UNIT-4

NLTK:

Natural Language:

The method of communication with the help of which humans can speak, read, and write, is language. In other words, we humans can think, make plans, make decisions in our natural language. Here the big question is, in the era of artificial intelligence, machine learning and deep learning, can humans communicate in natural language with computers/machines? Developing NLP applications is a huge challenge for us because computers require structured data, but on the other hand, human speech is unstructured and often ambiguous in nature.

Natural language is that subfield of computer science, more specifically of AI, which enables computers/machines to understand, process and manipulate human language. In simple words, NLP is a way of machines to analyze, understand and derive meaning from human natural languages like Hindi, English, French, Dutch, etc.

Natural Language Tool Kit (NLTK)

NLTK scores very high when it comes to the ease of use and explanation of the concept. The learning curve of Python is very fast and NLTK is written in Python so NLTK is also having very good learning kit. NLTK has incorporated most of the tasks like tokenization, stemming, Lemmatization, Punctuation, Character Count, and Word count.

Installing NLTK

We can install NLTK on various OS as follows:

On Windows In order to install NLTK on Windows OS, follow the below steps:

• First, open the Windows command prompt and navigate to the location of the pip folder.

• Next, enter the following command to install NLTK:

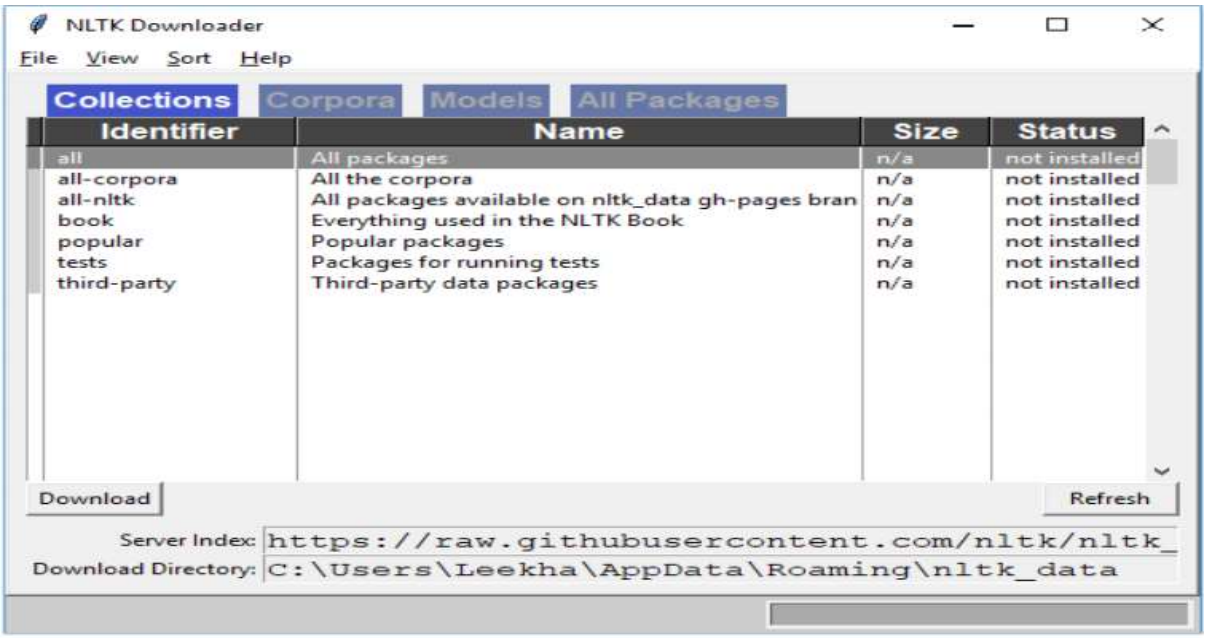
pip3 install nltk

Now we have NLTK installed on our computers but in order to use it we need to download the datasets (corpus) available in it.

Some of the important datasets available are stopwords, guntenberg, framenet_v15 and so on. With the help of following commands, we can download all the NLTK datasets:

import nltk

nltk.download()



Tokenization:

It may be defined as the process of breaking up a piece of text into smaller parts, such as sentences and words. These smaller parts are called tokens. For example, a word is a token in a sentence, and a sentence is a token in a paragraph.

nltk.tokenize is the package provided by NLTK module to achieve the process of tokenization. Tokenizing sentences into words Splitting the sentence into words or creating a list of words from a string is an essential part of every text processing activity.

Let us understand it with the help of various functions/modules provided by nltk.tokenize package. word_tokenize module word_tokenize module is used for basic word tokenization. Following example will use this module to split a sentence into words.

Example

```
import nltk

from nltk.tokenize

import word_tokenize

word_tokenize('Tutorialspoint.com provides high quality technical tutorials for free.')
```

Output

```
['Tutorialspoint.com', 'provides', 'high', 'quality', 'technical', 'tutorials', 'for', 'free', '.']
```

Tokenizing text into sentences

In this section we are going to split text/paragraph into sentences. NLTK provides sent_tokenize module for this purpose.

Why is it needed? An obvious question that came in our mind is that when we have word tokenizer then why do we need sentence tokenizer or why do we need to tokenize text into sentences. Suppose we need to count average words in sentences, how we can do this? For

accomplishing this task, we need both sentence tokenization and word tokenization.

Let us understand the difference between sentence and word tokenizer with the help of following simple example:

Example

```
import nltk from nltk.tokenize

import sent_tokenize

text = "Let us understand the difference between sentence & word tokenizer. It is going to be a simple example."

sent_tokenize(text)
```

NLTK stopwords corpus

Actually, Natural Language Tool kit comes with a stopword corpus containing word lists for many languages. Let us understand its usage with the help of the following example:

First, import the stopwords corpus from nltk.corpus package:

```
from nltk.corpus import stopwords
```

Now, we will be using stopwords from English Languages

```
english_stops = set(stopwords.words('english'))

words = ['I', 'am', 'a', 'writer']

[word for word in words if word not in english_stops]
```

Output ['I', 'writer']

Complete implementation example from nltk.corpus

```
import stopwords english_

stops = set(stopwords.words('english'))

words = ['I', 'am', 'a', 'writer']

[word for word in words if word not in english_stops]
```

Output ['I', 'writer']

What is Stemming?

Stemming is a technique used to extract the base form of the words by removing affixes from them. It is just like cutting down the branches of a tree to its stems. For example, the stem of the words eating, eats, eaten is eat.

NLTK has PorterStemmer class with the help of which we can easily implement Porter Stemmer algorithms for the word we want to stem. This class knows several regular word forms and suffixes with the help of which it can transform the input word to a final stem. The resulting stem is often a shorter word having the same root meaning. Let us see an example: First, we need to import the natural language toolkit(nltk).

import nltk

Now, import the PorterStemmer class to implement the Porter Stemmer algorithm.

from nltk.stem import PorterStemmer

Next, create an instance of Porter Stemmer class as follows:

word_stemmer = PorterStemmer()

Now, input the word you want to stem.

word_stemmer.stem('writing')

Output

'write’

What is POS tagging?

Tagging, a kind of classification, is the automatic assignment of the description of the tokens. We call the descriptors ‘tag’, which

represents one of the parts of speech (nouns, verb, adverbs, adjectives, pronouns, conjunction and their sub-categories), semantic information and so on.

On the other hand, if we talk about Part-of-Speech (POS) tagging, it may be defined as the process of converting a sentence in the form of a list of words, into a list of tuples. Here, the tuples are in the form of (word, tag). We can also call POS tagging a process of assigning one of the parts of speech to the given word.

Sr. No.	Tag	Description
1.	NNP	Proper noun, singular
2.	NNPS	Proper noun, plural
3.	PDT	Pre determiner
4.	POS	Possessive ending
5.	PRP	Personal pronoun
6.	PRP$	Possessive pronoun
7.	RB	Adverb
8.	RBR	Adverb, comparative
9.	RBS	Adverb, superlative
10.	RP	Particle
11.	SYM	Symbol (mathematical or scientific)
12.	TO	to
13.	UH	Interjection
14.	VB	Verb, base form
15.	VBD	Verb, past tense
16.	VBG	Verb, gerund/present participle
17.	VBN	Verb, past
18.	WP	Wh-pronoun

```
import nltk from nltk
 import word_tokenize
sentence = "I am going to school"
print (nltk.pos_tag(word_tokenize(sentence)))
```

Output

```
[('I', 'PRP'), ('am', 'VBP'), ('going', 'VBG'), ('to', 'TO'), ('school', 'NN')]
```

Scikit-learn:

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Components of scikit-learn:

Scikit-learn comes loaded with a lot of features. Here are a few of them to help you understand the spread:

- **Supervised learning algorithms:** Think of any supervised machine learning algorithm you might have heard about and there is a very high chance that it is part of scikit-learn. Starting from Generalized linear models (e.g Linear Regression), Support Vector Machines (SVM), Decision Trees to Bayesian methods – all of them are part of scikit-learn toolbox. The spread of machine learning algorithms is one of the big reasons for the high usage of scikit-learn. I started using scikit to solve supervised learning problems and would recommend that to people new to scikit / machine learning as well.
- **Cross-validation:** There are various methods to check the accuracy of supervised models on unseen data using sklearn.
- **Unsupervised learning algorithms:** Again there is a large spread of machine learning algorithms in the offering – starting from clustering, factor analysis, principal component analysis to unsupervised neural networks.
- **Various toy datasets:** This came in handy while learning scikit-learn. I had learned SAS using various academic datasets (e.g. IRIS dataset, Boston House prices dataset). Having them handy while learning a new library helped a lot.
- **Feature extraction:** Scikit-learn for extracting features from images and text (e.g. Bag of words)

Machine learning is the process to automatically extract knowledge from data, usually with the goal of making predictions on new, unseen

data. A classical example is a spam filter, for which the user keeps labeling incoming mails as either spam or not spam. A machine learning algorithm then “learns” what distinguishes spam from normal emails, and can predict for new emails whether they are spam or not.

Central to machine learning is the concept of making decision automatically from data, without the user specifying explicit rules how this decision should be made.

For the case of emails, the user doesn’t provide a list of words or characteristics that make an email spam. Instead, the user provides examples of spam and non-spam emails.

The second central concept is generalization. The goal of a machine learning algorithm is to predict on new, previously unseen data. We are not interested in marking an email as spam or not, that the human already labeled. Instead, we want to make the users life easier by making an automatic decision for new incoming mail.

The data is presented to the algorithm usually as an array of numbers. Each data point (also known as sample) that we want to either learn from or make a decision on is represented as a list of numbers, called features, that reflect properties of this point.

There are two kinds of machine learning we will talk about today: Supervised learning and unsupervised learning

Supervised Learning: Classification and regression

In Supervised Learning, we have a dataset consisting of both input features and a desired output, such as in the spam / no-spam example. The task is to construct a model (or program) which is able to predict the desired output of an unseen object given the set of features.

Some more complicated examples are:

- given a multicolor image of an object through a telescope, determine whether that object is a star, a quasar, or a galaxy.
- given a photograph of a person, identify the person in the photo.
- given a list of movies a person has watched and their personal rating of the movie, recommend a list of movies they would like.
- given a persons age, education and position, infer their salary

What these tasks have in common is that there is one or more unknown quantities associated with the object which needs to be determined from other observed quantities.

Supervised learning is further broken down into two categories, classification and regression. In classification, the label is discrete, such as “spam” or “no spam”. In other words, it provides a clear-cut distinction between categories. In regression, the label is continuous, that is a float output. For example, in astronomy, the task of determining whether an object is a star, a galaxy, or a quasar is a classification problem: the label is from three distinct categories. On the other hand, we might wish to estimate the age of an object based on such observations: this would be a regression problem, because the label (age) is a continuous quantity.

In supervised learning, there is always a distinction between a training set for which the desired outcome is given, and a test set for which the desired outcome needs to be inferred. More about that later.

Step 1: Import the relevant libraries and read the dataset

```
import numpy as np

import matplotlib as plt

from sklearn import datasets

from sklearn import metrics

from sklearn.linear_model import LogisticRegression
```

We have imported all the libraries. Next, we read the dataset:

```
dataset = datasets.load_iris()
```

Step 3: Build a logistic regression model on the dataset and making predictions

```
model.fit(dataset.data, dataset.target)

expected = dataset.target

predicted = model.predict(dataset.data)
```

Step 4: Print confusion matrix

```
print(metrics.classification_report(expected, predicted))

print(metrics.confusion_matrix(expected, predicted))
```

scikit-learn is an open source Python library that implements a range of machine learning, pre-processing, cross-validation and visualization algorithms using a unified interface.

Important features of scikit-learn:

- Simple and efficient tools for data mining and data analysis. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means, etc.
- Accessible to everybody and reusable in various contexts.
- Built on the top of NumPy, SciPy, and matplotlib.
- Open source, commercially usable – BSD license.

In this article, we are going to see how we can easily build a machine learning model using scikit-learn.

Installation:

Scikit-learn requires:

- NumPy
- SciPy as its dependencies.

Before installing scikit-learn, ensure that you have NumPy and SciPy installed. Once you have a working installation of NumPy and SciPy, the easiest way to install scikit-learn is using pip:

pip install -U scikit-learn
Let us get started with the modeling process now.

Step 1: Load a dataset
A dataset is nothing but a collection of data. A dataset generally has two main components:

- **Features**: (also known as predictors, inputs, or attributes) they are simply the variables of our data. They can be more than one and hence represented by a **feature matrix** ('X' is a common notation to represent feature matrix). A list of all the feature names is termed as **feature names**.
- **Response**: (also known as the target, label, or output) This is the output variable depending on the feature variables. We generally have a single response column and it is represented by a **response vector** ('y' is a common notation to represent response vector). All the possible values taken by a response vector is termed as **target names**.

Loading exemplar dataset: scikit-learn comes loaded with a few example datasets like the iris and digits datasets for classification and the boston house price dataset for regression.

Given below is an example of how one can load an exemplar dataset:

```
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target

# store the feature and target names
feature_names = iris.feature_names
target_names = iris.target_names

# printing features and target names of our dataset
print("Feature names:", feature_names)
print("Target names:", target_names)
```

```
# X and y are numpy arrays
print("\nType of X is:", type(X))

# printing first 5 input rows
print("\nFirst 5 rows of X:\n", X[:5])
```

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


Type of X is:


First 5 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]]
```

In pandas, important data types are:

Series: Series is a one-dimensional labeled array capable of holding any data type.

DataFrame: It is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object.
Note: The CSV file used in example below can be downloaded from here: weather.csv

```
import pandas as pd

# reading csv file

data = pd.read_csv('weather.csv')

# shape of dataset

print("Shape:", data.shape)

# column names

print("\nFeatures:", data.columns)

# storing the feature matrix (X) and response vector (y)

X = data[data.columns[:-1]]

y = data[data.columns[-1]]

# printing first 5 rows of feature matrix

print("\nFeature matrix:\n", X.head())

# printing first 5 values of response vector

print("\nResponse vector:\n", y.head())
```

Output:

```
Shape: (14, 5)

Features: Index([u'Outlook', u'Temperature', u'Humidity',
                 u'Windy', u'Play'], dtype='object')
```

Feature matrix:

	Outlook	Temperature	Humidity	Windy
0	overcast	hot	high	False
1	overcast	cool	normal	True
2	overcast	mild	high	True
3	overcast	hot	normal	False
4	rainy	mild	high	False

Response vector:

0 yes

1 yes

2 yes

3 yes

4 yes

Name: Play, dtype: object

Step 2: Splitting the dataset

One important aspect of all machine learning models is to determine their accuracy. Now, in order to determine their accuracy, one can train the model using the given dataset and then predict the response values for the same dataset using that model and hence, find the accuracyof the model.But this method has several flaws in it, like:

- Goal is to estimate likely performance of a model on an **out-of-sample** data.
- Maximizing training accuracy rewards overly complex models that won't necessarily generalize our model.
- Unnecessarily complex models may over-fit the training data.

A better option is to split our data into two parts: first one for training our machine learning model, and second one for testing our model.

To summarize:

- Split the dataset into two pieces: a training set and a testing set.
- Train the model on the training set.
- Test the model on the testing set, and evaluate how well our model did.

Advantages of train/test split:

- Model can be trained and tested on different data than the one used for training.
- Response values are known for the test dataset, hence predictions can be evaluated
- Testing accuracy is a better estimate than training accuracy of out-of-sample performance.

```
# load the iris dataset as an example

from sklearn.datasets import load_iris

iris = load_iris()

# store the feature matrix (X) and response vector (y)

X = iris.data

y = iris.target

# splitting X and y into training and testing sets

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)

# printing the shapes of the new X objects

print(X_train.shape)

print(X_test.shape)

# printing the shapes of the new y objects

print(y_train.shape)

print(y_test.shape)
```

Output:

(90L, 4L)

(60L, 4L)

(90L,)

(60L,)

The **train_test_split** function takes several arguments which are explained below:

- **X, y**: These are the feature matrix and response vector which need to be splitted.
- **test_size**: It is the ratio of test data to the given data. For example, setting test_size = 0.4 for 150 rows of X produces test data of 150 x 0.4 = 60 rows.
- **random_state**: If you use random_state = some_number, then you can guarantee that your split will be always the same. This is useful if you want reproducible results, for example in testing for consistency in the documentation (so that everybody can see the same numbers).

Step 3: Training the model

Now, its time to train some prediction-model using our dataset. Scikit-learn provides a wide range of machine learning algorithms which have a unified/consistent interface for fitting, predicting accuracy, etc.

The example given below uses KNN (K nearest neighbors) classifier.
Note: We will not go into the details of how the algorithm works as we are interested in understanding its implementation only.
Now, consider the example below:

```
# load the iris dataset as an example

from sklearn.datasets import load_iris

iris = load_iris()

# store the feature matrix (X) and response vector (y)

X = iris.data

y = iris.target

# splitting X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
random_state=1)

# training the model on training set

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=3)

knn.fit(X_train, y_train)
```

```
# making predictions on the testing set
y_pred = knn.predict(X_test)
# comparing actual response values (y_test) with predicted response values (y_pr
from sklearn import metrics
print("kNN model accuracy:", metrics.accuracy_score(y_test, y_pred))
# making prediction for out of sample data
sample = [[3, 5, 4, 2], [2, 3, 5, 4]]
preds = knn.predict(sample)
pred_species = [iris.target_names[p] for p in preds]
print("Predictions:", pred_species)
```

Output:

kNN model accuracy: 0.983333333333

Predictions:

['versicolor', 'virginica']

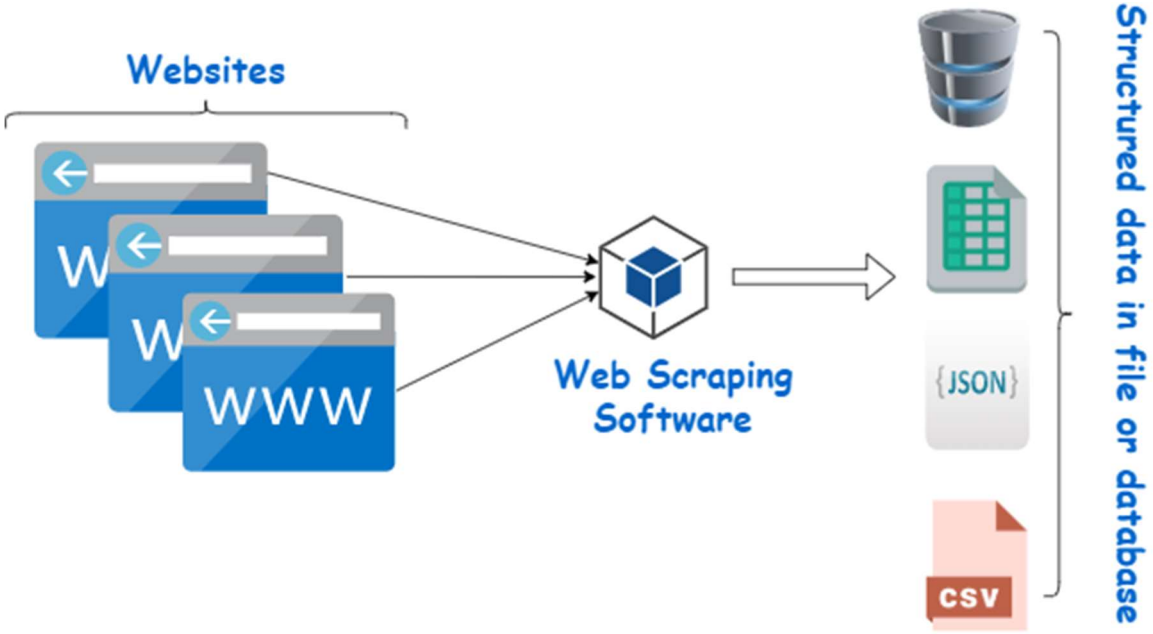
What Is Web Scraping?

The automated gathering of data from the Internet is nearly as old as the Internet itself. Although web scraping is not a new term, in years past the practice has been more commonly known as screen scraping, data mining, web harvesting, or similar variations.

Web Scraping (also termed Screen Scraping, Web Data Extraction, Web Harvesting etc.) is a technique employed to extract large

amounts of data from websites whereby the data is extracted and saved to a local file in your computer or to a database in table (spreadsheet) format.

Web Scraping is the technique of automating this process, so that instead of manually copying the data from websites, the Web Scraping software will perform the same task within a fraction of the time.



In theory, web scraping is the practice of gathering data through any means other than a program interacting with an API. This is most commonly accomplished by writing an automated program that queries a web server, requests data, and then parses that data to extract needed information.

Why Web Scraping?

If the only way you access the Internet is through a browser, you're missing out on a huge range of possibilities. Although browsers are

handy for executing JavaScript, displaying images, and arranging objects in a more human-readable format.

Web scrapers are excellent at gathering and processing large amounts of data. Rather than viewing one page at a time through the narrow window of a monitor, you can view databases spanning thousands or even millions of pages at once. In addition, web scrapers can go places that traditional search engines cannot. A Google search for “cheapest flights to Boston” will result in a slew of advertisements and popular flight search sites. Google only knows what these websites say on their content pages, not the exact results of various queries entered into a flight search application. However, a well-developed web scraper can chart the cost of a flight to Boston over time, across a variety of websites, and tell you the best time to buy your ticket.

Steps in Web scrapers:

1. Identify the target website
2. Collect URLs of the pages where you want to extract data from
3. Make a request to these URLs to get the HTML of the page
4. Use locators to find the data in the HTML
5. Save the data in a JSON or CSV file or some other structured format

Applications of Web Scraping

Price intelligence is the biggest use case for web scraping. Extracting product and pricing information from e-commerce websites, then turning it into intelligence is an important part of modern e-commerce companies that want to make better pricing/marketing decisions based on data.

How web pricing data and price intelligence can be useful:

- Dynamic pricing
- Revenue optimization

- Competitor monitoring
- Product trend monitoring
- Brand and MAP compliance

Running BeautifulSoup

The most commonly used object in the BeautifulSoup library is, appropriately, the BeautifulSoup object.

```
from urllib.request import urlopen

from bs4 import BeautifulSoup

html =
urlopen("http://www.pythonscraping.com/exercises/exercise1.html")

bsObj = BeautifulSoup(html.read());

print(bsObj.h1)
```

The output is:

<h1> An Interesting Title </h1>

As in the example before, we are importing the urlopen library and calling html.read() in order to get the HTMLcontent of the page. This HTMLcontent is then transformed into a BeautifulSoup object, with the following structure:

- **html** → *<html><head>...</head><body>...</body></html>*
  head → *<head><title>A Useful Page<title></head>*
  - **title** → *<title>A Useful Page</title>*

  body → *<body><h1>An Int...</h1> Lorem ip... </body>*
  - **h1** → *<h1>An Interesting Title</h1>*
  - **div** → * Lorem Ipsum dolor... *

find() and findAll() with BeautifulSoup

BeautifulSoup's find() and findAll() are the two functions you will likely use the most. With them, you can easily filter HTMLpages to

find lists of desired tags, or a single tag, based on their various attributes.

The two functions are extremely similar, as evidenced by their definitions in the BeautifulSoup documentation:

findAll(tag, attributes, recursive, text, limit, keywords)

find(tag, attributes, recursive, text, keywords)

In all likelihood, 95% of the time you will find yourself only needing to use the first two arguments: tag and attributes. However, let's take a look at all of the arguments in greater detail.

The tag argument is one that we've seen before — you can pass a string name of a tag or even a Python list of string tag names. For example, the following will return a list of all the header tags in a document:

.findAll({"h1","h2","h3","h4","h5","h6"}

The attributes argument takes a Python dictionary of attributes and matches tags that contain any one of those attributes. For example, the following function would return both the green and red span tags in the HTMLdocument: .

findAll("span", {"class":"green", "class":"red"})

What is Data Cleaning?

- Data cleaning, also referred to as data cleansing, is the process of finding and correcting inaccurate data from a particular data set or data source .
- The primary goal is to identify and remove inconsistencies without deleting the necessary data to produce insights.
- It's important to remove these inconsistencies in order to increase the validity of the data set.
- Cleaning encompasses a multitude of activities such as identifying duplicate records, filling empty fields and fixing structural errors.

- These tasks are crucial for ensuring the quality of data is accurate, complete, and consistent.

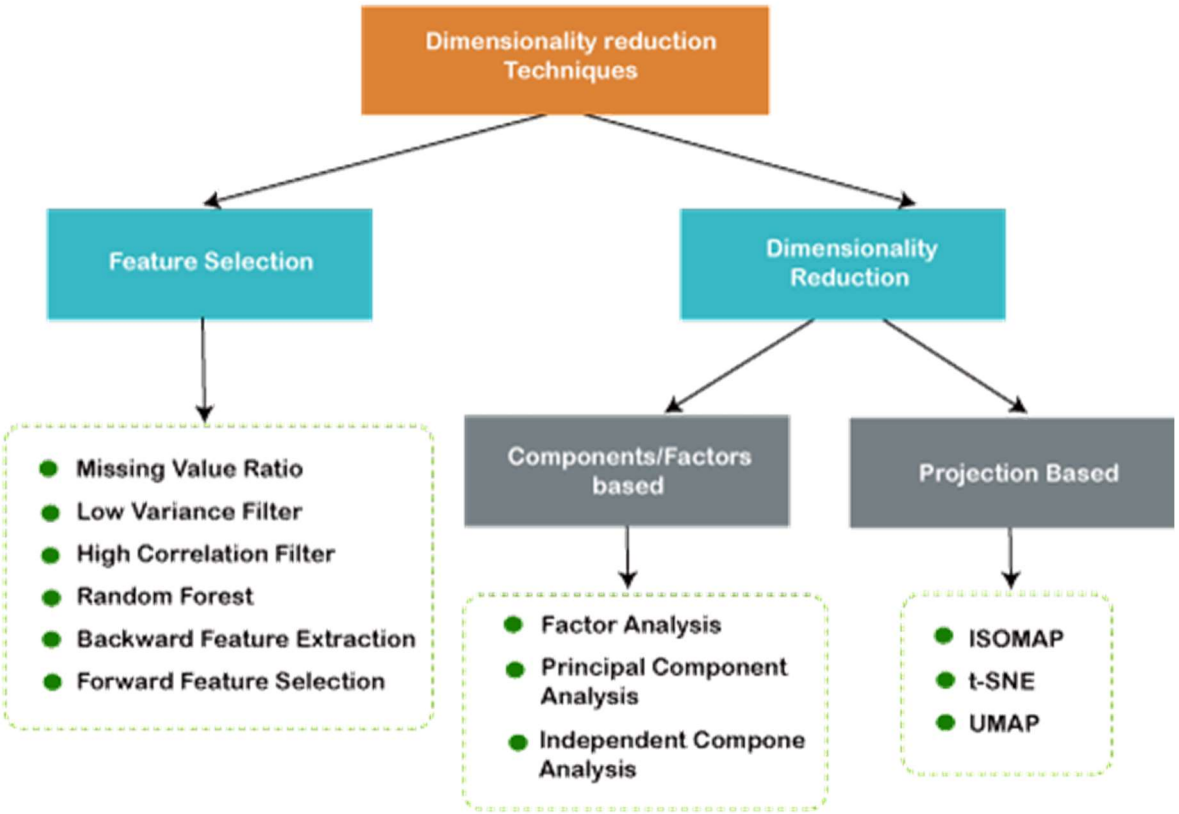
Dimensionality Reduction:

The number of input features, variables or columns present in a given dataset is known as dimensionality and the process of reducing these features is called as dimensionality reduction.

A dataset contains huge number of input features in various cases, which makes the predictive modeling task more complicated. Because it is difficult to visualize or make predictions for the training dataset with a high number of features for such cases, dimensionality reduction techniques are required to use.

Dimensionality reduction technique can be defined as " It is a way of converting the higher dimensions dataset into lesser dimensions dataset ensuring that it provides similar information. These techniques are widely used in machine learning for obtaining a better fit predictive model while solving the classification and regression.

It is commonly used in the fields that deal with high dimensional data such as speech recognition, signal processing, bioinformatics.



The Curse of Dimensionality

- Handling the high-dimensional data is very difficult in practice, commonly known as the curse of dimensionality.
- If the dimensionality of the input dataset increases, any machine learning algorithm and model becomes more complex.
- As the number of features increases, the number of samples also gets increased proportionally, and the chance of overfitting also increases.
- If the machine learning model is trained on high-dimensional data, it becomes overfitted and results in poor performance.
- Hence, it is often required to reduce the number of features, which can be done with dimensionality reduction.

Benefits of Dimensionality Reduction:

1. By reducing the dimensions of the feature, the space required to store the dataset also gets reduced.

2. Less Computation training time is required for reduced dimensions of features

3. Reduced dimensions of features of the dataset help in visualizing the data quickly.

Disadvantage of dimensionality Reduction:

1. Some data may be lost due to dimensionality reduction

2. In PCA, sometimes the principle components required considered to be unknown.