Classes in Scikit-learn

- Understanding how classes work is an important prerequisite for being able to use the Scikit-learn package appropriately.
- Scikit-learn is the package for machine learning and data science experimentation favored by most data scientists.
- It contains wide range of well-established learning algorithms, error functions and testing procedures.
- Install : conda install scikit-learn
- There are four class types covering all the basic machine-learning functionalities,
  - Classifying
  - Regressing
  - Grouping by clusters
  - Transforming data
- Even though each base class has specific methods and attributes, the core functionalities for data processing and machine learning are guaranteed by one or more series of methods and attributes called interfaces.
- The interfaces provide a uniform Application Programming Interface (API) to enforce similarity of methods and attributes between all the different algorithms present in the package. There are four Scikit-learn object-based interfaces:
  - **estimator**: For fitting parameters, learning them from data, according to the algorithm.
  - **predictor**: For generating predictions from the fitted parameters
  - **transformer**: For transforming data, implementing the fitted parameters
  - **model**: For reporting goodness of fit or other score measures
- The package groups the algorithms built on base classes and one or more object interfaces into modules, each module displaying a specialization in a particular type of machine-learning solution. For example, the "**linear_model**" module is for linear modeling, and "**metrics**" is for score and loss measure.

Supervised V/S Unsupervised Learning

Supervised Learning	Unsupervised Learning
Supervised learning algorithms are trained using labeled data.	Unsupervised learning algorithms are trained using unlabeled data.

Supervised learning model takes direct feedback to check if it is predicting correct output or not.	Unsupervised learning model does not take any feedback.
The goal of supervised learning is to train the model so that it can predict the output when it is given new data.	The goal of unsupervised learning is to find the hidden patterns and useful insights from the unknown dataset.
Supervised learning can be categorized in **Classification** and **Regression** problems .	Unsupervised Learning can be classified in **Clustering** and **Associations** problems .
Supervised learning model produces an accurate result.	Unsupervised learning model may give less accurate result as compared to supervised learning.
It includes various algorithms such as Linear Regression, Logistic Regression, Support Vector Machine, Multi-class Classification, Decision tree, Bayesian Logic, etc.	It includes various algorithms such as Clustering, KNN, and Apriori algorithm.

Hashing Trick

- Most Machine Learning algorithms uses numeric inputs, if our data contains text instead we need to convert those text into **numeric values** first, this can be done using hashing tricks.
- For Example,

Id	Salary	Gender
1	10000	Male
2	15000	Female
3	12000	Male
4	11000	Female
5	20000	Male

Id	Salary	Gender	Gender_Num
1	10000	Male	1
2	15000	Female	0
3	12000	Male	1
4	11000	Female	0
5	20000	Male	1

- When dealing with text, one of the most useful solutions provided by the Scikit-learn package is the hashing trick.
- Example:

Actual Text

```
darshan is the best engineering college in rajkot
rajkot is famous for engineering
college is located in rajkot morbi road
```

Numeric Representation

```
1 2 3 4 5 6 7 8
8 2 9 10 5
6 2 11 7 8 12 13
```

Sparse Matrix

darshan	is	the	best	engineering	college	in	rajkot	famous	for	located	morbi	road
1	1	1	1	1	1	1	1	0	0	0	0	0
0	1	0	0	1	0	0	1	1	1	0	0	0
0	1	0	0	0	1	1	1	0	0	1	1	1

Example

```python
a = ["darshan is the best engineering college in rajkot","rajkot
is famous for engineering","college is located in rajkot morbi
road"]
from sklearn.feature_extraction.text import *
countVector = CountVectorizer()
X_train_counts = countVector.fit_transform(a)
print(countVector.vocabulary_)
print(X_train_counts.toarray())
```

Output:

```
{'darshan': 2,
 'is': 7,
 'the': 12,
 'best': 0,
 'engineering': 3,
 'college': 1,
 'in': 6,
 'rajkot': 10,
 'famous': 4,
 'for': 5,
 'located': 8,
 'morbi': 9,
 'road': 11}
array([[1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1],
       [0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0]], dtype=int64)
```

Considering Timing and Performance

- Managing the best use of machine resources is indeed an art, the art of optimization, and it requires time to master.
- However, we can start immediately becoming proficient in it by doing some accurate speed measurement and realizing what your problems really are.
- Profiling the time that operations require, measuring how much memory adding more data takes, or performing
- a transformation on your data can help you to spot the bottlenecks in your code and start looking for alternative solutions.
- IPython is the perfect environment for experimenting, tweaking, and improving your code.
- Working on blocks of code, recording the results and outputs, and writing additional notes and comments will help your data science solutions take shape in a controlled and reproducible way.

Benchmarking with timeit

- We can find the time taken to execute a statement or a cell in a Jupyter Notebook with the help of timeit magic command.
- This command can be used both as a line and cell magic:
    - In **line mode** you can time a single-line statement (though multiple ones can be chained with using semicolons).
      Syntax: %timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o]
    - In **cell mode**, the statement in the first line is used as setup code (executed but not timed) and the body of the cell is timed. The cell body has access to any variables created in the setup code.
      Syntax: %%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o]
      Here, **-n** flag represents the number of loops and –**r** flag represents the number of repeats
      Example:

```
%%timeit -n 1000 -r 7

for i in range(2,1000) :
    listPrime  = []
    flag = 0
    for j in range(2,i) :
        if i % j == 0 :
            flag = 1
            break
    if flag == 0 :
        listPrime.append(i)
#print(listPrime)
```

Output:

```
5.17 ms ± 705 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Memory Profiler

- This is a python module for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption for python programs.
- **Installation**: pip install memory_profiler
- **Usage**:
  - First Load the profiler by writing %load_ext memory_profiler in the notebook
  - Then write %memit on each statement you want to monitor memory.

  Example:

```
%load_ext memory_profiler
from sklearn.feature_extraction.text import CountVectorizer
%memit countVector =
        CountVectorizer(stop_words='english',analyzer='word')
```

Output:

```
peak memory: 85.61 MiB, increment: 0.02 MiB
```

Running in parallel

- Most computers today are multicore (two or more processors in a single package), some with multiple physical CPUs.
- One of the most important limitations of Python is that it uses a single core by default. (It was created in a time when single cores were the norm.)
- Data science projects require quite a lot of computations.
- Part of the scientific aspect of data science relies on repeated tests and experiments on different data matrices.
- Also the data size may be huge, which will take lots of processing power.

- Using more CPU cores accelerates a computation by a factor that almost matches the number of cores.
- For example, having four cores would mean working at best four times faster, but practically we will not have four times faster processing as assigning the different cores to different processes will take some amount of time.
- So parallel processing will work best with huge datasets or where extreme processing power is required.
- In Scikit-learn library we need not to do any special programming in order to do the parallel (multiprocessing) processing, we can simply use **n_jobs** parameter to do so.
- If we set number grater than 1 in the **n_jobs** it will uses that much of the processor, if we set -1 to **n_jobs** it will use all available processor from the system.
- Let's see the Example how we can achieve parallel processing in Scikit-learn library.
- Example (Without parallel processing)

```
%%timeit -n 1 -r 1
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_info
rmative=15, n_redundant=5, random_state=3)
# define the model
model = RandomForestClassifier(n_estimators=500, n_jobs=1)
```

Output:

```
25.2 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

- Example (With parallel processing)

```
%%timeit -n 1 -r 1
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_info
rmative=15, n_redundant=5, random_state=3)
# define the model
model = RandomForestClassifier(n_estimators=500, n_jobs=-1)
```

Output:

```
12.8 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Exploratory Data Analysis (EDA)

- Exploratory Data Analysis (EDA) refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations.

- EDA was developed at Bell Labs by John Tukey, a mathematician and statistician who wanted to promote more questions and actions on data based on the data itself.
- In one of his famous writings, Tukey said:
  "*The only way humans can do **BETTER** than computers is to take a chance of doing **WORSE** than them.*"
- Above statement explains why, as a data scientist, your role and tools aren't limited to automatic learning algorithms but also to manual and creative exploratory tasks.
- Computers are unbeatable at optimizing, but humans are strong at discovery by taking unexpected routes and trying unlikely but very effective solutions.
- With EDA we can,
  - Describe data
  - Closely explore data distributions
  - Understand the relationships between variables
  - Notice unusual or unexpected situations
  - Place the data into groups
  - Notice unexpected patterns within the group
  - Take note of group differences

Measuring Central Tendency

We can use many inbuilt pandas functions in order to find central tendency of the numerical data, Some of the functions are,

- df.mean()
- df.median()
- df.std()
- df.max()
- df.min()
- df.quantile(np.array([0,0.25,0.5,0.75,1]))

Normality

We can define normality of the data using below measures,

- **Skewness** defines the asymmetry of data with respect to the mean.
  - It will be negative if the left tail is too long.
  - It will be positive if the right tail is too long.
  - It will be zero if left and right tail are same.

- **Kurtosis** shows whether the data distribution, especially the peak and the tails, are of the right shape.
  - It will be positive if distribution has marked peak.
  - It will be zero if distribution is flat.

Frequencies

We can use pandas built-in function value_counts() to obtain the frequency of the categorical variables

Describe

We also can use another pandas built-in function describe() to obtain insights of the data.

Visualization

We can use many types of graphs, some of them are listed below,

- if we want to show how various data elements contribute towards a whole, we should use pie chart.
- If we want to compare data elements, we should use bar chart.
- If we want to show distribution of elements, we should use histograms.
- If we want to depict groups in elements, we should use boxplots.
- If we want to find patterns in data, we should use scatterplots.
- If we want to display trends over time, we should use line chart.
- If we want to display geographical data, we should use basemap.
- If we want to display network, we should use networkx.

In addition to all these graphs, we also can use seaborn library to plot advanced graphs like,

- Countplot
- Heatmap
- Distplot
- Pairplot

Covariance

- Covariance is a measure used to determine how much two variables change in tandem.
- The unit of covariance is a product of the units of the two variables.
- Covariance is affected by a change in scale, The value of covariance lies between -∞ and +∞.
- Pandas does have built-in function to find covariance in DataFrame named cov()

Correlation

- The correlation between two variables is a normalized version of the covariance.
- The value of correlation coefficient is always between -1 and 1.
- Once we've normalized the metric to the -1 to 1 scale, we can make meaningful statements and compare correlations.