Machine Learning
Lab : 1
Introduction to Machine Learning Libraries in Python

December 2022

Perform Date: December 12-16, 2022

1 Objective

To introduce different machine learning libraries available in python.

2 Description

Python is the preferred language in Machine Learning. The availability of libraries and open-source tools make it ideal choice for developing ML models. Python offers some of the best flexibilities and features to developers that not only increase their productivity but the quality of the code as well, not to mention the extensive libraries that help ease the workload. In this lab we will get introduced to some of the python libraries useful in Machine learning model development.

3 Implementation GuideLines

3.1 Linear Algebra in Python with Numpy

Numpy is one of the most used libraries in Python for arrays manipulation. It adds to Python a set of functions that allows us to operate on large multidimensional arrays with just a few lines.

[ ]:
```
import numpy as np
```

3.1.1 Defining lists and numpy arrays

[ ]:
```
x=5
print(x," TYPE: ",type(x))
x="Hello"
print(x," TYPE: ",type(x))
x=[1,2,3]
print(x," TYPE: ",type(x))
```

```
5  TYPE:  <class 'int'>
Hello  TYPE:  <class 'str'>
[1, 2, 3]  TYPE:  <class 'list'>
```

[ ]:
```
alist = [1, 2, 3, 4, 5]   # Define a python list. It looks like an np array

narray = np.array([1, 2, 3, 4]) # Define a numpy array
```

Note the difference between a Python list and a NumPy array.

[ ]:
```
print(alist)
print(narray)

print(type(alist))
print(type(narray))
```

```
[1, 2, 3, 4, 5]
[1 2 3 4]
<class 'list'>
<class 'numpy.ndarray'>
```

[ ]:
```
narray1 = np.array([1, 2, 3])
```

3.1.2 Algebraic operators on NumPy arrays vs. Python lists

Note that the '+' operator on NumPy arrays perform an element-wise addition, while the same operation on Python lists results in a list concatenation.

[ ]:
```
print(narray + narray)
print(alist + alist)
```

```
[2 4 6 8]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

[ ]:
```
print(narray + narray1) #operands needs to be of same shape
print(alist + alist)
```

[ ]:
```
alist1 = [1, 2, 3, 4]
```

[ ]:
```
print(alist + alist1)
```

```
[1, 2, 3, 4, 5, 1, 2, 3, 4]
```

It is the same as with the product operator, `*`. In the first case, we scale the vector, while in the second case, we concatenate three times the same list.

[ ]:
```
print(narray * 3)
print(alist * 3)
```

```
[ 3  6  9 12]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

3.1.3 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[ ]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

[ ]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

[ ]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

[ ]:
```
print(x.dot(y))
```

```
[[19 22]
 [43 50]]
```

[ ]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

[ ]:
```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

[ ]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

You can also use the `@` operator which is equivalent to numpy's `dot` operator.

[ ]:
```
print(v @ w)
```

```
219
```

[ ]:
```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29. 67.]
[29. 67.]
[29. 67.]
```

[ ]:
```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

```
[[19. 22.]
 [43. 50.]]
[[19. 22.]
 [43. 50.]]
[[19. 22.]
 [43. 50.]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

[ ]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

[ ]:
```
print(x)
print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

[ ]:
```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

3.1.4 Matrix or Array of Arrays

With NumPy, we have two ways to create a matrix: * Creating an array of arrays using `np.array` (recommended). * Creating a matrix using `np.matrix` (still available but might be removed soon).

[ ]:
```
npmatrix1 = np.array([narray, narray, narray]) # Matrix initialized with NumPy␣
 ↪arrays
npmatrix2 = np.array([alist, alist, alist]) # Matrix initialized with lists
npmatrix3 = np.array([narray, [1, 1, 1, 1], narray]) # Matrix initialized with␣
 ↪both types

print(npmatrix1)
```

```
print(npmatrix2)
print(npmatrix3)
```

```
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
[[1 2 3 4]
 [1 1 1 1]
 [1 2 3 4]]
```

However, when defining a matrix, be sure that all the rows contain the same number of elements. Otherwise, the linear algebra operations could lead to unexpected results.

Analyze the following two examples:

[ ]:
```
# Example 1:

okmatrix = np.array([[1, 2], [3, 4]]) # Define a 2x2 matrix
print(okmatrix) # Print okmatrix
print(okmatrix * 2) # Print a scaled version of okmatrix
```

```
[[1 2]
 [3 4]]
[[2 4]
 [6 8]]
```

[ ]:
```
# Example 2:

badmatrix = np.array([[1,2 ], [3, 4], [6,7, 8]], dtype=object) # Define a matrix.
 ↪ Note the third row contains 3 elements
print(badmatrix) # Print the malformed matrix
print(badmatrix * 2) # It is supposed to scale the whole matrix
```

```
[list([1, 2]) list([3, 4]) list([6, 7, 8])]
[list([1, 2, 1, 2]) list([3, 4, 3, 4]) list([6, 7, 8, 6, 7, 8])]
```

3.1.5 Scaling and translating matrices

Now that you know how to build correct NumPy arrays and matrices, let us see how easy it is to operate with them in Python using the regular algebraic operators like + and -.

Operations can be performed between arrays and arrays or between arrays and scalars.

[ ]:
```
# Scale by 2 and translate 1 unit the matrix
result = okmatrix * 2 + 1 # For each element in the matrix, multiply by 2 and␣
 ↪add 1
print(result)
```

```
[[3 5]
 [7 9]]
```

[ ]:
```
# Add two sum compatible matrices
result1 = okmatrix + okmatrix
print(result1)

# Subtract two sum compatible matrices. This is called the difference vector
result2 = okmatrix - okmatrix
print(result2)
```

```
[[2 4]
 [6 8]]
[[0 0]
 [0 0]]
```

The product operator * when used on arrays or matrices indicates element-wise multiplications. Do not confuse it with the dot product.

[ ]:
```
result = okmatrix * okmatrix # Multiply each element by itself
print(result)
```

```
[[ 1  4]
 [ 9 16]]
```

3.1.6 Transpose a matrix

In linear algebra, the transpose of a matrix is an operator that flips a matrix over its diagonal, i.e., the transpose operator switches the row and column indices of the matrix producing another matrix. If the original matrix dimension is n by m, the resulting transposed matrix will be m by n.

T denotes the transpose operations with NumPy matrices.

[ ]:
```
matrix3x2 = np.array([[1, 2], [3, 4], [5, 6]]) # Define a 3x2 matrix
print('Original matrix 3 x 2')
print(matrix3x2)
print('Transposed matrix 2 x 3')
print(matrix3x2.T)
```

```
Original matrix 3 x 2
[[1 2]
 [3 4]
 [5 6]]
Transposed matrix 2 x 3
[[1 3 5]
 [2 4 6]]
```

However, note that the transpose operation does not affect 1D arrays.

[ ]:
```
nparray = np.array([1, 2, 3, 4]) # Define an array
print('Original array')
print(nparray)
print('Transposed array')
print(nparray.T)
```

```
Original array
[1 2 3 4]
Transposed array
[1 2 3 4]
```

perhaps in this case you wanted to do:

[ ]:
```
nparray = np.array([[1, 2, 3, 4]]) # Define a 1 x 4 matrix. Note the 2 level of square brackets
print('Original array')
print(nparray)
print('Transposed array')
print(nparray.T)
```

```
Original array
[[1 2 3 4]]
Transposed array
[[1]
 [2]
 [3]
 [4]]
```

3.1.7 Get the norm of a nparray or matrix

In linear algebra, the norm of an n-dimensional vector $\vec{a}$ is defined as:

$$norm(\vec{a}) = ||\vec{a}|| = \sqrt{\sum_{i=1}^{n} a_i^2}$$

Calculating the norm of vector or even of a matrix is a general operation when dealing with data. Numpy has a set of functions for linear algebra in the subpackage **linalg**, including the **norm** function. Let us see how to get the norm a given array or matrix:

[ ]:
```
nparray1 = np.array([1, 2, 3, 4]) # Define an array
norm1 = np.linalg.norm(nparray1)

nparray2 = np.array([[1, 2], [3, 4]]) # Define a 2 x 2 matrix. Note the 2 level of square brackets
norm2 = np.linalg.norm(nparray2)

print(norm1)
print(norm2)
```

```
5.477225575051661
5.477225575051661
```

Note that without any other parameter, the norm function treats the matrix as being just an array of numbers. However, it is possible to get the norm by rows or by columns. The **axis** parameter controls the form of the operation: * **axis=0** means get the norm of each column * **axis=1** means get the norm of each row.

[ ]:
```
nparray2 = np.array([[1, 1], [2, 2], [3, 3]]) # Define a 3 x 2 matrix.

normByCols = np.linalg.norm(nparray2, axis=0) # Get the norm for each column.␣
 ↪Returns 2 elements
normByRows = np.linalg.norm(nparray2, axis=1) # get the norm for each row.␣
 ↪Returns 3 elements

print(normByCols)
print(normByRows)
```

```
[3.74165739 3.74165739]
[1.41421356 2.82842712 4.24264069]
```

3.1.8 The dot product between arrays: All the flavors

The dot product or scalar product or inner product between two vectors $\vec{a}$ and $\vec{a}$ of the same size is defined as:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i$$

The dot product takes two vectors and returns a single number.

[ ]:
```
nparray1 = np.array([0, 1, 2, 3]) # Define an array
nparray2 = np.array([4, 5, 6, 7]) # Define an array

flavor1 = np.dot(nparray1, nparray2) # Way-1
print(flavor1)

flavor2 = np.sum(nparray1 * nparray2) # Way-2
print(flavor2)

flavor3 = nparray1 @ nparray2         # Way-3
print(flavor3)

# As you never should do:         #Way-4
flavor4 = 0
for a, b in zip(nparray1, nparray2):
  flavor4 += a * b
print(flavor4)
```

```
38
38
```

```
38
38
```

Recommend np.dot: since it is the only method that accepts arrays and lists without problems

[ ]:
```
norm1 = np.dot(np.array([1, 2]), np.array([3, 4])) # Dot product on nparrays
norm2 = np.dot([1, 2], [3, 4]) # Dot product on python lists

print(norm1, '=', norm2 )
```

```
11 = 11
```

Finally, note that the norm is the square root of the dot product of the vector with itself.

$$norm(\vec{a}) = ||\vec{a}|| = \sqrt{\sum_{i=1}^{n} a_i^2} = \sqrt{a \cdot a}$$

3.1.9 Sums by rows or columns

Another general operation performed on matrices is the sum by rows or columns. Just as we did for the function norm, the **axis** parameter controls the form of the operation: * **axis=0** means to sum the elements of each column together. * **axis=1** means to sum the elements of each row together.

[ ]:
```
nparray2 = np.array([[1, -1], [2, -2], [3, -3]]) # Define a 3 x 2 matrix.

sumByCols = np.sum(nparray2, axis=0) # Get the sum for each column. Returns 2␣
 ↪elements
sumByRows = np.sum(nparray2, axis=1) # get the sum for each row. Returns 3␣
 ↪elements

print('Sum by columns: ')
print(sumByCols)
print('Sum by rows:')
print(sumByRows)
```

```
Sum by columns:
[ 6 -6]
Sum by rows:
[0 0 0]
```

3.1.10 Get the mean by rows or columns

As with the sums, one can get the **mean** by rows or columns using the **axis** parameter. Just remember that the mean is the sum of the elements divided by the length of the vector

$$mean(\vec{a}) = \frac{\sqrt{\sum_{i=1}^{n} a_i}}{n}$$

[ ]:
```
nparray2 = np.array([[1, -1], [2, -2], [3, -3]]) # Define a 3 x 2 matrix. Chosen
 ↪to be a matrix with 0 mean

mean = np.mean(nparray2) # Get the mean for the whole matrix
meanByCols = np.mean(nparray2, axis=0) # Get the mean for each column. Returns 2
 ↪elements
meanByRows = np.mean(nparray2, axis=1) # get the mean for each row. Returns 3
 ↪elements

print('Matrix mean: ')
print(mean)
print('Mean by columns: ')
print(meanByCols)
print('Mean by rows:')
print(meanByRows)
```

```
Matrix mean:
0.0
Mean by columns:
[ 2. -2.]
Mean by rows:
[0. 0. 0.]
```

3.1.11 Center the columns of a matrix

Centering the attributes of a data matrix is another essential preprocessing step. Centering a matrix means to remove the column mean to each element inside the column. The sum by columns of a centered matrix is always 0.

With NumPy, this process is as simple as this:

[ ]:
```
nparray2 = np.array([[1, 1], [2, 2], [3, 3]]) # Define a 3 x 2 matrix.

nparrayCentered = nparray2 - np.mean(nparray2, axis=0) # Remove the mean for
 ↪each column

print('Original matrix')
print(nparray2)
print('Centered by columns matrix')
print(nparrayCentered)

print('New mean by column')
print(nparrayCentered.mean(axis=0))
```

```
Original matrix
[[1 1]
 [2 2]
 [3 3]]
Centered by columns matrix
```

```
[[-1. -1.]
 [ 0.  0.]
 [ 1.  1.]]
New mean by column
[0. 0.]
```

Warning: This process does not apply for row centering. In such cases, consider transposing the matrix, centering by columns, and then transpose back the result.

See the example below:

[ ]:
```
nparray2 = np.array([[1, 3], [2, 4], [3, 5]]) # Define a 3 x 2 matrix.

nparrayCentered = nparray2.T - np.mean(nparray2, axis=1) # Remove the mean for␣
 ↪each row
nparrayCentered = nparrayCentered.T # Transpose back the result

print('Original matrix')
print(nparray2)
print('Centered by columns matrix')
print(nparrayCentered)

print('New mean by rows')
print(nparrayCentered.mean(axis=1))
```

```
Original matrix
[[1 3]
 [2 4]
 [3 5]]
Centered by columns matrix
[[-1.  1.]
 [-1.  1.]
 [-1.  1.]]
New mean by rows
[0. 0. 0.]
```

Exercise:

1) Create Two numpy array of size 4 X 5 and 5 X 4.
2) Randomly Initalize that array
3) Perform matrix multiplication
4) Perform elementwise matrix multiplication
5) Find mean, median of the first matrix.
6) Get the transpose of that Matrix that you created. Create a square matrix and find its determinant.
7) Obtain each row in the second column of the first array.

8) Convert Numeric entries(columns) of mtcars.csv to Mean Centered Version

3.2 NLTK Library

3.2.1 Preprocessing

In this lab, we will be exploring how to preprocess tweets for sentiment analysis. We use the NLTK package to perform a preprocessing pipeline for Twitter datasets.

[ ]:
```
import nltk                                    # Python library for NLP
from nltk.corpus import twitter_samples        # sample Twitter dataset from NLTK
import matplotlib.pyplot as plt                # library for visualization
import random                                  # pseudo-random number generator
```

3.2.2 About the Twitter dataset

The dataset contains 5000 positive tweets and 5000 negative tweets exactly. (Not a Real World Scenario !!!)

[ ]:
```
#downloads sample twitter dataset.
nltk.download('twitter_samples')
```

```
[nltk_data] Downloading package twitter_samples to /root/nltk_data...
[nltk_data]   Unzipping corpora/twitter_samples.zip.
```

[ ]:
```
True
```

We can load the text fields of the positive and negative tweets by using the module's `strings()` method like this:

[ ]:
```
# select the set of positive and negative tweets
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
```

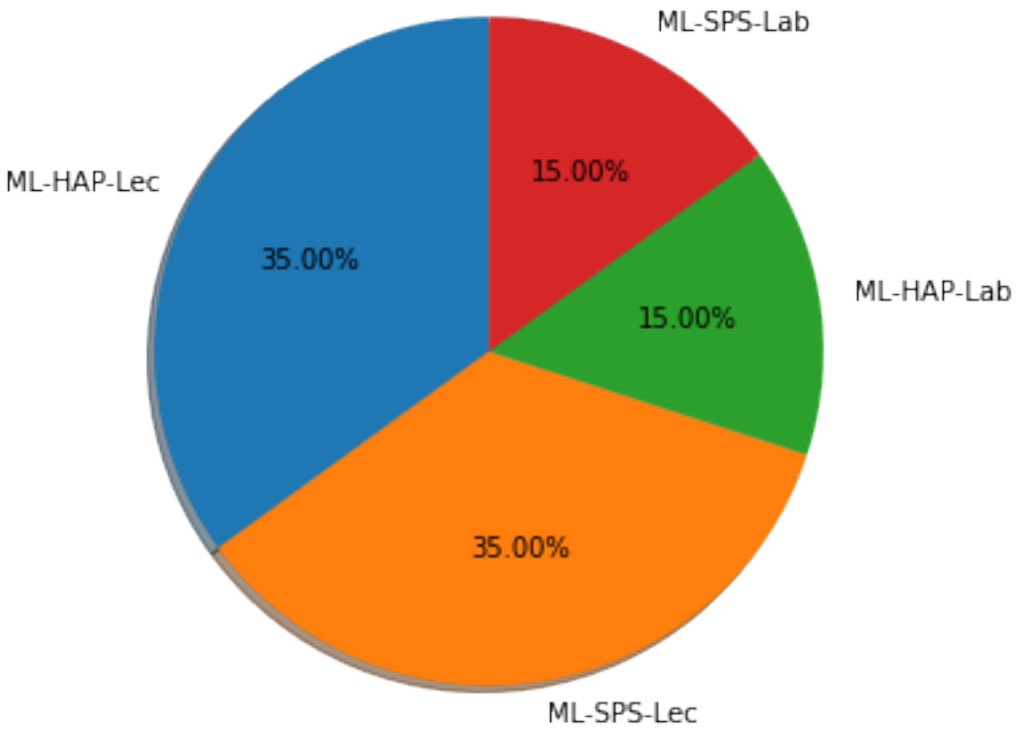
[ ]:
```
print('Number of positive tweets: ', len(all_positive_tweets))
print('Number of negative tweets: ', len(all_negative_tweets))

print('\nThe type of all_positive_tweets is: ', type(all_positive_tweets))
print('The type of a tweet entry is: ', type(all_negative_tweets[0]))
```

```
Number of positive tweets:  5000
Number of negative tweets:  5000

The type of all_positive_tweets is:  <class 'list'>
The type of a tweet entry is:  <class 'str'>
```

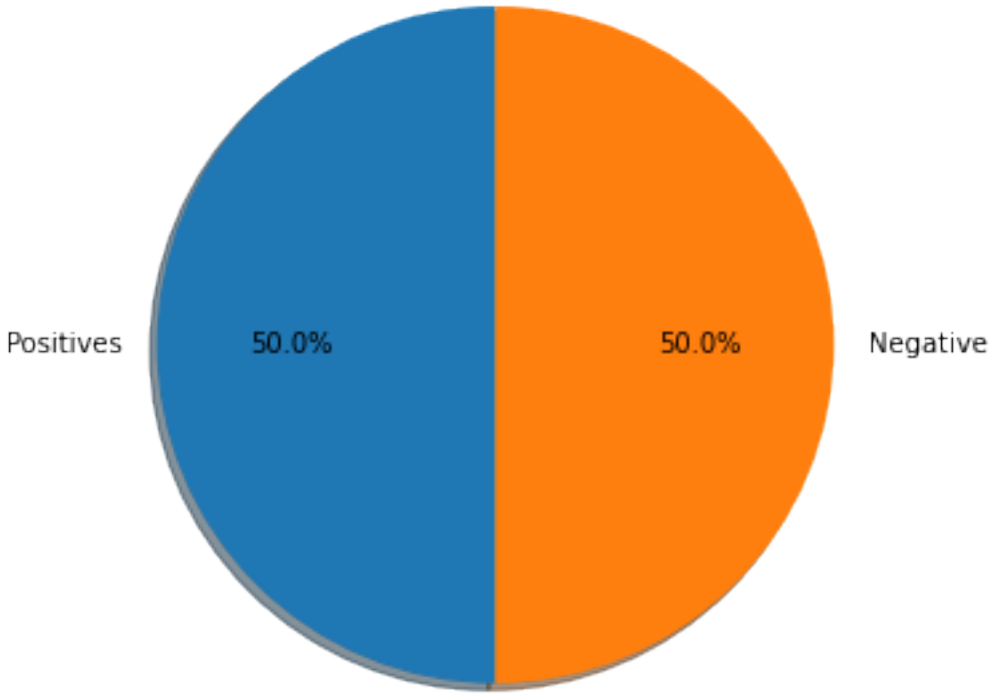
We can see that the data is stored in a list and individual tweets are stored as strings.

Let's have more visually appealing report by using Matplotlib's pyplot library.

[ ]:
```
# Declare a figure with a custom size
fig = plt.figure(figsize=(5, 5))

# labels for the classes
labels = 'ML-HAP-Lec', 'ML-SPS-Lec','ML-HAP-Lab','ML-SPS-Lab'

# Sizes for each slide
sizes = [35, 35, 15,15]

# Declare pie chart, where the slices will be ordered and plotted␣
 ↪counter-clockwise:
plt.pie(sizes, labels=labels, autopct='%.2f%%',
        shadow=True, startangle=90)

#autopct enables you to display the percent value using Python string formatting.
 ↪
#For example, if autopct='%.2f', then for each pie wedge, the format string is␣
 ↪'%.2f' and


# Equal aspect ratio ensures that pie is drawn as a circle.
plt.axis('equal')

# Display the chart
plt.show()
```



[ ]:
```
# Declare a figure with a custom size
fig = plt.figure(figsize=(5, 5))

# labels for the two classes
labels = 'Positives', 'Negative'

# Sizes for each slide
sizes = [len(all_positive_tweets), len(all_negative_tweets)]

# Declare pie chart, where the slices will be ordered and plotted␣
 ↪counter-clockwise:
plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)

# Equal aspect ratio ensures that pie is drawn as a circle.
plt.axis('equal')

# Display the chart
plt.show()
```



3.2.3 Looking at raw texts

[ ]:
```
# print positive in greeen
print('\033[92m' + all_positive_tweets[random.randint(0,5000)])

# print negative in red
print('\033[91m' + all_negative_tweets[random.randint(0,5000)])
```

```
duh emesh :p https://t.co/FtiAVocFll

@Atunci_CoV @QuetaAuthor :( rude
```

Here is a list of some common colors you can use:

Red = '\033[91m' Green = '\033[92m' Blue = '\033[94m' Cyan = '\033[96m' White = '\033[97m' Yellow = '\033[93m' Magenta = '\033[95m' Grey = '\033[90m' Black = '\033[90m' Default = '\033[99m'

One observation you may have is the presence of emoticons and URLs in many of the tweets.

3.2.4 Preprocess raw text for Sentiment analysis

Data preprocessing:

For NLP, the preprocessing steps are comprised of the following tasks:

- Tokenizing the string
- Lowercasing
- Removing stop words and punctuation
- Stemming

Let's see how we can do these to a given tweet. We will choose just one and see how this is transformed by each preprocessing step.

[ ]:
```
# Our selected sample
tweet = all_positive_tweets[2277]
print(tweet)
```

```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers #favourites
#happy #Friday off... https://t.co/3tfYom0N1i
```

Let's import a few more libraries for this purpose.

[ ]:
```
# download the stopwords from NLTK
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

[ ]:
```
True
```

[ ]:
```
import re                                # library for regular expression␣
 ↪operations
```

```
import string                                   # for string operations

from nltk.corpus import stopwords               # module for stop words that come␣
 ↪with NLTK
from nltk.stem import PorterStemmer             # module for stemming
from nltk.tokenize import TweetTokenizer    # module for tokenizing strings
from nltk.stem import          WordNetLemmatizer
```

3.2.5 Remove hyperlinks, Twitter marks and styles

Since we have a Twitter dataset, we'd like to remove some substrings commonly used on the platform like the hashtag, retweet marks, and hyperlinks. We'll use the re library to perform regular expression operations on our tweet. We'll define our search pattern and use the `sub()` method to remove matches by substituting with an empty character (i.e. `''`)

[ ]:
```
print('\033[92m' + tweet)
print('\033[94m')

# remove hyperlinks
tweet2 = re.sub(r'https?:\/\/.*[\r\n]*', '', tweet)

# remove hashtags
# only removing the hash # sign from the word
tweet2 = re.sub(r'#', '', tweet2)

print(tweet2)
```

```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers

#favourites #happy #Friday off... https://t.co/3tfYomON1i


My beautiful sunflowers on a sunny Friday morning off :) sunflowers favourites

happy Friday off...
```

3.2.6 Tokenize the string

To tokenize means to split the strings into individual words without blanks or tabs. In this same step, we will also convert each word in the string to lower case. The tokenize module from NLTK allows us to do these easily:

[ ]:
```
print()
print('\033[92m' + tweet2)
print('\033[94m')

# instantiate tokenizer class
tokenizer = TweetTokenizer(preserve_case=False)
```

```
# tokenize tweets
tweet_tokens = tokenizer.tokenize(tweet2)

print()
print('Tokenized string:')
print(tweet_tokens)
```

```
My beautiful sunflowers on a sunny Friday morning off :) sunflowers

favourites happy Friday off...


Tokenized string:

['my', 'beautiful', 'sunflowers', 'on', 'a', 'sunny', 'friday', 'morning',

'off', ':)', 'sunflowers', 'favourites', 'happy', 'friday', 'off', '...']
```

3.2.7 Remove stop words and punctuations

The next step is to remove stop words and punctuation. Stop words are words that don't add significant meaning to the text.

[ ]:
```
#Import the english stop words list from NLTK
stopwords_english = stopwords.words('english')

print('Stop words\n')
print(stopwords_english)

print('\nPunctuation\n')
print(string.punctuation)
```

```
Stop words

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're",
"you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he',
'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's",
'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what',
'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is',
'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having',
'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about',
'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above',
'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under',
'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why',
'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some',
'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now',
'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn',
```

```
"couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn',
"hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't",
'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn',
"shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn',
"wouldn't"]

Punctuation

!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

We can see that the stop words list above contains some words that could be important in some contexts.

These could be words like *i, not, between, because, won, against*.

You might need to customize the stop words list for some applications.

For our exercise, we will use the entire list.

For the punctuation, certain token like ':)' should be retained when dealing with tweets because they are used to express emotions.

[ ]:
```
print()
print('\033[92m')
print(tweet_tokens)
print('\033[94m')

tweets_clean = []

for word in tweet_tokens: # Go through every word in your tokens list
    if (word not in stopwords_english and  # remove stopwords
        word not in string.punctuation):  # remove punctuation
        tweets_clean.append(word)

print('removed stop words and punctuation:')
print(tweets_clean)
```

```
['my', 'beautiful', 'sunflowers', 'on', 'a', 'sunny', 'friday', 'morning',
'off', ':)', 'sunflowers', 'favourites', 'happy', 'friday', 'off', '...']


removed stop words and punctuation:
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers',
'favourites', 'happy', 'friday', '...']
```

3.2.8 Stemming

Stemming is the process of converting a word to its most general form, or stem. This helps in reducing the size of our vocabulary.

Consider the words: * **learn** * **learn**ing * **learn**ed * **learn**t

All these words are stemmed from its common root **learn**.

However, in some cases, the stemming process produces words that are not correct spellings of the root word. For example, **happi** and **sunni**. That's because it chooses the most common stem for related words. For example, we can look at the set of words that comprises the different forms of happy:

- **happ**y
- **happi**ness
- **happi**er

We can see that the prefix **happi** is more commonly used. We cannot choose **happ** because it is the stem of unrelated words like **happen**.

NLTK has different modules for stemming and we will be using the PorterStemmer

[ ]:
```
words =['happier','happiness','happy','studying','study','studies','meeting']
```

[ ]:
```
print()
print('\033[92m')
print(tweets_clean)
print('\033[94m')

# Instantiate stemming class
stemmer = PorterStemmer()

# Create an empty list to store the stems
tweets_stem = []

for word in tweets_clean:
    stem_word = stemmer.stem(word)  # stemming word
    tweets_stem.append(stem_word)  # append to the list

print('stemmed words:')
print(tweets_stem)
```

```
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers',
'favourites', 'happy', 'friday', '...']


stemmed words:
['beauti', 'sunflow', 'sunni', 'friday', 'morn', ':)', 'sunflow', 'favourit',
'happi', 'friday', '...']
```

[ ]:
```
print()
print('\033[92m')
print(tweets_clean)
print('\033[94m')

# Instantiate stemming class
stemmer = PorterStemmer()

# Create an empty list to store the stems
tweets_stem = []

for word in words:
    stem_word = stemmer.stem(word)  # stemming word
    tweets_stem.append(stem_word)  # append to the list

print('stemmed words:')
print(tweets_stem)
```

```
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers',
'favourites', 'happy', 'friday', '...']


stemmed words:
['happier', 'happi', 'happi', 'studi', 'studi', 'studi', 'meet']
```

[ ]:
```
import nltk
nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
```

[ ]:
```
True
```

[ ]:
```
print()
print('\033[92m')
print(tweets_clean)
print('\033[94m')

# Instantiate stemming class
wordnet_lemmatizer = WordNetLemmatizer()

# Create an empty list to store the stems
tweets_lem = []

for word in tweets_clean:
    lem_words = wordnet_lemmatizer.lemmatize(word)# stemming word
    tweets_lem.append(lem_words)  # append to the list

print('lemmatized words:')
print(tweets_lem)
```

```
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers',
'favourites', 'happy', 'friday', '...']


lemmatized words:
['beautiful', 'sunflower', 'sunny', 'friday', 'morning', ':)', 'sunflower',
'favourite', 'happy', 'friday', '...']
```

[ ]:
```
print()
print('\033[92m')
print(tweets_clean)
print('\033[94m')

# Instantiate stemming class
wordnet_lemmatizer = WordNetLemmatizer()

# Create an empty list to store the stems
tweets_lem = []

for word in words:
    lem_words = wordnet_lemmatizer.lemmatize(word,pos ="v")# stemming word
    tweets_lem.append(lem_words)  # append to the list

print('lemmatized words:')
print(tweets_lem)
```

```
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers',
'favourites', 'happy', 'friday', '...']
```

```
lemmatized words:
['happier', 'happiness', 'happy', 'study', 'study', 'study', 'meet']
```

Exercise:

Perform all of the above preprocessing tasks on any other text dataset

3.3 Pandas Library

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Mount Drive

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
data=pd.read_csv('/content/mtcars.csv')
d=pd.crosstab(index=data['cyl'],columns="count",dropna=True)
print(d)
```

```
col_0  count
cyl
4         11
6          7
8         14
```

[ ]:
```
data.head()
```

[ ]:

	Unnamed: 0	mpg	cyl	disp	hp	...	qsec	vs	am	gear	carb
0	Mazda RX4	21.0	6	160.0	110	...	16.46	0	1	4	4
1	Mazda RX4 Wag	21.0	6	160.0	110	...	17.02	0	1	4	4
2	Datsun 710	22.8	4	108.0	93	...	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	...	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	...	17.02	0	0	3	2

[5 rows x 12 columns]

[ ]:
```
data.tail()
```

[ ]:
```
        Unnamed: 0   mpg  cyl   disp   hp  ...  qsec  vs  am  gear  carb
27    Lotus Europa  30.4    4   95.1  113  ...  16.9   1   1     5     2
28  Ford Pantera L  15.8    8  351.0  264  ...  14.5   0   1     5     4
29    Ferrari Dino  19.7    6  145.0  175  ...  15.5   0   1     5     6
30   Maserati Bora  15.0    8  301.0  335  ...  14.6   0   1     5     8
31      Volvo 142E  21.4    4  121.0  109  ...  18.6   1   1     4     2

[5 rows x 12 columns]
```

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

[ ]:
```
              mpg        cyl        disp  ...         am       gear     carb
count  32.000000  32.000000   32.000000  ...  32.000000  32.000000  32.0000
mean   20.090625   6.187500  230.721875  ...   0.406250   3.687500   2.8125
std     6.026948   1.785922  123.938694  ...   0.498991   0.737804   1.6152
min    10.400000   4.000000   71.100000  ...   0.000000   3.000000   1.0000
25%    15.425000   4.000000  120.825000  ...   0.000000   3.000000   2.0000
50%    19.200000   6.000000  196.300000  ...   0.000000   4.000000   2.0000
75%    22.800000   8.000000  326.000000  ...   1.000000   4.000000   4.0000
max    33.900000   8.000000  472.000000  ...   1.000000   5.000000   8.0000

[8 rows x 11 columns]
```

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

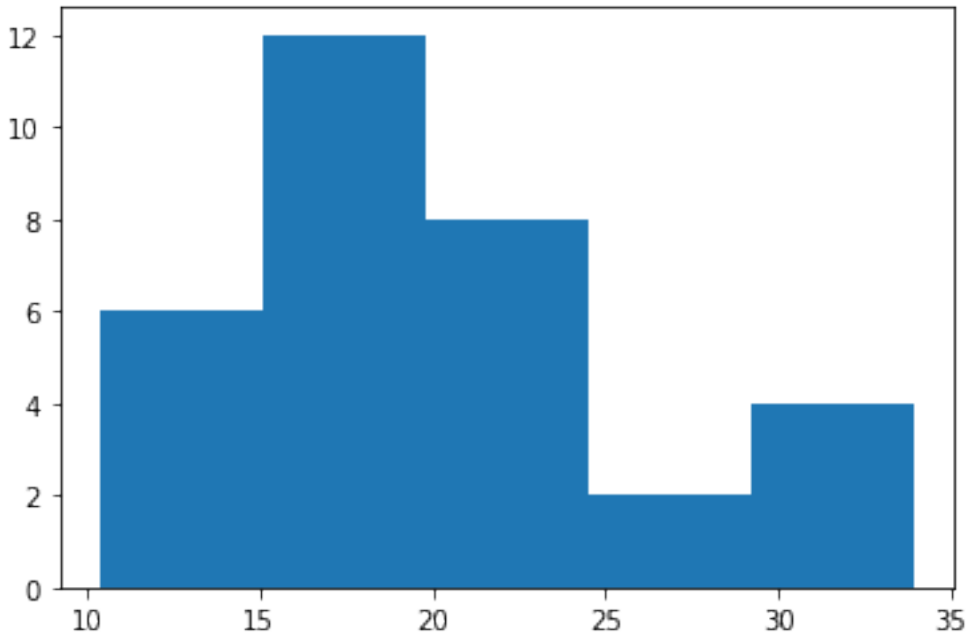
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 12 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Unnamed: 0  32 non-null     object
 1   mpg         32 non-null     float64
 2   cyl         32 non-null     int64
 3   disp        32 non-null     float64
 4   hp          32 non-null     int64
 5   drat        32 non-null     float64
 6   wt          32 non-null     float64
 7   qsec        32 non-null     float64
 8   vs          32 non-null     int64
 9   am          32 non-null     int64
 10  gear        32 non-null     int64
 11  carb        32 non-null     int64
dtypes: float64(5), int64(6), object(1)
memory usage: 3.1+ KB
```

[ ]:
```
#Count Total Null values in each column
print("Total Null Data:",data.isnull().sum())
```

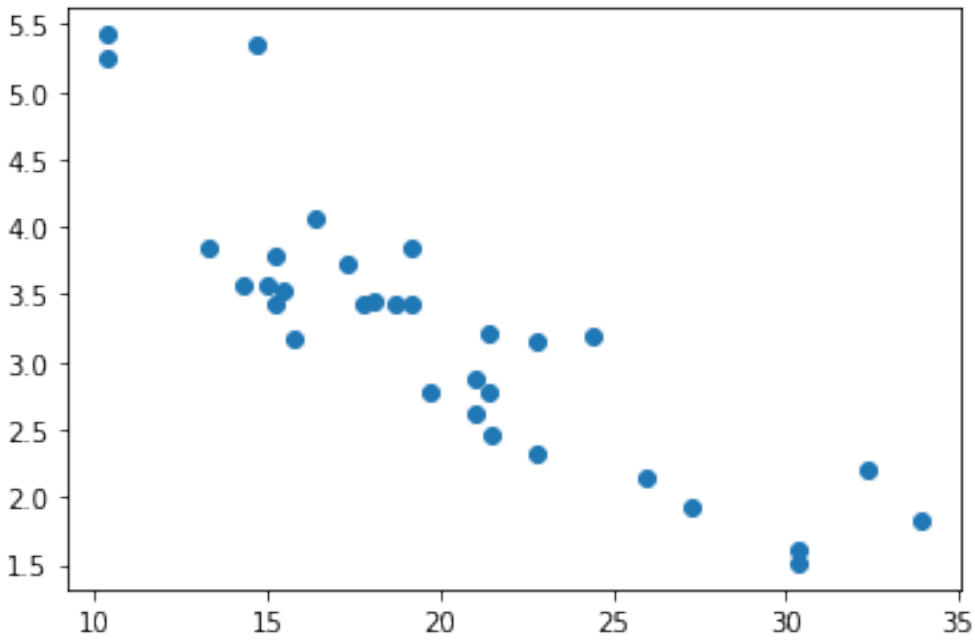
```
Total Null Data: Unnamed: 0    0
```

```
mpg           0
cyl           0
disp          0
hp            0
drat          0
wt            0
qsec          0
vs            0
am            0
gear          0
carb          0
dtype: int64
```

[ ]:
```
# Finding the Histogram
# From the given dataset 'mtcars.csv', plot a histogram to check the frequency␣
 ↪distributi on of the variable 'mpg'.
plt.hist(data['mpg'],bins=5)
plt.show()
```



[ ]:
```
#scatter plot of 'mpg' (Miles per gallon) vs 'wt' (Weight of car)
plt.scatter(data['mpg'],data['wt'])
plt.show()
```



[ ]:
```
#In the dataframe, under the variable gear count total records in each value
df=pd.DataFrame(data,columns=['gear'])
print("Count How many values:\n",df['gear'].value_counts())
```

```
Count How many values:
 3    15
4    12
5     5
Name: gear, dtype: int64
```

[ ]:
```
data.values
```

[ ]:
```
array([['Mazda RX4', 21.0, 6, 160.0, 110, 3.9, 2.62, 16.46, 0, 1, 4, 4],
       ['Mazda RX4 Wag', 21.0, 6, 160.0, 110, 3.9, 2.875, 17.02, 0, 1, 4,
        4],
       ['Datsun 710', 22.8, 4, 108.0, 93, 3.85, 2.32, 18.61, 1, 1, 4, 1],
       ['Hornet 4 Drive', 21.4, 6, 258.0, 110, 3.08, 3.215, 19.44, 1, 0,
        3, 1],
       ['Hornet Sportabout', 18.7, 8, 360.0, 175, 3.15, 3.44, 17.02, 0,
        0, 3, 2],
       ['Valiant', 18.1, 6, 225.0, 105, 2.76, 3.46, 20.22, 1, 0, 3, 1],
       ['Duster 360', 14.3, 8, 360.0, 245, 3.21, 3.57, 15.84, 0, 0, 3, 4],
       ['Merc 240D', 24.4, 4, 146.7, 62, 3.69, 3.19, 20.0, 1, 0, 4, 2],
       ['Merc 230', 22.8, 4, 140.8, 95, 3.92, 3.15, 22.9, 1, 0, 4, 2],
       ['Merc 280', 19.2, 6, 167.6, 123, 3.92, 3.44, 18.3, 1, 0, 4, 4],
       ['Merc 280C', 17.8, 6, 167.6, 123, 3.92, 3.44, 18.9, 1, 0, 4, 4],
```

```
 ['Merc 450SE', 16.4, 8, 275.8, 180, 3.07, 4.07, 17.4, 0, 0, 3, 3],
 ['Merc 450SL', 17.3, 8, 275.8, 180, 3.07, 3.73, 17.6, 0, 0, 3, 3],
 ['Merc 450SLC', 15.2, 8, 275.8, 180, 3.07, 3.78, 18.0, 0, 0, 3, 3],
 ['Cadillac Fleetwood', 10.4, 8, 472.0, 205, 2.93, 5.25, 17.98, 0,
  0, 3, 4],
 ['Lincoln Continental', 10.4, 8, 460.0, 215, 3.0,
  5.4239999999999995, 17.82, 0, 0, 3, 4],
 ['Chrysler Imperial', 14.7, 8, 440.0, 230, 3.23, 5.345, 17.42, 0,
  0, 3, 4],
 ['Fiat 128', 32.4, 4, 78.7, 66, 4.08, 2.2, 19.47, 1, 1, 4, 1],
 ['Honda Civic', 30.4, 4, 75.7, 52, 4.93, 1.615, 18.52, 1, 1, 4, 2],
 ['Toyota Corolla', 33.9, 4, 71.1, 65, 4.22, 1.835, 19.9, 1, 1, 4,
  1],
 ['Toyota Corona', 21.5, 4, 120.1, 97, 3.7, 2.465, 20.01, 1, 0, 3,
  1],
 ['Dodge Challenger', 15.5, 8, 318.0, 150, 2.76, 3.52, 16.87, 0, 0,
  3, 2],
 ['AMC Javelin', 15.2, 8, 304.0, 150, 3.15, 3.435, 17.3, 0, 0, 3,
  2],
 ['Camaro Z28', 13.3, 8, 350.0, 245, 3.73, 3.84, 15.41, 0, 0, 3, 4],
 ['Pontiac Firebird', 19.2, 8, 400.0, 175, 3.08, 3.845, 17.05, 0,
  0, 3, 2],
 ['Fiat X1-9', 27.3, 4, 79.0, 66, 4.08, 1.935, 18.9, 1, 1, 4, 1],
 ['Porsche 914-2', 26.0, 4, 120.3, 91, 4.43, 2.14, 16.7, 0, 1, 5,
  2],
 ['Lotus Europa', 30.4, 4, 95.1, 113, 3.77, 1.5130000000000001,
  16.9, 1, 1, 5, 2],
 ['Ford Pantera L', 15.8, 8, 351.0, 264, 4.22, 3.17, 14.5, 0, 1, 5,
  4],
 ['Ferrari Dino', 19.7, 6, 145.0, 175, 3.62, 2.77, 15.5, 0, 1, 5,
  6],
 ['Maserati Bora', 15.0, 8, 301.0, 335, 3.54, 3.57, 14.6, 0, 1, 5,
  8],
 ['Volvo 142E', 21.4, 4, 121.0, 109, 4.11, 2.78, 18.6, 1, 1, 4, 2]],
dtype=object)
```

Pandas library of python is very useful for the manipulation of mathematical data and is widely used in the field of machine learning. It comprises of many methods for its proper functioning. loc() and iloc() are one of those methods. These are used in slicing of data from the Pandas DataFrame. They help in the convenient selection of data from the DataFrame. They are used in filtering the data according to some conditions.

loc() : loc() is label based data selecting method which means that we have to pass the name of the row or column which we want to select.

iloc(): iloc() is a indexed based selecting method which means that we have to pass integer index in the method to select specific row/column.

[ ]:
```
data.loc[5:,'mpg']
```

[ ]:
```
5     18.1
6     14.3
7     24.4
8     22.8
9     19.2
10    17.8
11    16.4
12    17.3
13    15.2
14    10.4
15    10.4
16    14.7
17    32.4
18    30.4
19    33.9
20    21.5
21    15.5
22    15.2
23    13.3
24    19.2
25    27.3
26    26.0
27    30.4
28    15.8
29    19.7
30    15.0
31    21.4
Name: mpg, dtype: float64
```

[ ]:
```
# selecting range of rows from 2 to 5
display(data.loc[2 : 5])
```

	Unnamed: 0	mpg	cyl	disp	hp	...	qsec	vs	am	gear	carb
2	Datsun 710	22.8	4	108.0	93	...	18.61	1	1	4	1
3	Hornet 4 Drive	21.4	6	258.0	110	...	19.44	1	0	3	1
4	Hornet Sportabout	18.7	8	360.0	175	...	17.02	0	0	3	2
5	Valiant	18.1	6	225.0	105	...	20.22	1	0	3	1

[4 rows x 12 columns]

[ ]:
```
# selecting rows from 1 to 4 and columns from 2 to 4
display(data.iloc[1 : 5, 2 : 5])
```

	cyl	disp	hp
1	6	160.0	110

```
2    4  108.0   93
3    6  258.0  110
4    8  360.0  175
```

Exercise : 1)Draw Scatter Plot between SepalLengthCm and SepalWidthCm for "Iris.csv" file with proper labelling.
2) Draw Histogram of SepalLengthCm with proper labelling.
3) Plot bar chart of Species.
4) Count total null values for each column in this dataset.
5) i) Print first 5 rows of SepalLengthCm. ii) Print from 5th row and onwards and entire column of Iris.csv dataset.

3.4 Scikit Learn Library

[ ]:
```
#Scikit Learn Library
```

Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib.

[ ]:
```
#Import the relevant libraries and read the dataset
```

[ ]:
```
import numpy as np

import matplotlib as plt

from sklearn import datasets #various toy datasets

from sklearn import metrics #Check accuracy of model

from sklearn.linear_model import LogisticRegression #various supervised and␣
 ↪unsupervised learning algorithms
```

Dataset Loading

[ ]:
```
iris = datasets.load_iris()
```

Features The variables of data are called its features. They are also known as predictors, inputs or attributes.

Feature matrix It is the collection of features, in case there are more than one.

Feature Names It is the list of all the names of the features.

Response It is the output variable that basically depends upon the feature variables. They are also known as target, label or output.

Response Vector It is used to represent response column. Generally, we have just one response column.

Target Names It represent the possible values taken by a response vector.

```
[ ]: X = iris.data
     y= iris.target

     feature_names = iris.feature_names
     target_names = iris.target_names
     print("Feature names:", feature_names)
     print("Target names:", target_names)
     print("\nFirst 10 rows of X:\n", X[:10])
```

```
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']

First 10 rows of X:
 [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]]
```

To check the accuracy of our model, we can split the dataset into two pieces-a training set and a testing set. Use the training set to train the model and testing set to test the model. After that, we can evaluate how well our model did.

```
[ ]: from sklearn.model_selection import train_test_split
```

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size = 0.4, random_state=1
     )
```

Exercise: Perform the above steps on any other dataset available in Sklearn library.

References

[1] Scikit Learn Library https://scikit-learn.org/stable/
https://scikit-learn.org/stable/tutorial/index.html

[2] Numpy Library https://numpy.org/doc/stable/user/absolute_beginners.html

[3] Pandas Library https://pandas.pydata.org/docs/user_guide/index.html

[4] NLTK Library https://www.nltk.org/api/nltk.html