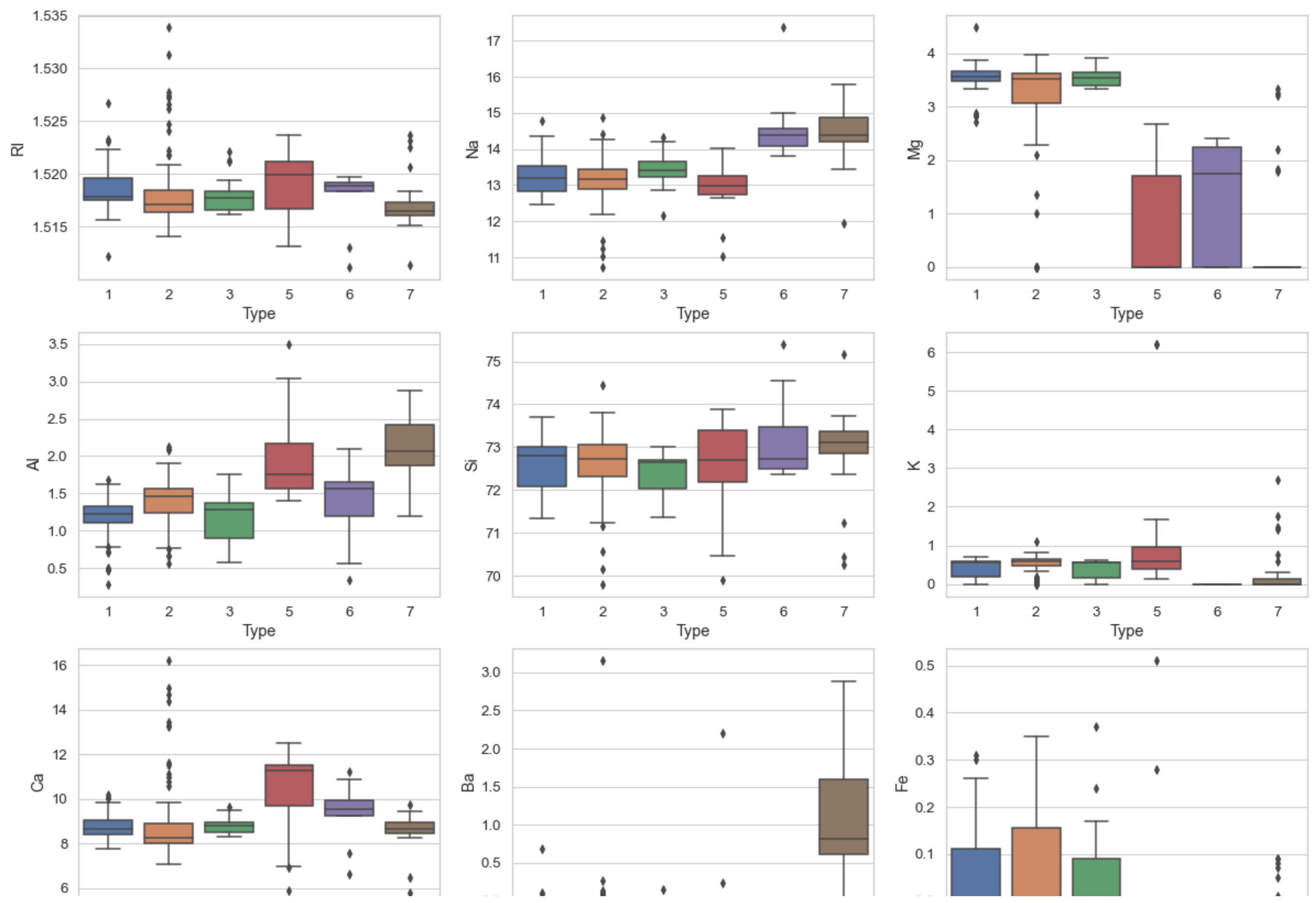
Calculate `mean` material content for each kind of glass. This has been done for you

```
# Compute mean material content for each kind of glass
data.groupby('Type', as_index=False).mean()
```

	Type	RI	Na	Mg	Al	Si	K	Ca	Ba
2	3	1.517964	13.437059	3.543529	1.201176	72.404706	0.406471	8.782941	0.008824

Create box plot to see distribution of each content in the glass. See here for further details. This has been done for you.

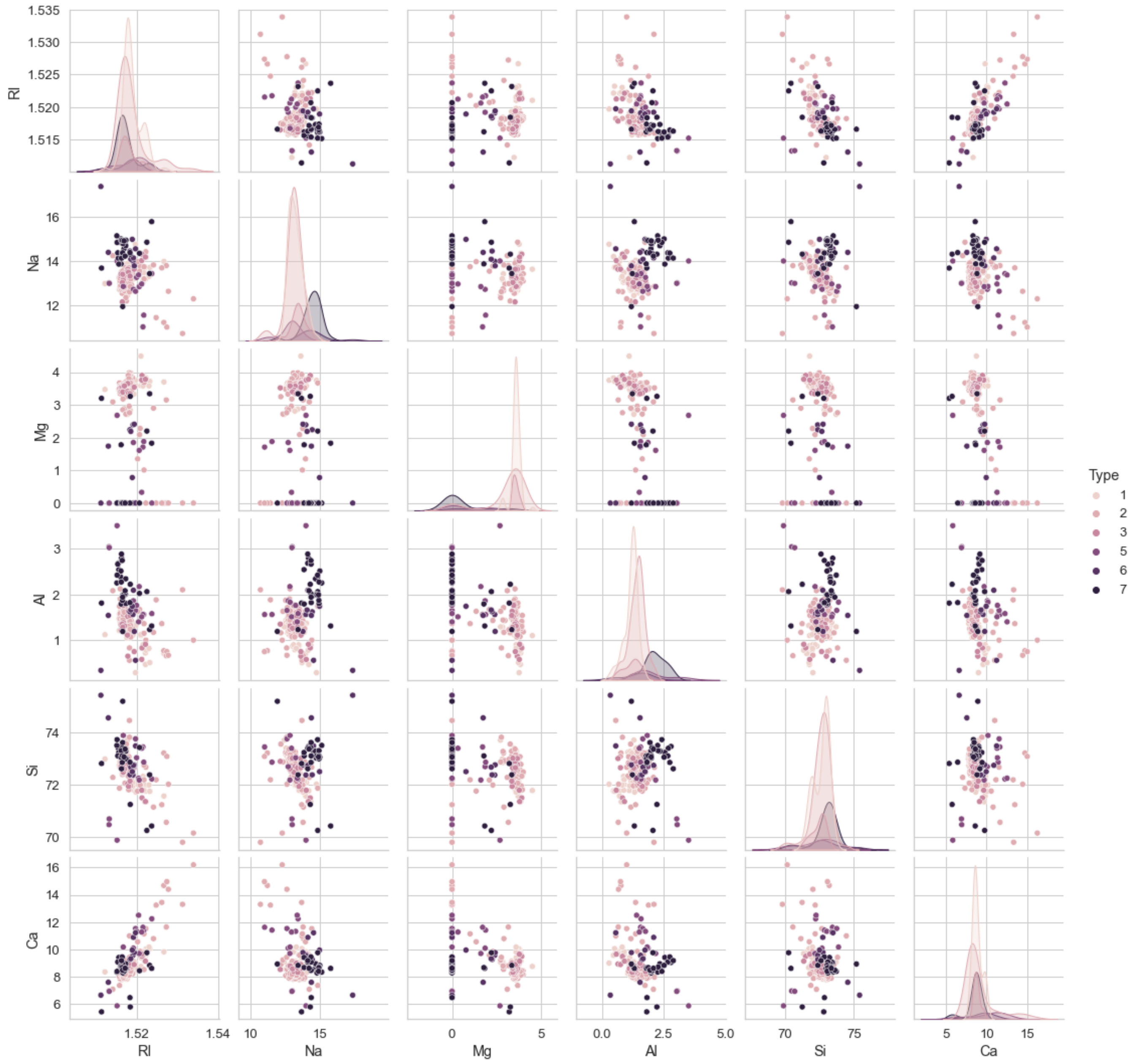
```
sns.set(style="whitegrid", font_scale=1.2)
plt.subplots(figsize = (20,15))
plt.subplot(3,3,1)
sns.boxplot(x='Type', y='RI', data=data)
plt.subplot(3,3,2)
sns.boxplot(x='Type', y='Na', data=data)
plt.subplot(3,3,3)
sns.boxplot(x='Type', y='Mg', data=data)
plt.subplot(3,3,4)
sns.boxplot(x='Type', y='Al', data=data)
plt.subplot(3,3,5)
sns.boxplot(x='Type', y='Si', data=data)
plt.subplot(3,3,6)
sns.boxplot(x='Type', y='K', data=data)
plt.subplot(3,3,7)
sns.boxplot(x='Type', y='Ca', data=data)
plt.subplot(3,3,8)
sns.boxplot(x='Type', y='Ba', data=data)
plt.subplot(3,3,9)
sns.boxplot(x='Type', y='Fe', data=data)
plt.show()
```



Create a pairplot to display pairwise relationship. See here for further details. This has been done for you.

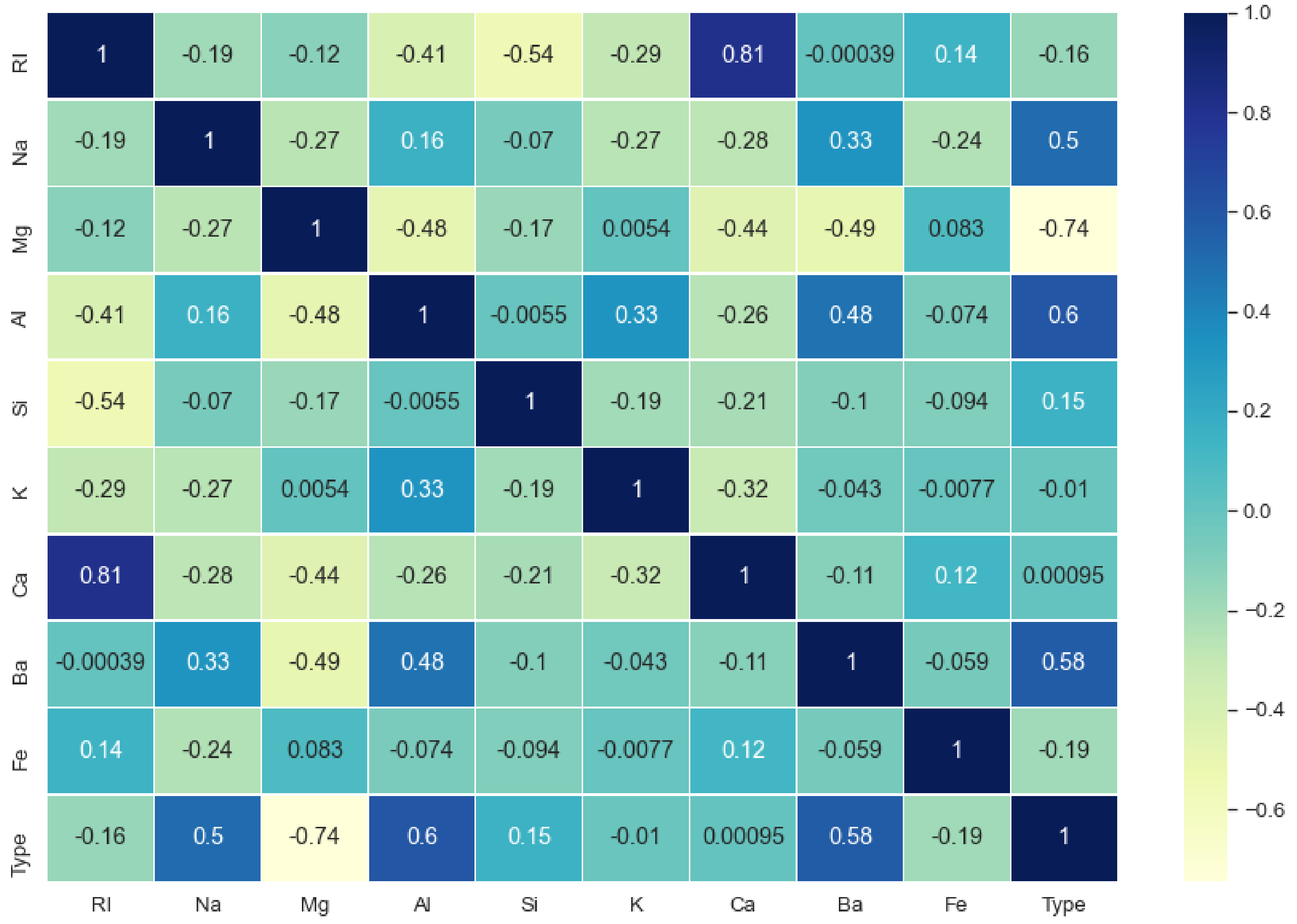
```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
sns.pairplot(data[['RI','Na','Mg','Al','Si','Ca','Type']], hue='Type')
```

```
<seaborn.axisgrid.PairGrid at 0x7fe1e8b957b8>
```



```
# Plot heatmap showing correlation between different features
plt.subplots(figsize=(15,10))
sns.heatmap(data.corr(),cmap='YlGnBu',annot=True, linewidth=.5)
```

```
<AxesSubplot:>
```



Extract target and descriptive features (1 point)

Add the following features to the dataset to model interactions between the pairs of glass materials. (See here for an example.)

```
- Ca*Na
- Al*Mg
- Ca*Mg
- Ca*RI
```

```
# Additional features to be added to the data
data['Ca_Na'] = data.Ca*data.Na
data['Al_Mg'] = data.Al*data.Mg
data['Ca_Mg'] = data.Ca*data.Mg
data['Ca_RI'] = data.Ca*data.RI
```

Separate the target and features from the data.

```
# Store all the features from the data in X
X= data.drop('Type',axis=1)
print(X)
# Store all the labels in y
y= data['Type']
print(y)
```

```
          RI     Na    Mg    Al  ...     Ca_Na   Al_Mg    Ca_Mg      Ca_RI
0    1.52101  13.64  4.49  1.10  ...  119.3500  4.9390  39.2875  13.308837
1    1.51761  13.89  3.60  1.36  ...  108.7587  4.8960  28.1880  11.882886
2    1.51618  13.53  3.55  1.54  ...  105.2634  5.4670  27.6190  11.795880
3    1.51766  13.21  3.69  1.29  ...  108.5862  4.7601  30.3318  12.475165
4    1.51742  13.27  3.62  1.24  ...  107.0889  4.4888  29.2134  12.245579
..       ...    ...   ...   ...  ...       ...     ...      ...        ...
209  1.51623  14.14  0.00  2.88  ...  129.8052  0.0000   0.0000  13.918991
210  1.51685  14.92  0.00  1.99  ...  125.3280  0.0000   0.0000  12.741540
211  1.52065  14.36  0.00  2.02  ...  121.1984  0.0000   0.0000  12.834286
212  1.51651  14.38  0.00  1.94  ...  121.9424  0.0000   0.0000  12.860005
213  1.51711  14.23  0.00  2.08  ...  122.6626  0.0000   0.0000  13.077488

[214 rows x 13 columns]
0      1
1      1
2      1
3      1
4      1
      ..
209    7
210    7
211    7
212    7
213    7
Name: Type, Length: 214, dtype: int64
```

```
# Convert data to numpy array
X = X.to_numpy()
y = y.to_numpy()
```

Create training and validation datasets (1 point)

We will split the dataset into training and validation set. Generally in machine learning, we split the data into training, validation and test set (this will be covered in later chapters). The model with best performance on the validation set is used to evaluate perfromance on the test set which is the unseen data. In this assignment, we will using `train set` for training and evaluate the performance on the `test set` for various model configurations to determine the best hyperparameters (parameter setting yielding the best performance).

Split the data into training and validation set using `train_test_split`. See here for details. To get consistent result while splitting, set `random_state` to the value defined earlier. We use 80% of the

```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=.2,random_state=random_state)
```

Preprocess the dataset by normalizing each feature to have zero mean and unit standard deviation. This can be done using `StandardScaler()` function. See here for more details.

```
# Define the scaler for scaling the data
scaler = StandardScaler()

# Normalize the training data
X_train = scaler.fit_transform(X_train)

# Use the scaler defined above to standardize the validation data by applying the same transf
X_test = scaler.transform(X_test)
```

Training K-nearest neighbor models (18 points)

We will use the `sklearn` library to train a K-nearest neighbors (kNN) classifier. Review ch.5 and see here for more details.

Exercise 1: Learning a kNN classifier (18 points)

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
```

Exercise 1a: Evaluate the effect of the number of neighbors (6 points)

Train kNN classifiers with different number of neighbors among {1,5,25,100, length(X_train)}.

Keep all other parameters at their default values.

Report the model's accuracy over the test set.

```
for k in [1,5,25,100,len(X_train)]:
    clf = KNeighborsClassifier(n_neighbors=k)
    # Train Classifer on training set
    clf = clf.fit(X_train,y_train)
    #Predict the response for test dataset
    y_pred_test = clf.predict(X_test)

    print("# neighbors: %d"%k)
    print("test accuracy: %.2f" % accuracy_score(y_test, y_pred_test))
```

```
 # neighbors: 1
 test accuracy: 0.77
 # neighbors: 5
 test accuracy: 0.60
 # neighbors: 25
 test accuracy: 0.67
 # neighbors: 100
 test accuracy: 0.44
 # neighbors: 171
 test accuracy: 0.35
```

Explain the effect of increasing the number of neighbors on the performance observed on training and test sets.

Increasing k reduces accuracy on the training dataset because the model relies on instances in addition to the point identical to the training query data point to make its classification. Increasing k generally reduces test accuracy, even at low k values. This could be due to the class imbalance and small size of the training dataset, with over 60% (102+ samples) of the data coming from 2 classes, and the remaining (69 samples) almost evenly distributed among the remaining classes. The small sample size of minority classes might degrade the quality of the nearest neighbors set, due to a lack of representative and nearby samples, thereby degrading accuracy over minority classes (inspecting the performance by class seems to support this). This issue worsens with larger k, since more samples from the majority class contribute to the classification decision.

Exercise 1b: Evaluate the effect of a weighted kNN (6 points)

Train kNN classifiers with distance-weighting and vary the number of neighbors among {1,5,25,100,length(X_train)}.

Keep all other parameters at their default values.

Report the model's accuracy over the test set.

```
for k in [1,5,25,100,len(X_train)]:
    clf = KNeighborsClassifier(n_neighbors=k, weights='distance')
    # Train Classifer on training set
    clf = clf.fit(X_train,y_train)
    #Predict the response for test dataset
    y_pred_test = clf.predict(X_test)

    print("# neighbors: %d"%k)
    print("test accuracy: %.2f" % accuracy_score(y_test, y_pred_test))
```

```
# neighbors: 1
test accuracy: 0.77
# neighbors: 5
test accuracy: 0.70
# neighbors: 25
test accuracy: 0.67
# neighbors: 100
test accuracy: 0.72
# neighbors: 171
test accuracy: 0.67
```

Compare the effect of the number of neighbors on model performance (train and test) under the distance-weighted kNN against the uniformly weighted kNN. Explain any differences observed.

Increasing k has no impact on training performance i.e., training accuracy remains at 100% (as achieved when $k = 1$). For any point in the training set, its nearest neighbors set includes itself, for which the distance is 0, resulting in infinite weight assigned to this point. Consequently, the classification decision is effectively based only on itself*, unlike the uniformly weighted kNN for which the classification decision is equally affected by itself and the remaining neighbors.

Similar to the trends observed under the uniformly weighted kNN, increasing k reduces test accuracy, most likely due to the class imbalance and small size of the training dataset (see Ex 1b. solution).

However, the test performance is generally higher under the distance-weighted kNN than under the the uniformly weighted kNN. This is because under the distance-weighted kNN, data points distant from the query, especially points from a different class, have less impact on the classification decision. In particular, distance-weighting can help counter class imbalance effects by downweighting the contributions of training data points further away from the query that most likely belong to one of the majority classes.

*In the scikit implementation, the weight for a training point with 0 distance is set to 1 and all other weights are set to 0.

Exercise 1c: Evaluate the effect of the power parameter in the Minkowski distance metric (6 points)

Train kNN classifiers with different distance functions by varying the power parameter for the Minkowski distance among {1,2,10,100}.

Fix the number of neighbors to be 25, and use the uniformly-weighted kNN. Keep all other parameters at their default values.

Report the model's accuracy over the test set.

```
for p in [1,2,10,100]:
    clf = KNeighborsClassifier(n_neighbors=25, weights='uniform', p=p)
    # Train Classifer on training set
    clf = clf.fit(X_train,y_train)

    #Predict the response for test dataset
    y_pred_test = clf.predict(X_test)

    print("p: %d"%p)
    print("test accuracy: %.2f" % accuracy_score(y_test, y_pred_test))
```

```
p: 1
test accuracy: 0.72
p: 2
test accuracy: 0.67
p: 10
test accuracy: 0.67
p: 100
test accuracy: 0.67
```

Explain any effect observed on the model performance upon increasing the power parameter.

Increasing p generally degrades the test set performance, with $p = 1$ (Manhattan distance) yielding the best performance.

Increasing p amplifies larger differences between features relative to smaller differences between features. Consequently, the overall distance is affected more by features with larger differences, affecting the set of nearest neighbors.

For this dataset, it seems that choosing neighbors based on differences across all features rather than features with larger differences yields a more performant classifier.