MACHINE LEARNING ALGORITHMS LABORATORY

V Semester: CSE (AI & ML)

VI Semester: CSE (DS)

Course Code	Category	Hours / Week			Credits	Maximum Marks		
		L	T	P	C	CIA	SEE	Total
ACAD11	Core	0	0	2	1	40	60	100
Contact Classes: Nil	Tutorial Classes: Nil	Practical Classes: 45				Total Classes: 45		
Prerequisite: Python Programming								

I. COURSE OVERVIEW:

This course introduces the fundamental concepts and methods of machine learning, including the description and analysis of several modern algorithms, their theoretical basis, and the illustration of their applications. Machine learning as a field is now incredibly pervasive, with applications spanning from business intelligence to text and speech processing, bioinformatics, and other areas in real-world products and services. This will familiarize students with a broad cross-section of models and algorithms for machine learning, and prepare students for research or industry application of machine learning techniques.

II. COURSE OBJECTIVES:

The students will try to learn:

I. The essential aspect of data preprocessing, integration, and visualization techniques to prepare and explore datasets for machine learning tasks.

II. The practical application of supervised, unsupervised, and ensemble learning algorithms for solving classification, regression, and clustering problems.

III. The suitability of advanced techniques such as semi-supervised learning, reinforcement learning, and evolutionary algorithms to address real-world challenges involving limited labels or optimization.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO1	Apply data preprocessing techniques to prepare datasets for machine learning models.
CO2	Identify appropriate supervised and unsupervised learning algorithms to solve classification, regression, and clustering problems.
CO3	Build machine learning models using ensemble methods and dimensionality reduction techniques to evaluate performance using appropriate metrics.
CO4	Utilize semi-supervised and reinforcement learning techniques to develop models for real-world scenarios.
CO5	Make use of evolutionary algorithms and deep learning architectures to solve optimization and generative tasks.
CO6	Solve natural language processing problems using machine learning and deep learning methods.

MACHINE LEARNING ALGORITHMS LABORATORY COURSE CONTENT

IV. COURSE CONTENT:

EXERCISES FOR MACHINE LEARNING ALGORITHMS LABORATORY

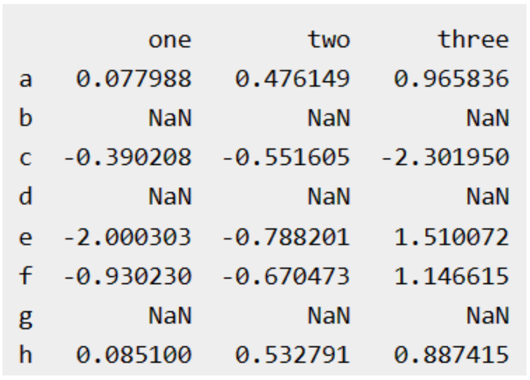
Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 Missing Values

In the present era, where data plays a pivotal role, businesses and organizations of all sizes encounter a substantial volume of data. However, ensuring the accuracy and reliability of this data is vital for making well-informed decisions and extracting meaningful information. Understand data consistency checks as being a set of expert rules to check whether a characteristic follows an expected behavior. The goal of this exercise is to increase the data consistency and quality by finding and removing errors like missing entries from data.

Input: Data Frame with missing values



```
        one        two      three
a  0.077988   0.476149   0.965836
b       NaN        NaN        NaN
c -0.390208  -0.551605  -2.301950
d       NaN        NaN        NaN
e -2.000303  -0.788201   1.510072
f -0.930230  -0.670473   1.146615
g       NaN        NaN        NaN
h  0.085100   0.532791   0.887415
```

Output: Data frame filled with missing values, value zero or with any other value.

Explanation: Handle missing values (say NA or NaN) using Pandas. To make detecting missing values easier (and across different array dtypes), Pandas provides the isnull( ) and notnull( ) functions, which are also methods on Series and DataFrame objects. The fillna function can “fill in” NA values with non-null data in a couple of ways.

Hint:

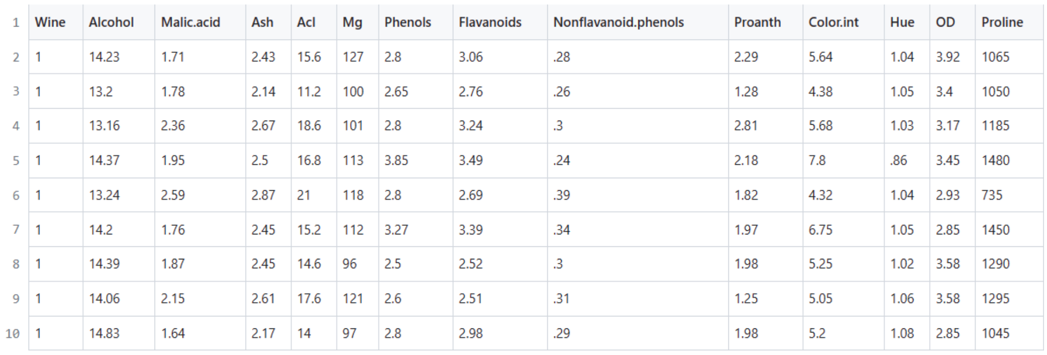
```
# Import the pandas library
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5,3), index=[‘a’, ‘c’, ‘e’, ‘f’, ‘h’],
     columns=[‘one’,’two’,’three’])

df = df.reindex([‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’]

# Print the dataframe
```

Try: Write the code to fill missing data with NaN and NaN with a Scalar Value

1.2 Feature Selection

Chemical analysis of wines grown in the same region and to determine the quantities of 13 constituents found in each of the three types of wines. Perform the feature selection by selecting the subset of the most relevant features from the original features set by removing the redundant, irrelevant, or noisy feature.

Input: wine.csv dataset and pima Indians diabetes dataset



Wine	Alcohol	Malic.acid	Ash	Acl	Mg	Phenols	Flavanoids	Nonflavanoid.phenols	Proanth	Color.int	Hue	OD	Proline
1	14.23	1.71	2.43	15.6	127	2.8	3.06	.28	2.29	5.64	1.04	3.92	1065
1	13.2	1.78	2.14	11.2	100	2.65	2.76	.26	1.28	4.38	1.05	3.4	1050
1	13.16	2.36	2.67	18.6	101	2.8	3.24	.3	2.81	5.68	1.03	3.17	1185
1	14.37	1.95	2.5	16.8	113	3.85	3.49	.24	2.18	7.8	.86	3.45	1480
1	13.24	2.59	2.87	21	118	2.8	2.69	.39	1.82	4.32	1.04	2.93	735
1	14.2	1.76	2.45	15.2	112	3.27	3.39	.34	1.97	6.75	1.05	2.85	1450
1	14.39	1.87	2.45	14.6	96	2.5	2.52	.3	1.98	5.25	1.02	3.58	1290
1	14.06	2.15	2.61	17.6	121	2.6	2.51	.31	1.25	5.05	1.06	3.58	1295
1	14.83	1.64	2.17	14	97	2.8	2.98	.29	1.98	5.2	1.08	2.85	1045

Output: heatmap showing the correlation of all the features of the dataset.

Explanation: Load the dataset first before loading the variables. Implement the code for Pearson Correlation and observe the results. If the value is near to 1 that means those two features are correlated and we can drop any one of them.

Hint:

```
# Importing the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# Read the csv file and print the dataframe
```

Try: Write the code to implement the Pearson Correlation and perform Chi-square test

1.3 Encoding the categorical data.

It has been observed that machine learning models do not understand any forms of data except integers. But there are many applications that need categorical data as input which will be in the form of strings and object data types. So, being a machine learning engineer, develop the code to convert the categorical data to numeric form.

Input: A finite set of categorical data represented as strings or categories.

1. The city where a person lives: Delhi, Mumbai, Ahmedabad, Bangalore, etc.
2. The department a person works in: Finance, Human resources, IT, Production.
3. The highest degree a person has: High school, Diploma, Bachelors, Masters, PhD.
4. The grades of a student: A+, A, B+, B, B- etc.

Output: Encoded data frame

Hint:

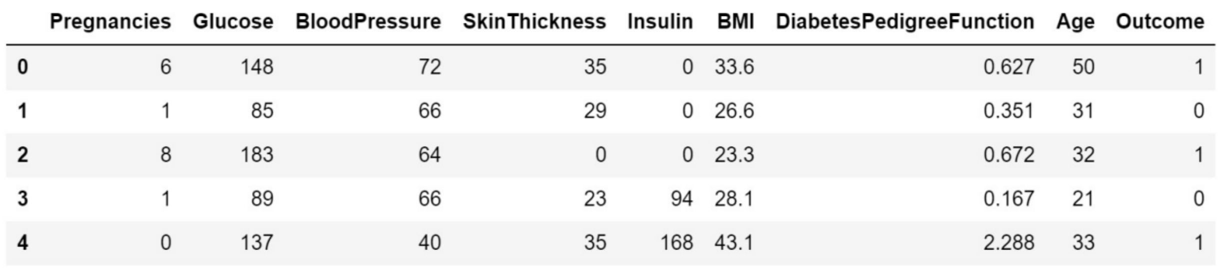
```
import category_encoders as ce
import pandas as pd
train df=pd.DataFrame({‘Degree’:[‘High school’, ‘Masters’,’Diploma’,
‘Bachelores’, ‘Masters’, ‘PhD’, ‘High school’, ‘High school’]})

# Create object for ordinal encoding

Encoder = ce.OrdinalEncoder(cols=[‘Degree’],return_df=True, mapping =
[{col:’Degree’, ‘mapping’:{‘None’:0,’High
school’:1,’Diploma’:2,’Bachelors’:3,’Masters’:4,’PhD’:5}}])

# Print the Original data
```

Try: Write the code to implement the one hot encoding and dump encoding.

1.4 Conversion of raw data into a clean data set

Data scientists process and analyze data using several methods and tools, such as statistical models, machine learning algorithms, and data visualization software. Data science seeks to uncover patterns in data that can help with decision-making, process improvement, and the creation of new opportunities. For achieving better results from the applied model in Machine Learning projects the format of the data must be in a proper manner. Some specified Machine Learning models need information in a specified format. The goal of this exercise is that the data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithm are executed in one data set, and the best out of them is chosen. Initialize the minmax scalar and learn the statistical parameters for each of the data and transforming.

Input: Pima Indian Diabetes dataset



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Output: Rescaled dataset with outcomes proportionality.

Explanation: Pre-processing refers to the transformations applied to our data before feeding it to the algorithm. Data preprocessing is a technique that is used to convert raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw fo rmat which is not feasible for analysis.

Hint:

```
# import the necessary libraries
import pandas as pd
import scipy
import numpy as np
from sklearn.preprocessing import MinMaxScaler
```

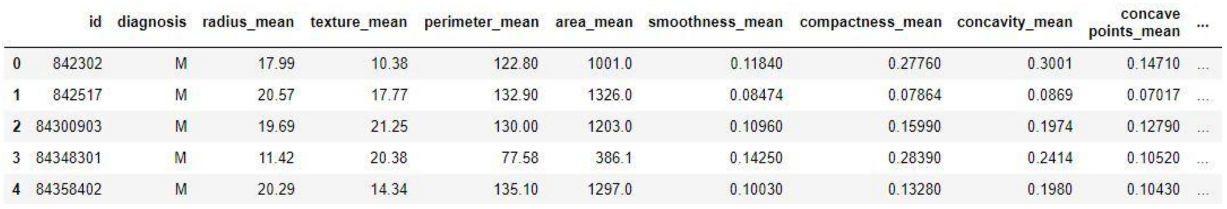
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('csv file path')
print(df.head()

# Write the code to check the dataset information and perform the statistical
analysis by checking and dropping the outliers.
.
.
.
# Use Correlation, check the outcomes proportionality, and separate the independent
features and target variable.
.
.
# Initialize the MinMaxScalar and learning the statistical parameters
scaler = MinMaxScaler(features_range=(0,1))
rescaledX = scaler.fit_transform(X)
rescaledX[:5]
```

Try: Apply Normalization and Standardization techniques on the above same dataset and generate the heat map.

1.5 Feature Scaling

In most cases, we shall work with datasets whose features are not on the same scale. Some features often have tremendous values, and others have small values. Suppose we implement our machine learning model on such datasets. In that case, features with tremendous values dominate those with small values, and the machine learning model treats those with small values as if they don't exist (their influence on the data is not accounted for). To ensure this is not the case, we need to scale our features on the same range, i.e., within the interval of -3 and 3.

Input: Breast Cancer Dataset (csv file)



	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...

Output: Scaled values for 'Age' and 'Salary' columns.

Explanation: Load the dataset, take care of missing values, and transform the text data into numeric data. Split the dataset into training and testing sets and perform the feature scaling.

Hint:

```
from sklearn.preprocessing import StandardScalar

sc = StandardScaler()

# Apply the feature scaling on the features other than dummy variables

x_train[:, 3:] = sc.fit_trandorm(x_train[:, 3:])
```

```
x_test[:, 3:] = sc.fit_tranform(x_test[:, 3:])
```

Try: Output the training and testing values. Visualize the results with different plots.

2. Data Integration and Visualization

2.1 Data Integration, Transformation and Visualization

In today's world of technology and smart business decisions, data integration plays a significant role. Integrating data generated from multiple applications and working on it has become the flagship of some of the IT projects run by various organizations around the world. The need for improving data accessibility has given rise to the idea of Data integration. The goal of this exercise is to perform some of the basic transformations and visualize the data.

Input: Superstore Sales Dataset with Orders, Returns, and People data.

Output: An integrated dataset, basic filtering outcomes, and a histogram

Explanation:

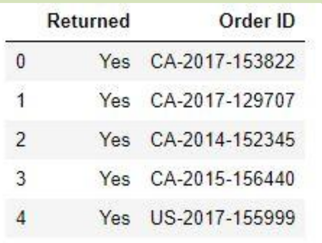
The dataset I will use for this tutorial is the superstore sales dataset. This dataset contains information on the sales made by a fictional retail outlet in the USA. The dataset is provided in .xls format and the data is spread across three different sheets:

- Orders: This is the main table in the dataset and contains information for each order such as the order id, the store location, the quantity, and the profit.
- Returns: This table contains a list of order id's which have been returned.
- People: This table contains the names of the regional manager for each region contained within the orders table.

We can think of these sheets as three tables in a database. We can easily read each of these sheets into memory using pandas and examine the first few lines as demonstrated below.

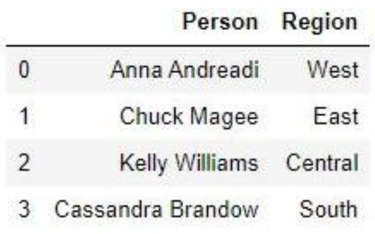
Hint:

```
import pandas as pd
orders = pd.read_excel('./data/input_data/superstore.xls', sheet_name='Orders')
returns = pd.read_excel('./data/input_data/superstore.xls', sheet_name='Returns')
people = pd.read_excel('./data/input_data/superstore.xls', sheet_name='People')

# View the first few lines of the data
orders.head()
returns.head()
```



	Returned	Order ID
0	Yes	CA-2017-153822
1	Yes	CA-2017-129707
2	Yes	CA-2014-152345
3	Yes	CA-2015-156440
4	Yes	US-2017-155999

```
people.head()
```



	Person	Region
0	Anna Andreadi	West
1	Chuck Magee	East
2	Kelly Williams	Central
3	Cassandra Brandow	South

Joining the data sources together

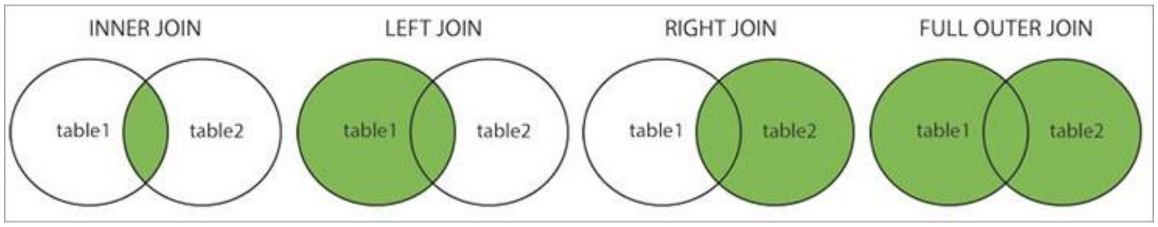
If you load and explore the data sources, you will notice that the orders table shares common fields with both the returns (Order ID) and people (Region) tables. These relationships mean that we can combine the three data sources into a single "master" table using a join.

Pandas make joining tables very simple through the merge. This allows us to join two tables using two or more columns and specify the type of join (inner, left, right, outer) we wish to use.

```
merged_df = orders.merge(
    returns, how='left', on='Order ID'
    ).merge(people, on='Region')
```

The code above demonstrates how one would go about joining our three data sources together into one master table (merged_df). A few key points to note:

1. The dataframe we apply the. merge method to (orders in our case) is considered the left table in the join while the dataframe inside the parenthesis (returns and people in our case) are considered the right.
2. We can perform both joins in a single line by simply chaining the. merge() operations together.
3. By default, the merge method preforms an inner join. However, since the returns table only contained information on returned orders an inner join here would mean that we would have lost all the non-returned orders from the orders dataset. As a result of this we had to specify that we wanted to use a left join. Since we knew that all regions in the orders dataset appear in the people dataset, we were comfortable using an inner join when merging the people dataset. If you are unfamiliar with these joins, see below for a visual explanation.



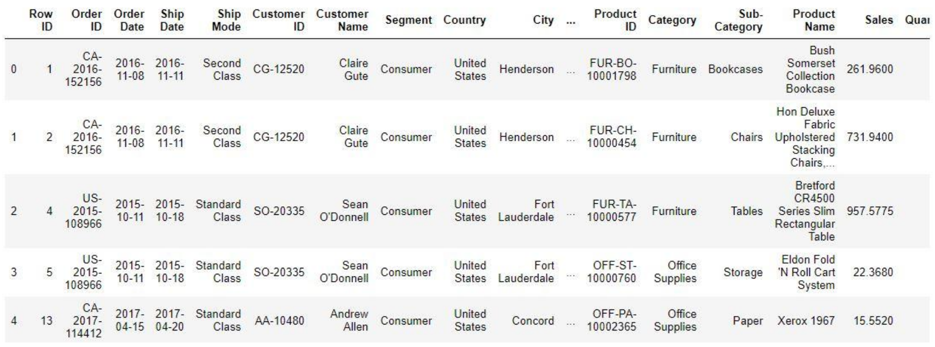
Some Basic Filtering

Once we have our datasets merged, the next logical step might be to filter out rows which relate to returned orders (i.e order ID's which exist in the returns table) as we might consider these invalid for our analysis.

Pandas make filtering incredibly easy using the loc method. This method allows us to access a column, or a group of columns using a boolean index. When using the loc method, we declare our boolean index as the first input, followed by the group of columns we wish to select from the filtered subset.

In the example below, I use the loc method to filter the dataframe based on rows which have a NA value for the column'Returned' (since these are the ID's which are not in the returned table). I then specify that I want to select all columns using the : slicer.

```
not_returned_df = merged_df.loc[merged_df['Returned'].isna(), :]
```

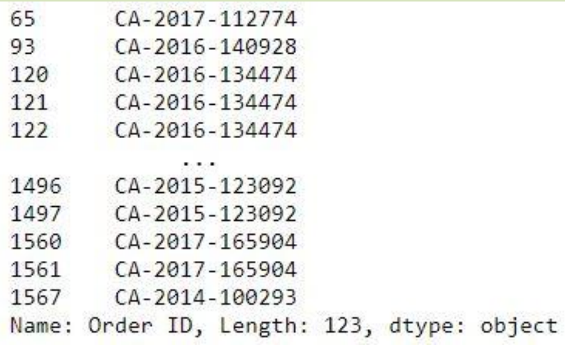


	Row ID	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	...	Product ID	Category	Sub-Category	Product Name	Sales	Quai
0	1	CA-2016-152156	2016-11-08	2016-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	...	FUR-BO-10001798	Furniture	Bookcases	Bush Somerset Collection Bookcase	261.9600	
1	2	CA-2016-152156	2016-11-08	2016-11-11	Second Class	CG-12520	Claire Gute	Consumer	United States	Henderson	...	FUR-CH-10000454	Furniture	Chairs	Hon Deluxe Fabric Upholstered Stacking Chairs,...	731.9400	
2	4	US-2015-108966	2015-10-11	2015-10-18	Standard Class	SO-20335	Sean O'Donnell	Consumer	United States	Fort Lauderdale	...	FUR-TA-10000577	Furniture	Tables	Bretford CR4500 Series Slim Rectangular Table	957.5775	
3	5	US-2015-108966	2015-10-11	2015-10-18	Standard Class	SO-20335	Sean O'Donnell	Consumer	United States	Fort Lauderdale	...	OFF-ST-10000760	Office Supplies	Storage	Eldon Fold 'N Roll Cart System	22.3680	
4	13	CA-2017-114412	2017-04-15	2017-04-20	Standard Class	AA-10480	Andrew Allen	Consumer	United States	Concord	...	OFF-PA-10002365	Office Supplies	Paper	Xerox 1967	15.5520	

In addition to performing simple filtering base on the values in one column we can also chain logic together across multiple columns. Using the code below, we can extend our example from above to identify the order ID's of non-returned items in the city of Jacksonville.

Hint:

```
not_returned_jax = merged_df.loc[
    (merged_df['Returned'].isna()) &
    (merged_df['City'] == 'Jacksonville'),
    'Order ID'
]
```



```
65      CA-2017-112774
93      CA-2016-140928
120     CA-2016-134474
121     CA-2016-134474
122     CA-2016-134474
             ...
1496    CA-2015-123092
1497    CA-2015-123092
1560    CA-2017-165904
1561    CA-2017-165904
1567    CA-2014-100293
Name: Order ID, Length: 123, dtype: object
```

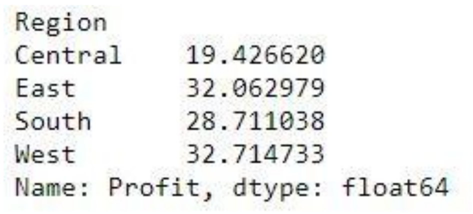
Aggregating the dataset

When exploring a dataset, it is often useful to aggregate the data up to a particular column value and view the data at a higher level.

For example, if we want to explore the mean profit per item in each region, we can use the groupby method (illustrated below) to group the dataset by the region column before specifying that we want to see the mean of the "Profit" column.

Hint:

```
agg_example_1 = not_returned_df.groupby(
    by='Region'
)['Profit'].mean()
```



```
Region
Central    19.426620
East       32.062979
South      28.711038
West       32.714733
Name: Profit, dtype: float64
```

I have illustrated this in the example below by aggregating the data up to region level before calculating the mean profit and median sales within each region.

Hint:

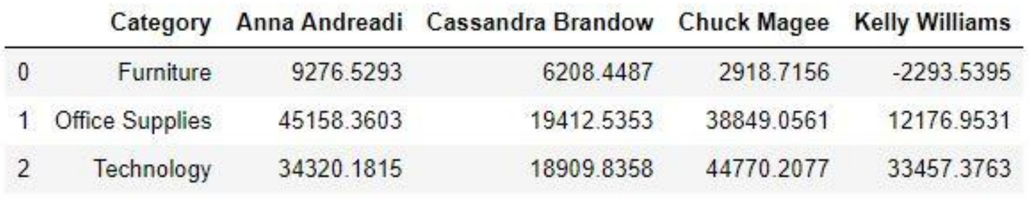
```
agg_example_2 = not_returned_df.groupby(
    by='Region'
).agg({'Profit':'mean', 'Sales':'median'})
```

Pivoting and Unpivoting the dataset

Like aggregating the dataset, a pivot can often be useful when summarizing a dataset for a report. Pivoting involves turning a single column into multiple columns (1 for each value in the original column). This idea can be difficult to understand but let's illustrate this by using the pivot method in pandas. In our example we use the groupby method to calculate the profit per manager for each product category before using a pivot to display the results in a more readable format.

Hint:

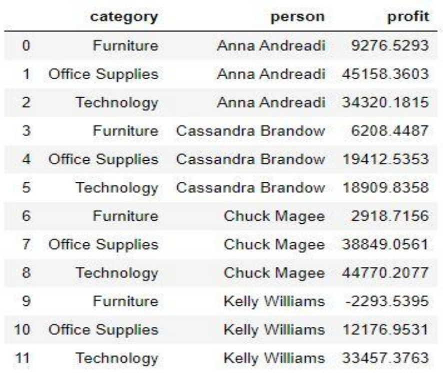
```
# group by person and category to aggregate
data_for_pivot = not_returned_df.groupby(
    by=['Person', 'Category']
)['Profit'].sum().reset_index()


# Perform pivot
data_for_pivot = pd.DataFrame(
    data_for_pivot.pivot(
        index='Category',
        columns='Person',
        values='Profit'
).to_records())
data_for_pivot
```



	Category	Anna Andreadi	Cassandra Brandow	Chuck Magee	Kelly Williams
0	Furniture	9276.5293	6208.4487	2918.7156	-2293.5395
1	Office Supplies	45158.3603	19412.5353	38849.0561	12176.9531
2	Technology	34320.1815	18909.8358	44770.2077	33457.3763

Hint:

```
unpivoted_data = data_for_pivot.melt(id_vars=['Category'])
unpivoted_data.columns = ['category', 'person', 'profit']
unpivoted_data
```



	category	person	profit
0	Furniture	Anna Andreadi	9276.5293
1	Office Supplies	Anna Andreadi	45158.3603
2	Technology	Anna Andreadi	34320.1815
3	Furniture	Cassandra Brandow	6208.4487
4	Office Supplies	Cassandra Brandow	19412.5353
5	Technology	Cassandra Brandow	18909.8358
6	Furniture	Chuck Magee	2918.7156
7	Office Supplies	Chuck Magee	38849.0561
8	Technology	Chuck Magee	44770.2077
9	Furniture	Kelly Williams	-2293.5395
10	Office Supplies	Kelly Williams	12176.9531
11	Technology	Kelly Williams	33457.3763

Apply a custom function to a dataframe.

Oftentimes when transforming a dataset, we will want to apply a function we have created to a dataframe. Some examples of this include:

- A function that will convert an address column to coordinates.
- A function that will create a binary flag based on some logical condition.
- A function that will convert a text column to a vector representation.

Pandas allows us to do this using the apply method with lambda. This method will simply loop through each row of the data and apply our function to the column(s) we specify. For example, we can use the code below to create a "special_offer" column by applying a function which determines if an item has a discount rate of > 50%, and if so it returns a 1.

Hint:

```
def special_offer(discount):
      if discount > 0.5:
        return 1
    else:
        return 0


not_returned_df.loc[:, 'special_offer'] = not_returned_df.apply(
    lambda row: special_offer(row['Discount']), axis=1
)
```

Note that when applying a custom function it is important to include the lambda within the method. It is also important to note that setting axis = 1 indicates that we want to loop through the dataframe row-wise instead of column-wise.

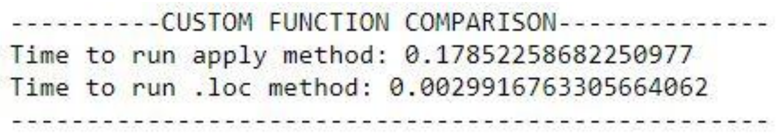
While it is great to be able to leverage the apply method for custom functions on our dataframe, it is important to remember that this method is essentially just looping through every row of the dataframe and applying the function over and over. This can often result in long wait times when dealing with larger datasets so it is important to be smart about how and when we use the method.

For example, when creating a binary column it is more computationally efficient to create a column filled with zeros, then use the loc method to filter to the columns which meet our criteria and set them to one.

You will see in the code and output below that using this method for our simple function takes much less time than the apply method approach.

Hint:

```
# Apply approach
start_time_loc = time.time()
not_returned_df.loc[:, 'special_offer'] = 0
# loc approach
not_returned_df.loc[not_returned_df['Discount'] > 0.5, 'special_offer'] = 1
print ("Time to run .loc method:", time.time() - start_time_loc)
```



```
----------CUSTOM FUNCTION COMPARISON--------------
Time to run apply method: 0.17852258682250977
Time to run .loc method: 0.0029916763305664062
--------------------------------------------------
```

Plotting a dataframe

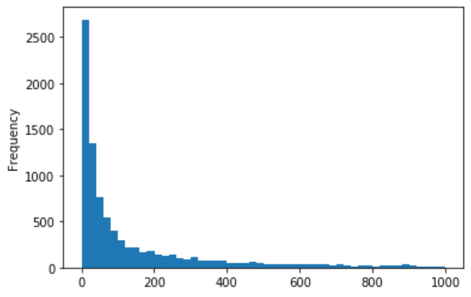
Visualization is a key part of the data exploration process and should be used in every data science task. While modules such as matplotlib, seaborn, and plotly allow us to produce some lovely, clean visuals for reports and presentations, sometimes we just want to pull together something quick and examine a particular feature in our dataset. In these cases, I will always try to use the pandas plot method as it is incredibly simple and effective.

In the examples below you can see how we can apply this method to very easily to create a histogram of sales prices (< 1000 due to large tail) and a scatter plot (sales vs discount). These types of visuals can be extremely helpful when understanding how a dataset is distributed, identifying outliers, and understanding correlations.

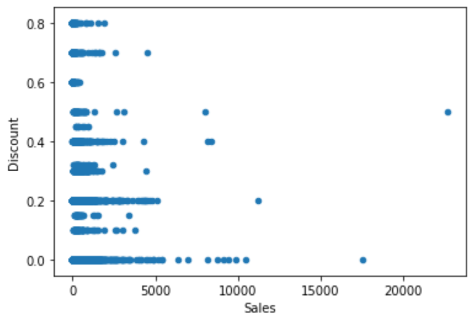
Hint:

```
ax = not_returned_df.loc[not_returned_df['Sales'] < 1000,
'Sales'].plot.hist(bins=50)
```

```
fig = ax.get_figure()
```



```
ax = not_returned_df.plot.scatter(x='Sales', y='Discount')
fig = ax.get_figure()
```



Try:

1. Join the datasets together in such a way that you are only left with information for returned orders.
2. Filter the dataset to find orders with a sales value < 500 in the bookcases or tables sub-category.
3. Aggregate the data to show the mean profit per sub-category.
4. Aggregate and pivot the data to show the profit for each shipping method (rows) in each sub-category (columns)
5. Create a function which identifies if an item was shipped within 3 days of ordering and apply it to the dataset.
6. Create a bar plot of the number of orders per category.

2.2 Data Reduction

Since data mining is a technique that is used to handle huge amounts of data. While working with a huge volume of data, analysis became harder in such cases. The goal of this exercise is to use the data reduction technique and aims to increase storage efficiency, reduce data storage and analysis costs.

Input: Iris Dataset

Output: Dataset with removed features whose variance doesn't meet the threshold.

Explanation:

Data reduction is also called dimensionality reduction. This reduces the size of data by encoding mechanisms. It can be lossy or lossless. If after reconstruction from compressed data, original data can be retrieved, such reduction is called lossless reduction else it is called lossy reduction. The two effective methods of dimensionality reduction are: Wavelet transforms and PCA (Principal Component Analysis).

PCA is sensitive to the relative scaling of the original variables.

Hint:

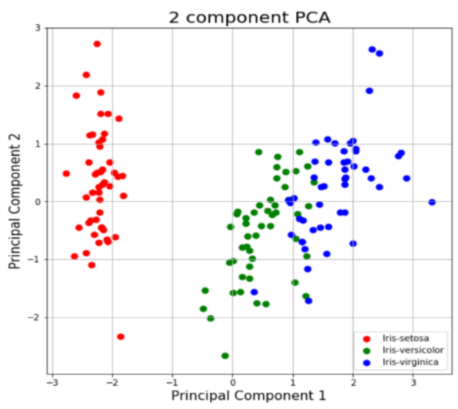
```
url=”iris dataset file path’
# load dataset into Pandas Dataframe

df.head( )
```

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Hint:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pct = pca.fit_transform(x)
principle_df = pd.DataFrame(pct, columns=[‘pc1’, ‘pc2’])
finaldf = pd.concat([principal_df, df[[‘target’]]], axis=1)

# Read the top 5 records of the dataset
```

	pc1	pc2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa



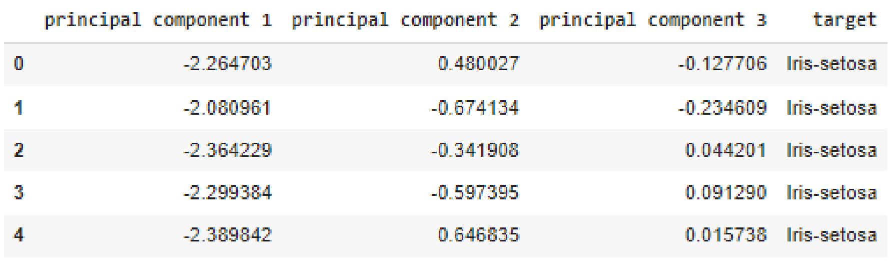
The explained variance tells you how much information (variance) can be attributed to each of the principal components. This is important as while you can convert 4-dimensional space to 2-dimensional space, you lose some of the variance (information) when you do this. By using the attribute explained_variance_ratio_, you can see that the first principal component contains 72.77% of the variance and the second principal component contains 23.03% of the variance. Together, the two components contain 95.80% of the information.

PCA Projected to 3D.

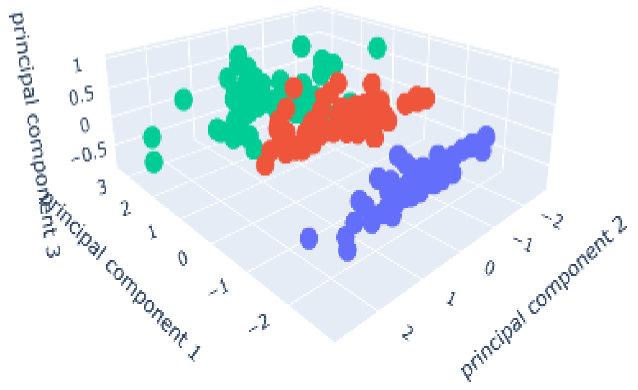
The original data has 4 columns (sepal length, sepal width, petal length, and petal width). Here, the code projects the original data which is 4 dimensional into 3 dimensions. The new components are just the three main dimensions of variation.

Hint:

```
pca = PCA(n_components=3)
# Use fit function to transform X
principal_Df1 = pd.DataFrame(data = principalComponents,
      columns = [‘principal component1’, ‘principal component2’, ‘principal
component3’,])
finalD_f1.head(5)
```



	principal component 1	principal component 2	principal component 3	target
0	-2.264703	0.480027	-0.127706	Iris-setosa
1	-2.080961	-0.674134	-0.234609	Iris-setosa
2	-2.364229	-0.341908	0.044201	Iris-setosa
3	-2.299384	-0.597395	0.091290	Iris-setosa
4	-2.389842	0.646835	0.015738	Iris-setosa



Variance Threshold

Variance Threshold is a simple baseline approach to feature selection. It removes all features whose variance doesn’t meet some threshold. By default, it removes all zero-variance features. Our dataset has no zero-variance feature, so our data isn’t affected here.

Hint:

```
sel_variance_threshold = VarianceThreshold( )

X_train_remove_variance = sel_variance_threshold.fit_tranform(X_train)

X_train_remove_variance.shape

# Variance threshof is applied but since the noise valued columns have non-zero
variance, they aren’t deleted.
```

```
(120, 8)
```

Try: Imagine you have 1000 predictor features and 1 target feature in a machine learning problem. You must select the 100 most important features based on the relationship between input features and the target features. Develop the code to state whether the problem belongs to dimensionality reduction or not.

2.3 Data Visualization

In today's world, a lot of data is being generated daily. And sometimes to analyze this data for certain trends, patterns may become difficult if the data is in its raw format. To overcome this data visualization comes into play. Data visualization provides a good, organized pictorial representation of the data which makes it easier to understand, observe, and analyze. The goal of this exercise is to understand how to visualize data using Python.

Input: Tips Dataset

Output: Different data plots

Explanation:

Python provides various libraries that come with different features for visualizing data. All these libraries come with different features and can support various types of graphs. In this tutorial, we will be discussing four such libraries.

- Matplotlib
- Seaborn
- Bokeh
- Plotly

Tips database is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s. It contains 6 columns such as total_bill, tip, sex, smoker, day, time, size.

Hint:

```
import pandas as pd

# Write the code to read the database here

# printing the top 10 rows
display(data.head(10))
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
5	25.29	4.71	Male	No	Sun	Dinner	4
6	8.77	2.00	Male	No	Sun	Dinner	2
7	26.88	3.12	Male	No	Sun	Dinner	4
8	15.04	1.96	Male	No	Sun	Dinner	2
9	14.78	3.23	Male	No	Sun	Dinner	2

Matplotlib is an easy-to-use, low-level data visualization library that is built on NumPy arrays. It consists of various plots like scatter plot, line plot, histogram, etc. Matplotlib provides a lot of flexibility.

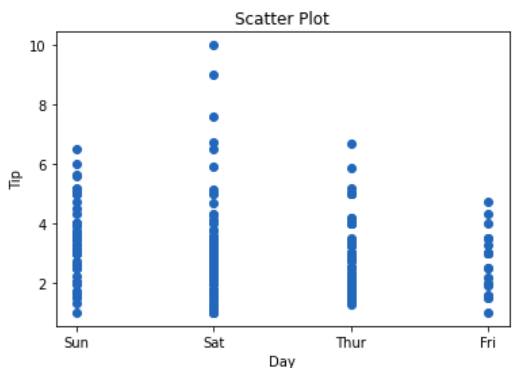
After installing Matplotlib, let's see the most used plots using this library.

Scatter Plot

Scatter plots are used to observe relationships between variables and uses dots to represent the relationship between them. The scatter() method in the matplotlib library is used to draw a scatter plot.

Hint:

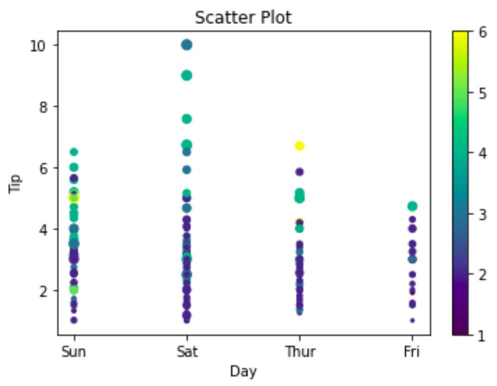
```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Write the code to display the scatter plot with day against tip

# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')

# Write the statement to show the plot
```



This graph can be more meaningful if we can add colors and change the size of the points. We can do this by using the **c and s** parameter respectively of the scatter function. We can also show the color bar using the colorbar() method.

Hint:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Scatter plot with day against tip
plt.scatter(data['day'], data['tip'], c=data['size'],
            s=data['total_bill'])
# Write the code to add Title to the Plot

# Set the X, Y labels and print the plot
```

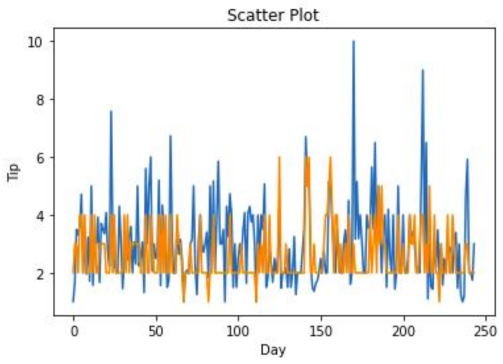


Line Chart

Line Chart is used to represent a relationship between two data X and Y on a different axis. It is plotted using the **plot()** function.

Hint:

```
import pandas as pd
import matplotlib.pyplot as plt

# Write the code to read the database and display the Scatter plot with day
against tip

# Adding Title to the Plot
plt.title("Scatter Plot")

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')
plt.show()
```

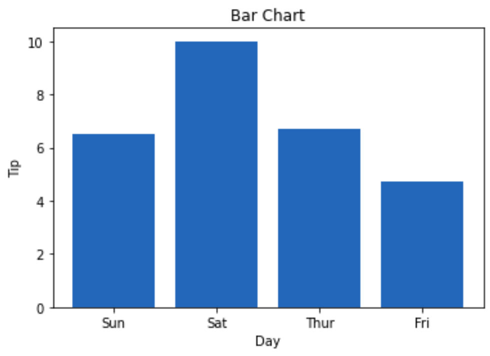


Bar Chart

A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. It can be created using the **bar()** method.

Hint:

```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")

# Writ the code to display the bar chart with a title and a day against tip

# Setting the X and Y labels
plt.xlabel('Day')
plt.ylabel('Tip')

# Adding the legends
plt.show()
```

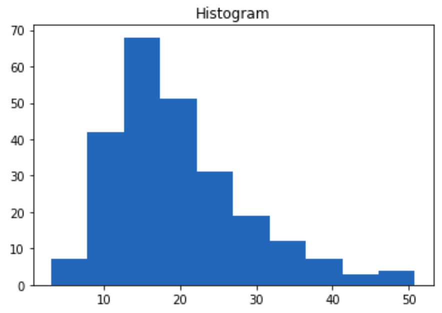


Histogram

A histogram is basically used to represent data in the form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The **hist()** function is used to compute and create a histogram. In histogram, if we pass categorical data then it will automatically compute the frequency of that data i.e. how often each value occurred.

Hint:

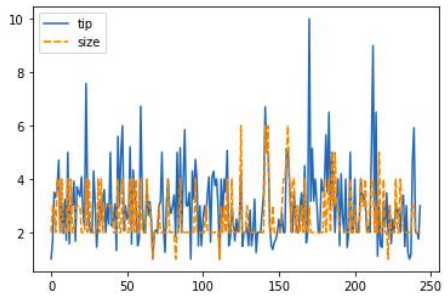
```
import pandas as pd
import matplotlib.pyplot as plt

# reading the database
data = pd.read_csv("tips.csv")
# histogram of total_bills
plt.hist(data['total_bill'])
plt.title("Histogram")

# Write the statement to add the legends and display the chart
```



Hint:

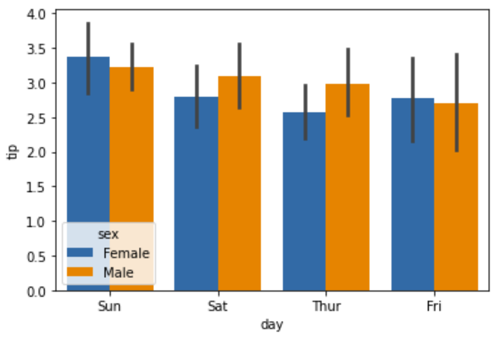
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# reading the database
data = pd.read_csv("tips.csv")
```

```
# Write the code to display the line chart using only data attribute
```



Hint:

```
# importing packages such as seaborn, matplotlib, and pandas

# reading the database
data = pd.read_csv("tips.csv")

sns.barplot(x='day',y='tip', data=data,
            hue='sex')
plt.show()
```



Try:

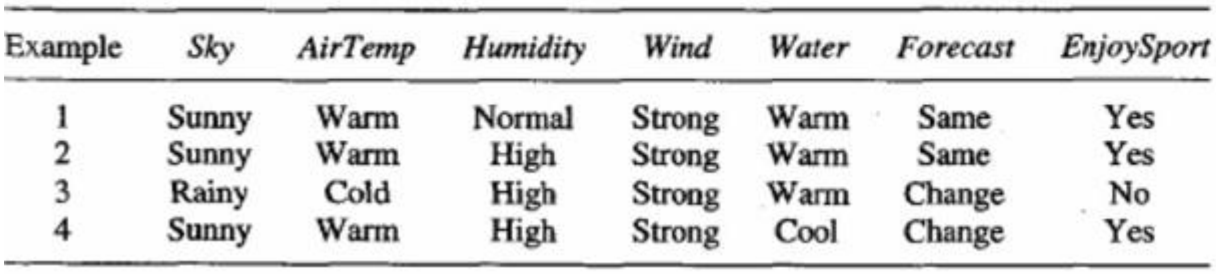
Develop the code to plot the visualizations like histogram, scatter plot, line chart, bar chart and other interactive data visualizations for different datasets.

3. Theory of Concept Learning

3.1 Find-S Algorithm

FIND S Algorithm is used to find the Maximally Specific Hypothesis. Using the Find-S algorithm gives a single maximally specific hypothesis for the given set of training examples.

Input:

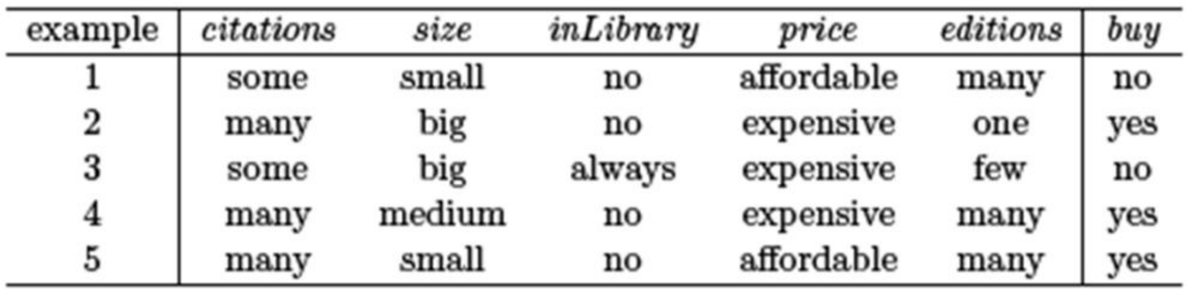


Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Output: The final maximally specific hypothesis is <Sunny, Warm, ?, Strong, ?, ?>
Hint:

```
# 1st iteration
h0 = (ϕ, ϕ, ϕ, ϕ, ϕ, ϕ, ϕ)
X1 = <Sunny, Warm, Normal, Strong, Warm, Same>
h1 = <Sunny, Warm, Normal, Strong, Warm, Same>

# 2nd iteration
h1 = <Sunny, Warm, Normal, Strong, Warm, Same>
X2 = <Sunny, Warm, High, Strong, Warm, Same>
h2 = <Sunny, Warm, ?, Strong, Warm, Same>

#Find the final maximally specific hypothesis
```

Try:



example	*citations*	*size*	*inLibrary*	*price*	*editions*	*buy*
1	some	small	no	affordable	many	no
2	many	big	no	expensive	one	yes
3	some	big	always	expensive	few	no
4	many	medium	no	expensive	many	yes
5	many	small	no	affordable	many	yes

For the above training set find out,
How many concepts are possible for this instance space?
How many hypotheses can be expressed by the hypothesis language?
Apply the FIND-S algorithm by hand on the given training set.

3.2 Candidate Elimination Algorithm

Candidate Elimination Algorithm is used to find the set of consistent hypotheses, that is Version space**.**

Input:

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rain	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

Output: Version Space by Candidate Elimination Algorithm for given data set is: S: G: (Small, ?, Circle)

Hint:

```
# Initialization
S0: (0, 0, 0) Most Specific Boundary
G0: (?,  ?,  ?) Most Generic Boundary

# 1st example
S1: (0, 0, 0)
G1: (Small, ?, ?), (?, Blue, ?), (?, ?, Triangle)
# 2nd iteration
h1 = <Sunny, Warm, Normal, Strong, Warm, Same>
X2 = <Sunny, Warm, High, Strong, Warm, Same>
h2 = <Sunny, Warm, ?, Strong, Warm, Same>

#Version Space by Candidate Elimination Algorithm for given data set
```

Try:

Example	Size	Color	Shape	Class/Label
1	Big	Red	Circle	No
2	Small	Red	Triangle	No
3	Small	Red	Circle	Yes
4	Big	Blue	Circle	No
5	Small	Blue	Circle	Yes

Find the version Space by Candidate Elimination Algorithm for given data set.

4. Supervised Learning Algorithms (Regression)

4.1 Linear Regression

Let's say we're the owners of a candy store, Willy Wonka's Candy, and we want to do a better job of predicting how much our customers will spend this week, to stock our shelves more appropriately. To get even more specific, let's explore one specific customer named George. George is a 65-year-old mechanic who has children and spent $10 at our store last week. The goal of this exercise is to have a simple model to predict how much George will spend at Willy Wonka's Candy this week.

Input: Custom Dataset

Output: Performance metric values including MSE, RMSE, MAE, and R2.

Explanation:

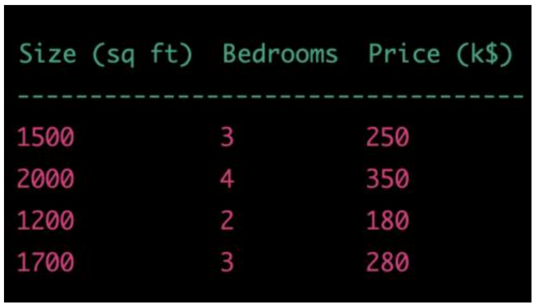
Linear Regression is one of the most widely used Artificial Intelligence algorithms in real-life Machine Learning problems — thanks to its simplicity, interpretability, and speed. In the next few minutes, we'll understand what's behind the working of this algorithm.

Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables. It is a popular technique for predicting the value of the dependent variable based on the values of the independent variables. Linear regression assumes that there is a linear relationship between the dependent variable and the independent variables, which means that a change in one independent variable leads to a proportional change in the dependent variable.

In regression, the difference between the real value of the dependent variable(yi) and the predicted value(predicted) is called the residuals.

# Equation To Calculate the Random Error
εi = ypredicted – yi
where ypredicted = B0 + B1 Xi

Example Dataset
Suppose we have a dataset containing information about houses in a particular city. The dataset has the following columns:
Size (in square feet)
Number of Bedrooms
Price (in thousands of dollars)
Here are the first few rows of the dataset:



```
Size (sq ft)  Bedrooms  Price (k$)
----------------------------------
1500          3         250
2000          4         350
1200          2         180
1700          3         280
```

We want to use this dataset to build a linear regression model that can predict the price of a house based on its size and number of bedrooms. To calculate Evaluation Metrics, we first make predictions using our linear regression model and then calculate the Evaluation Metrics

Hint:

```
#Import The LinearRegression from sklearn
from sklearn.linear_model import LinearRegression

# Load the dataset

X = [[1500, 3], [2000, 4], [1200, 2], [1700, 3]]
y = [250, 350, 180, 280]


# Fit the linear regression model and make the predictions on the same data
```

1 Mean Squared Error (MSE):

2. Root Mean Squared Error (RMSE):

3 Mean Absolute Error (MAE):

4 R-Squared (R2):

Hint:

```
from sklearn.metrics import mean_squared_error

# y_pred are the predicted values of the dependent variables

# y_actual are the actual values of the dependent variables
mse = mean_squared_error(y_actual, y_pred)

print("Mean Squared Error (MSE) = ", mse)
```

# Output MSE

18.75

Hint:

```
from sklearn.metrics import mean_squared_error

import numpy as np

# y_pred are the predicted values of the dependent variables

# y_actual are the actual values of the dependent variables
mse = mean_squared_error(y_actual, y_pred)
rmse = np.sqrt(mse)
print("Mean Squared Error (MSE) = ", mse)
```

# Output RMSE

2.8421709

# Equation For MAE is

MAE = (1/n) * ∑|y_pred - y_actual|

MAE gives a measure of how well the model predicts the dependent variable. A lower MAE indicates a better prediction.

# Output MAE

18.75

Hint:

```
from sklearn.metrics import r2_score

# y_pred are the predicted values of the dependent variables
# y_actual are the actual values of the dependent variables

mse = r2_score(y_actual, y_pred)
print("R-Squared (R2) score = ", r2)
```

# Output R-Square

1.0

Try: There are several evaluation metrics that can be used to assess the performance of a linear regression model. MSE, RMSE, and MAE give a measure of how well the model fits the data and predicts the dependent variable, while R2 measures how well the model explains the variation in the dependent variable. Run any machine learning model and use multiple evaluation metrics to ensure that the model is accurate and reliable.

4.2 Logistic Regression

Let's say we're the owners of a candy store, Willy Wonka's Candy, and we want to do a better job of predicting how much our customers will spend this week, to stock our shelves more appropriately. To get even more specific, let's explore one specific customer named George. George is a 65-year-old mechanic who has children and spent $10 at our store last week. The goal of this exercise is to predict whether George will be a high spender.

Input: Custom Dataset

Output: Predict whether George will be the high spender with optimal accuracy.

Explanation:

Logistic Regression is one of the most widely used Artificial Intelligence algorithms in real-life Machine Learning problems — thanks to its simplicity, interpretability, and speed. In the next few minutes, we'll understand what's behind the working of this algorithm.

In logistic regression, the dependent variable is binary, meaning it can only take on two values, typically labeled as 0 or 1. The independent variables can be either continuous or categorical.

Assume we have a binary classification problem, and we are given the predicted probabilities and the true labels for a set of instances:

Hint:

```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
roc_auc_score, confusion_metrix

# predicted probabilities
y_pred = np.array([0.3, 0.6, 0.8, 0.2, 0.4, 0.9, 0.1, 0.7, 0.5, 0.6])

# true labels
y_true = np.array( [0, 1, 1, 0, 0, 1, 0, 1, 0, 1] )
```

4 Accuracy:

Hint:

```
# Calculate accuracy

accuracy = accuracy_score(y_true, np.round(y_pred) )
print(f”Accuracy: {accuracy: .3f}”)
```

Hint:

```
# Calculate the confusion matrix and print the TP and FN values
```

Hint:

```
## Import The libraries
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, log_loss
from sklearn.model_selection import train_test_split

# Load iris dataset
iris = load_iris()

# Set features and target
X = iris.data
y = iris.target

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train a logistic regression model and make predictions on the testing set

# Compute evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
auc_roc = roc_auc_score(y_test, clf.predict_proba(X_test), multi_class='ovr')
logloss = log_loss(y_test, clf.predict_proba(X_test))

# Print evaluation metrics
print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1 score:', f1)
print('AUC-ROC:', auc_roc)
print('Log Loss:', logloss)
```

# Output:

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1 score: 1.0
AUC-ROC: 1.0
Log Loss: 0.04542674063376945
```

Try: Use the same code and apply on different datasets and do the comparative analysis for different performance metrics.

4.3 Polynomial Regression

A good result is provided if a linear model is applied to a linear database, as is the case with simple linear regression. However, a drastic output is produced if the same model is applied to a non-linear dataset with no modifications. These cause an increase in the loss function, high error rates, and a decrease in accuracy. The goal of this exercise is to prove the need of polynomial regression for the data points that are not arranged in a linear fashion.

Input: Generate some sample data points.

Output: Performance metric values for RMSE, R2, and Adjusted R2.

Explanation:

Regression analysis is a statistical technique used to estimate the relationship between a dependent variable to one or more independent variables. And a linear regression model may not provide accurate results. In such cases, polynomial regression can be used.

This is a type of regression analysis that models the relationship between the independent variable to the dependent variable as an nth-degree polynomial.

Let’s take a closer look at each of these evaluation metrics and how they can be calculated in Python using the scikit-learn library:

Hint:

```
#Import libraries
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Generate some sample data
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)
y = np.array([4, 5, 6, 9, 10, 11, 12, 13, 14, 15]).reshape(-1, 1)
```

```
# Fit a polynomial regression model of degree 2 and make predictions on the test data

# Calculate evaluation metrics
mse = mean_squared_error(y, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y, y_pred)
n = len(y)
p = 2 # number of predictors (degree of polynomial + 1)
adj_r2 = 1 - ((1-r2)*(n-1)/(n-p-1))

print("Mean Squared Error (MSE):", mse)
print("Root Mean Squared Error (RMSE):", rmse)
print("R-squared (R2) Score:", r2)
print("Adjusted R-squared Score:", adj_r2)
```

#The output of the code is as follows:

```
Mean Squared Error (MSE): 0.21013158229031288
Root Mean Squared Error (RMSE): 0.45828318396667815
R-squared (R2) Score: 0.9493669629929204
Adjusted R-squared Score: 0.9365232252020057
```

Try: One of the main challenges of polynomial regression is overfitting. If the degree of the polynomial is too high, the model may fit the training data too closely and may not generalize well to new data. Justify by implementing the code for the same statement.

4.4 Quantile Regression

The goal of this exercise is to focus on estimating the conditional quantiles of the dependent variable, rather than just the conditional mean, and provide insights into the shape and variability of the distribution of the dependent variable.

Input: Boston Dataset

Output: Quantile Specific Coefficients (QuantReg Regression Results)

Explanation:

In Machine Learning, Quantile regression is a statistical technique used to model the relationship between a dependent variable and one or more independent variables, by estimating the conditional quantiles of the dependent variable.

The Equation for Quantile Regression

Following is the equation of quantile regression in machine learning

#The equation for quantile regression

$Q(y \mid x) = x\beta(q)$

where,

Q(y | x) is the q-th quantile of the conditional distribution of y given x

β(q) is the vector of regression coefficients for the q-th quantile, and

x is the vector of independent variables.

In this equation, q represents the desired quantile, such as the 10th, 25th, 50th, 75th, or 90th percentile. The coefficient vector β(q) provides information on the effect of the independent variables on the q-th quantile of the dependent variable.

Hint:

```
# Import necessary modules
from sklearn.datasets import load_boston
from sklearn.linear_model import QuantileRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error

# load Boston Housing dataset
boston = load_boston()
X = boston.data
y = boston.target

# split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# fit quantile regression model
q = 0.5 # example quantile
model = QuantileRegressor(alpha=q)
model.fit(X_train, y_train)

# make predictions on test set
y_pred = model.predict(X_test)

# compute evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

# compute pinball loss
quantiles = [0.1, 0.5, 0.9] # example quantiles
weights = [1, 2, 1] # example weights
total_loss = 0
for i, q in enumerate(quantiles):
    total_loss += pinball_loss(y_test, y_pred, q, weights[i])
pinball = total_loss / sum(weights)

# Write the code to print evaluation metrics here
```

```
# Output
MAE: 3.107693208614685
MSE: 19.109698966683236
RMSE: 4.369178006574176
```

```
Pinball Loss: 2.579571985038757
```

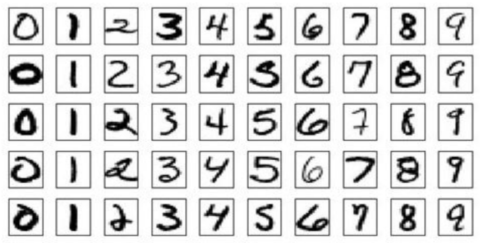
Try: What is the relationship between total household income and the proportion of income that is spent on food? Engel's law is an observation in economics stating that as income rises, the proportion of income spent on food falls, even if absolute expenditure on food rises. Apply quantile regression to these data and determine which food expense can cover 90% of families (for 100 families with a given income) when not interested in the mean food expense.

5. Supervised Learning Algorithms (Classification)

5.1 Identifying the ZIP code from handwriting digits on an envelope.

Here the input is a scan of the handwriting, and the desired output is the actual digits in the zip code. To create a dataset for building a machine learning model, you need to collect many envelopes. Then you can read the zip codes yourself and store the digits as your desired outcomes.

Input: Scan of the handwriting, load the Digits dataset into the notebook



Output: Actual digits in the ZIP code and predict the accuracy of the KNN classifier

Explanation: Recognizing the handwritten digits (0 to 9) using the famous *digits* data set from *Scikit-Learn*, using a classifier called *Logistic Regression*.

Input: Load the digits dataset into the notebook

Hint:

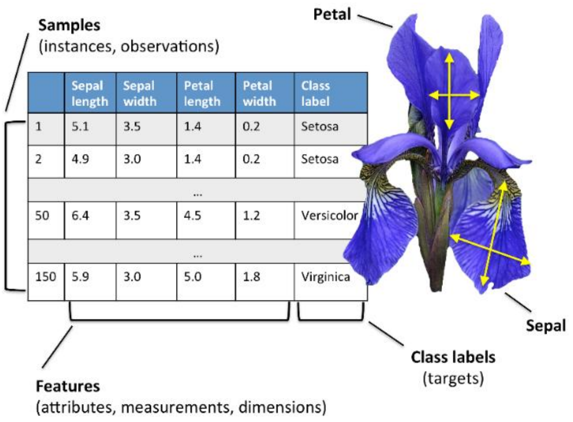
```
import pandas as pd
import numpy as np
from sklearn import svm
from sklearn import datasets
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
%matplotlib inline
     # Write the code to understand the dataset and evaluate the accuracy of a
     classification model
     …
```

Try: Develop a supervised machine learning model to identify the actual digits on a number plate.

5.2 Classifying Iris Species

Let's assume that a hobby botanist is interested in distinguishing the species of some iris flowers that she has found. She has collected some measurements associated with each iris: the length and width of the petals and the length and width of the sepals, all measured in centimeters. She also has the measurements of some irises that have been previously identified by an expert botanist as belonging to the species setosa, versicolor, or virginica. For these measurements, she can be certain of which species each iris belongs to. Let's assume that these are the only species our hobby botanist will encounter in the wild. Our goal is to build a machine learning model that can learn from the measurements of these irises whose species is known, so that we can predict the species for a new iris.

Input: Iris Dataset


Samples
(instances, observations)
Petal
Sepal
Features
(attributes, measurements, dimensions)
Class labels
(targets)

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Output: Different species of irises (3-class classification)

Explanation: In this problem, we want to predict one of several options (the species of iris). This is an example of a classification problem. The possible outputs (different species of irises) are called classes. Every iris in the dataset belongs to one of three classes, so this problem is a three-class classification problem.

Hint:
```
from sklearn.datasets import load_iris
iris_dataset = load_iris()

# Display keys and values
Print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))

# See that array contains measurements for 150 different flowers and inspect the
data.
# Build a machine learning model from this data that can predict the species of iris
for a new set of measurements.
.
.
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
.
.
# Write the code to make the prediction and evaluate the model
```

Try: Conduct the 2-class classification for different datasets and compare the performance of the machine learning algorithm.

5.3 K-Nearest Neighbors

The k in k-nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number k of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. k-nearest neighbors' classifier is easy to understand. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then it assigns the label of this training point to the new data point.

Input: Breast_cancer dataset

Output: Test set accuracy: 0.86

Explanation: The knn object encapsulates the algorithm that will be used to build the model from the training data, as well as the algorithm to make predictions on new data points. It will also hold the information that the algorithm has extracted from the training data. In the case of KNeighbors Classifier, it will just store the training set.

Hint:
```
import required libraries

from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys())) …
print("Shape of cancer data: {}".format(cancer.data.shape))
print("Sample counts per class:\n{}".format(
            {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
print("Feature names:\n{}".format(cancer.feature_names))
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))

# Plot the results of the classifier for values of n-neighbors with 1 and 3.

from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
from sklearn.neighbors import KNeighborsClassifier
 clf = KNeighborsClassifier(n_neighbors=3)
clf.fit(X_train, y_train)

# Write the code to print the test accuracy and the number of correct predictions
```

Try: Produce the code to visualize the decision boundaries for one, three, and nine neighbors

5.4 Naive Bayes Classifiers

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear models. The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in scikit- learn: GaussianNB, BernoulliNB, and MultinomialNB. GaussianNB can be applied to
any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of some-thing, like how often a word appears in a sentence). BernoulliNB and MultinomialNB are mostly used in text data classification.

Input: X = np.array([[0, 1, 0, 1],
                    [1, 0, 1, 1],
                    [0, 0, 0, 1],

```
                    [1, 0, 1, 0]])
    y = np.array([0, 1, 0, 1])
```

Output: Feature counts:

```
    {0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Explanation: Here, we have four data points, with four binary features each. There are two classes, 0 and 1. For class 0 (the first and third data points), the first feature is zero two times and nonzero zero times, the second feature is zero one time and nonzero one time, and so on. These same counts are then calculated for the data points in the second class.

Hint:

```
counts = {}
for label in np.unique(y):
counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

Try: Produce the code to build Gaussian naive Bayes classifiers using Iris data.

5.5 Decision Trees

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.
To distinguish between the following four animals: bears, hawks, penguins, and dolphins. Your goal is to get to the right answer by asking as few if/else questions as possible. You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two. If the answer is "yes," you can ask another question that could help you distinguish between hawks and penguins.

Input: Breast Cancer dataset
We import the dataset and split it into training and a test part. Then we build a model using the default setting of fully developing the tree.

Output: Accuracy on training set: 1.000

```
           Accuracy on test set: 0.937
```

Hint:

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
     cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)

# Write the code to display the training and testing accuracy.
```

Try: Produce the code to build Gaussian naive Bayes classifiers using Iris data.

6. More on Supervised Learning Techniques and its Performance Evaluation

6.1 k-neighbors Regression

There is also a regression variant of the k-nearest neighbors' algorithm. Again, let's start by using the single nearest neighbor, this time using the wave dataset. We've added three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor. The goal of this exercise is to compare the prediction made by nearest k-neighbors regression for different values.

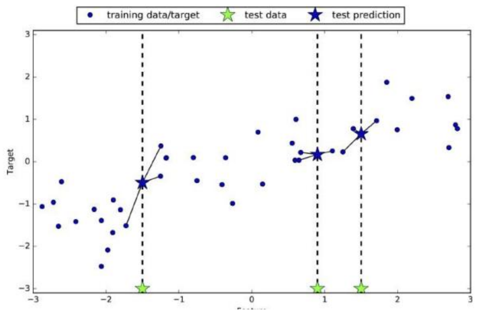
Input: Wave Dataset

Output: Comparing predictions made by nearest neighbors regression for different values of n_neighbors

Explanation:

- In *k-NN regression*, the output is the property value for the object. This value is the average of the values of *k* nearest neighbors. If *k* = 1, then the output is simply assigned to the value of that single nearest neighbor.

```
mglearn.plots.plot_knn_regression(n_neighbors=3)
```



The k-nearest neighbors algorithm for regression is implemented in the KNeighbors Regressor class in scikit-learn. It's used similarly to KNeighborsClassifier:

Hint:

```
from sklearn.neighbors import KNeighborsRegressor
X, y = mglearn.datasets.make_wave(n_samples=40)
# Split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# Instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)

# Fit the model using the training data and training targets
```

Now we can make predictions on the test set:

```
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

Test set predictions: [-0.054 0.357 1.137 -1.894 -1.139 -1.631 0.357 0.912 -0.447 -1.139]

We can also evaluate the model using the score method, which for regressors returns the R 2 score. The R 2 score, also known as the coefficient of determination, is a measure of goodness of a prediction for a regression model and yields a score between 0 and 1. A value of 1 corresponds to a perfect prediction, and a value of 0 corresponds to a constant model that just predicts the mean of the training set responses, y_train:

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

Test set R^2: 0.83

Here, the score is 0.83, which indicates a relatively good model fit.

Try: Analyze the performance of KNeighborsRegressor for one-dimensional dataset, create the testing set consisting of many points on the line and do the comparative analysis for different data points.

6.2 Linear Models for Classification

Let's say we're the owners of a candy store, Willy Wonka's Candy, and we want to do a better job of predicting how much our customers will spend this week, to stock our shelves more appropriately. To get even more specific, let's explore one specific customer named George. George is a 65-year-old mechanic who has children and spent $10 at our store last week. We're going to try to predict the following:

- How much George will spend this week (hint: this is *regression* because it is a dollar amount).
- Whether George will be a "high spender," which we've defined as someone who will spend at least $25 at Willy Wonka's Candy this week (hint: this is a *classification*, because we're predicting a distinct category, high spender or not).

Input: make_blobs dataset

Output: Apply linear and logistic regression and visualize the results.

Explanation:

Linear models are also extensively used for classification. Let's look at binary classification first. In this case, a prediction is made using the following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

Hint:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
X, y = mglearn.datasets.make_forge()
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
for model, ax in zip([LinearSVC(), LogisticRegression()], axes): clf = model.fit(X,
y)
mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5, ax=ax, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
ax.set_title("{}".format(clf.__class__.__name__))
ax.set_xlabel("Feature 0")
ax.set_ylabel("Feature 1")
axes[0].legend()
```

```
mglearn.plots.plot_linear_svc_regularization()
```

Try: Analyze the LinearLogistic in detail using Breast Cancer dataset and compare the results.

6.3 Linear Models for Multi-Class Classification

Many linear classification models are for binary classification only, and don't extend naturally to the multiclass case (except for logistic regression). A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the one-vs.-rest approach. The goal of this exercise is to implement a three-class classification model and visualize the predictions for all the regions.

Input: make_blobs dataset

Output: Visualize the multi-class decision boundaries derived from the three one-vs-rest classifiers

Explanation:

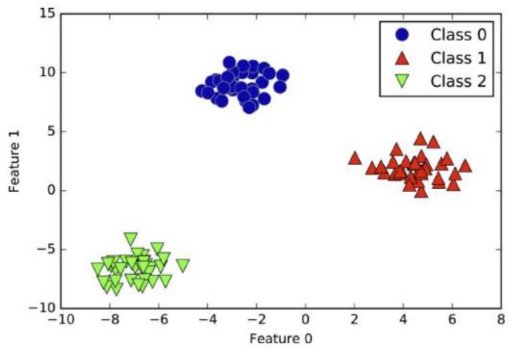
In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all the other classes, resulting in as many binary models as there are classes. To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class "wins," and this class label is returned as the prediction.

Having one binary classifier per class results in having one vector of coefficients (w) and one intercept (b) for each class. The class for which the result of the classification confidence formula given here is highest is the assigned class label:

$$w[0] * x[0] + w[1] * x[1] + \ldots + w[p] * x[p] + b$$

Hint:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```



Now, we train a LinearSVC classifier on the dataset:

Hint:

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

```
# Now, we train a LinearSVC classifier on the dataset:

print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

```
Coefficient shape: (3, 2)
Intercept shape: (3,)
```

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_, ['b',
'r', 'g']):
      plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1', 'Line
class 2'], loc=(1.01, 0.3))
```

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_, ['b', 'r',
'g']):
      plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1', 'Line
class 2'], loc=(1.01, 0.3)) plt.xlabel("Feature 0") plt.ylabel("Feature 1")
```

Try: Consider any dataset and perform the text classification using support vector machine technique.

6.4 Support vector Machine

Support vector machines are used for classification in “Linear models for classification”. support vector machines (often just referred to as SVMs) are an extension that allows for more complex models that are not defined simply by hyper planes in the input space.

Input: Breast Cancer dataset

We import the dataset and split it into training and a test part. Then we build a model using the default setting of fully developing the tree.

Output: Accuracy on training set: 1.000
Accuracy on test set: 0.63

```
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, random_state=0)
svc = SVC()

# Use the fit function here to fit the training datasets

print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

Try: Produce the code to preprocess the data set and to find minimum and maximum for each feature.

6.5 Confusion Matrix

Classification Problems are solved using Supervised Machine learning algorithms. In these problems, our goal is to categories an object using its features. For e.g, identify a fruit using its taste, color and size or check out if a patient has a disease or not using symptoms. Building a model is not a onetime deal, we have to do many experiments and record the output and check the performance of the model on each experiment. The goal of this exercise is to implement the code to evaluate values of various performance metrics that can assess the efficiency of a supervised learning model.

Input: Iris dataset

Output: Confusion matrix with accuracy, precision, and recall values including F1-Score. Obtain the optimal values for the mentioned metrics.

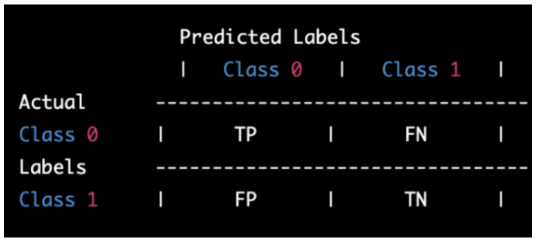
Explanation:

A confusion matrix, also known as an error matrix, is a special table structure that permits visualization of the performance of an algorithm, often a supervised learning one, in the field of machine learning and more precisely the problem of statistical classification!

Precision and recall are two important metrics used in machine learning, information retrieval, and other fields to evaluate the performance of a binary classification model.

The learning objective of precision and recall is to understand how well a model can correctly classify instances of a particular class (positive class) while avoiding false positives and false negatives. Specifically, precision measures the proportion of true positive predictions out of all positive predictions made by the model, while recall measures the proportion of true positive predictions out of all actual positive instances in the dataset.

Here’s an example of a confusion matrix:



```
              Predicted Labels
               |   Class 0   |   Class 1   |
Actual     -------------------------------
Class 0    |      TP     |      FN     |
Labels     -------------------------------
Class 1    |      FP     |      TN     |
```

An example code snippet for calculating the confusion matrix using scikit-learn library with iris data set:

Hint:

```
# Import modules
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

# Load the Iris dataset
iris = load_iris()

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.3, random_state=42)

# Train a K-Nearest Neighbors classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Calculate accuracy, precision, recall, and F1 score

print(f"Accuracy: {acc:.3f}")
print(f"Precision: {prec:.3f}")
print(f"Recall: {rec:.3f}")
print(f"F1 Score: {f1:.3f}")
```

The output of the above code is as below:

# Output:

```
Confusion Matrix:
[[19  0  0]
 [ 0 12  1]
 [ 0  1 12]]
```

Accuracy: 0.956
Precision: 0.959
Recall: 0.956
F1 Score: 0.956

Accuracy is a common metric used in classification tasks to measure the overall correctness of the model's predictions. It is defined as the number of correct predictions divided by the total number of predictions made:

```
# Calculating accuracy

accuracy = (true positives + true negatives) / (true positives + true negatives +
false positives + false negatives)
```

Precision is a metric used in classification tasks to measure the accuracy of positive predictions made by a model. It is defined as the number of true positives divided by the sum of true positives and false positives:

```
# Precision

precision = true positives / (true positives + false positives)
```

Recall is a metric used in classification tasks to measure the ability of a model to correctly identify all positive instances in the data. It is defined as the number of true positives divided by the sum of true positives and false negatives:

```
# Recall

recall = true positives / (true positives + false negatives)
```

F1 score is a metric used in classification tasks that combines precision and recall into a single score. It is the harmonic mean of precision and recall, and is defined as:

```
# F1 score

F1_score = 2 * (precision * recall) / (precision + recall)
```

False Positive Rate (FPR) and True Negative Rate (TNR) are two additional metrics commonly used in binary classification tasks, especially when dealing with imbalanced datasets.

False Positive Rate (FPR) is the proportion of negative instances that were incorrectly classified as positive by the model. It is calculated as:

```
# FPR

FPR = false positives / (true negatives + false positives)
```

In other words, FPR measures the rate at which the model falsely predicts the positive class when the true class is negative.

True Negative Rate (TNR), also known as specificity, is the proportion of negative instances that were correctly classified as negative by the model. It is calculated as:

```
# TNR

TNR = true negatives / (true negatives + false positives)
```

In other words, TNR measures the rate at which the model correctly predicts the negative class when the true class is negative.

Both FPR and TNR are important metrics to consider when evaluating the performance of a binary classifier, especially when the negative class is the minority class in an imbalanced dataset. A good model should have a low FPR and a high TNR, indicating that it is able to correctly identify negative instances while minimizing the number of false positives.

Try: Evaluate the performance of any classification model and analyze how good the model is in predicting the test data.

6.6 Controlling Complexity of Decision Trees

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. The presence of pure leaves means that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class. The goal of this exercise is to implement strategies to prevent overfitting using DecisionTreeClassifier and DecisionTreeRegressor.

Input: Breast Cancer Dataset

Output: Build a model using the default setting of fully developing the tree. Obtain the training and testing accuracy more than 95%.

Explanation:

There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called pre-pruning) or building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning). Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

Hint:

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()
X_train,  X_test,  y_train,  y_test  =  train_test_split(  cancer.data,  cancer.target,
stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0) tree.fit(X_train, y_train)

# Write the code here to display the training and testing accuracies
```

Try: Implement the code for at least three different datasets and compare the results.

7. Ensemble Learning Algorithms

7.1 Ensemble Methods

Ensembles are methods that combine multiple machine learning models to create more powerful models. There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building blocks: random forests and gradient boos-ted decision trees.

The gradient boosted regression tree is another ensemble method that combines multiple decision trees to create a more powerful model. Despite the "regression" in the name, these models can be used for regression and classification. In contrast to the random forest approach, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one.

Input: Breast Cancer dataset

As the training set accuracy is 100%, we are likely to be overfitting. To reduce overfit- ting, we could either apply stronger pre-pruning by limiting the maximum depth or lower the learning rate:

Output: Accuracy on training set: 1.000
Accuracy on test set: 0.958

Hint:
```
from sklearn.ensemble import GradientBoostingClassifier
X_train, X_test, y_train, y_test = train_test_split( cancer.data, cancer.target, random_state=0)

# Use the GradientBoostingClassifier with a random state of 0 and use the fit function on the training dataset before displaying the training and testing accuracies.

print("Accuracy on training set: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbrt.score(X_test, y_test)))
```

Try: Produce the code to overcome the overfitting problem.

7.2 Random forests

The idea behind random forests is that each tree might do a relatively good job of predicting but will likely be overfit on part of the data. If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

There are two ways in which the trees in a random forest are randomized: by selecting the data points used to build a tree and by select-ing the features in each split test.

Input: Breast Cancer dataset
We import the dataset and split it into training and a test part. Then we build a model using the default setting of fully developing the tree.

Output: Accuracy on training set: 1.000

Accuracy on test set: 0.972

Hint:

```
from sklearn.tree import DecisionTreeClassifier
cancer = load_breast_cancer()

# Perform the dataset split process and apply the DecisionTreeClassifier with a random
state 0.

tree.fit(X_train, y_train)
print("Accuracy   on   training   set:   {:.3f}".format(tree.score(X_train,
y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test))
```

Try: Produce a code to compare the predictions of the decision tree and the linear regression model.

7.3 AdaBoost for Binary Classification

AdaBoost (Adaptive Boosting) combines multiple weak learners (typically decision stumps) and adjusts their weights iteratively to minimize classification error. It's robust to noisy data and works well on structured datasets.

Input: Iris Dataset (use only 2 classes for binary classification)

Output: Classification report and decision boundary visualization (optional)

Hint:

```
# import libraries

from transformers import pipelinefrom sklearn.ensemble import AdaBoostClassifier

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report

# Load dataset

data = load_iris()

X = data.data[data.target != 2]

y = data.target[data.target != 2]

# Split data

# Train AdaBoost model

# Predict
# Evaluation
```

Try: Change the base estimator of AdaBoostClassifier to DecisionTreeClassifier (max_depth=2). Compare the performance with the default decision stump (depth=1). How does tree depth affect bias and variance?

7.4 Voting Classification (Hard vs Soft Voting)

Voting Classifier combines multiple models (e.g., logistic regression, decision tree, and SVM) and predicts the class based on majority vote (hard voting) or averaged probabilities (soft voting). It's a simple and powerful ensemble strategy.

Input: Iris Dataset (3-class classification)

Output: Classification report using hard and soft voting

Hint:

```
# import libraries

from sklearn.ensemble import VotingClassifier

from sklearn.linear_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.svm import SVC

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report

# Load data

iris = load_iris()

X, y = iris.data, iris.target

# Split data

# Base learners

# Voting classifiers

# Train & evaluate
```

Try: Remove one of the models (e.g., Decision Tree) from the voting ensemble. Evaluate how it affects the overall accuracy of both hard and soft voting. Which individual model has the most influence on ensemble performance?

8. Unsupervised Learning Algorithms

8.1 Preparing data for unsupervised learning.

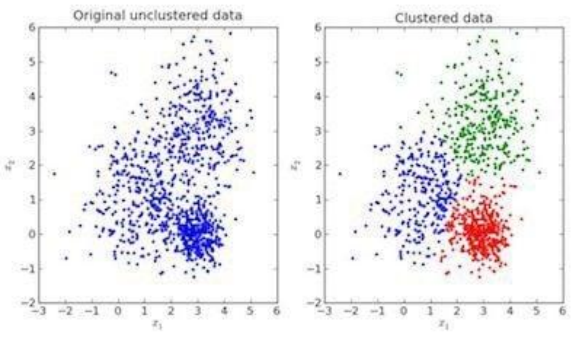
There is also a regression variant of the k-nearest neighbors' algorithm. Let's start by using the single nearest neighbor, this time using the wave dataset. There are three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor.

Input: The Iris dataset

Output: Predict which class the iris belongs with higher accuracy.

Explanation:

In clustering, the data is divided into several groups with similar traits.



In the image above, the left is raw data without classification, while the right is clustered based on its features. When an input is given which is to be predicted then it checks in the cluster it belongs to which class based on its features, and the prediction is made.

We use the scikit-learn library in Python to load the Iris dataset and matplotlib for data visualization.

Hint:

```
# Importing Modules

from sklearn import datasets
import matplotlib.pyplot as plt

# Loading dataset
# Available methods on dataset

print(dir(iris_df))
# Features

print(iris_df.feature_names)

# Write the code to display the Targets and Target Names

# Dataset Slicing
x_axis = iris_df.data[:, 0]  # Sepal Length
```

```
y_axis = iris_df.data[:, 2]  # Sepal Width


# Plotting
plt.scatter(x_axis, y_axis, c=iris_df.target)
plt.show()
```

Try: Select at least three different datasets and do the comparative analysis

8.2 K-Means Clustering

Every Machine Learning engineer wants to achieve accurate predictions with their algorithms. K-Means clustering is one of the unsupervised algorithms where the available input data does not have a labeled response. Clustering is a type of unsupervised learning wherein data points are grouped into different sets based on their degree of similarity. The goal of this exercise is to find the optimal number of clusters using the elbow method.

Input: Mall_Customers_data.csv dataset

Output: Find the optimal number of clusters and visualize each cluster with a different color.

Explanation:
K-Means clustering is an unsupervised learning algorithm. There is no labeled data for this clustering, unlike in supervised learning. K-Means performs the division of objects into clusters that share similarities and are dissimilar to the objects belonging to another cluster.
The term 'K' is a number. You need to tell the system how many clusters you need to create. For example, K = 2 refers to two clusters. There is a way of finding out what is the best or optimum value of K for a given data.

1. **Data Pre-Processing**. Import the libraries, datasets, and extract the independent variables.

Hint:
```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('Mall_Customers_data.csv')
x = dataset.iloc[:, [3, 4]].values
```

2. Find the optimal number of clusters using the elbow method. Here's the code you use:

Hint:
```
# Write the code to find optimal number of clusters using the elbow method
from sklearn.cluster import KMeans
wcss_list= []

#Initializing the list for the values of WCSS

#Using for loop for iterations from 1 to 10.
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', random_state= 42)
    kmeans.fit(x)
    wcss_list.append(kmeans.inertia_)
mtp.plot(range(1, 11), wcss_list)
mtp.title('The Elobw Method Graph')
mtp.xlabel('Number of clusters(k)')
mtp.ylabel('wcss_list')
mtp.show()
```

3. Train the K-means algorithm on the training dataset. Use the same two lines of code used in the previous section. However, instead of using i, use 5, because there are 5 clusters that need to be formed.

Hint:

```
#training the K-means model on a dataset
kmeans = KMeans(n_clusters=5, init='k-means++', random_state= 42)
y_predict= kmeans.fit_predict(x)
```

4. Visualize the Clusters. Since this model has five clusters, we need to visualize each one.

Hint:

```
# Write the code to visualize the clusters
```

Try: Select any dataset that belongs to their scores and categorize them into grades like A, B, and C.

8.3 Gaussian Mixture Models

In K-Means, we do what is called "hard labeling", where we simply add the label of the maximum probability. However, certain data points that exist at the boundary of clusters may simply have similar probabilities of being on either cluster. In such circumstances, we look at all the probabilities instead of the max probability. This is known as "soft labeling". The goal of this exercise is to implement the clustering technique to perform the 'hard labelling'.

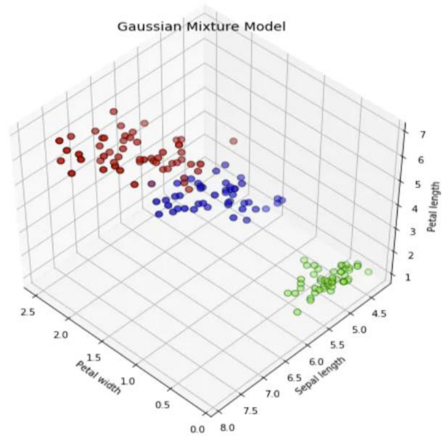
Input: Iris dataset

Output: Visualize the different clusters with different colors.

Hint:

```
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
%matplotlib inline

# Write the code here to apply the Gaussian Mixture model and fit it for 3 components.

proba_lists = gmm.predict_proba(X)#Plotting
colored_arrays = np.matrix(proba_lists)
colored_tuples = [tuple(i.tolist()[0]) for i in colored_arrays]
fig = plt.figure(1, figsize=(7,7))
ax = Axes3D(fig, rect=[0, 0, 0.95, 1], elev=48, azim=134)
ax.scatter(X[:, 3], X[:, 0], X[:, 2],
          c=colored_tuples, edgecolor="k", s=50)
```

```
ax.set_xlabel("Petal width")
ax.set_ylabel("Sepal length")
ax.set_zlabel("Petal length")
plt.title("Gaussian Mixture Model", fontsize=14)
```



Explanation:
For the above Gaussian Mixture Model, the colors of the datapoints are based on the Gaussian probability of being near the cluster. The RGB values are based on the nearness to each of the red, blue, and green clusters. If you look at the datapoints near the boundary of the blue and red cluster, you shall see purple, indicating the datapoints are close to either cluster.

Try: Prove experimentally how Gaussian Discriminant analysis is more robust than logistic regression with a limited volume of data.

8.4 Hidden Markov Model

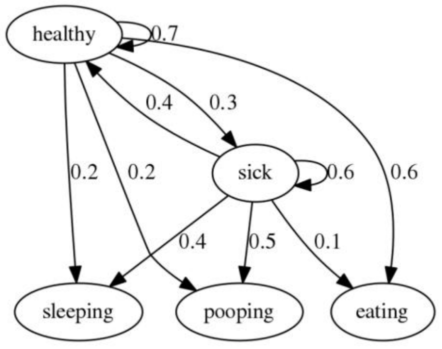
In a Markov Model, we look for states and the probability of the next state given the current state. The goal of this exercise is to implement hidden Markov model and perform the clustering.

Input: Boston Dataset

Output: Visualize the performance of HMM model

Explanation:

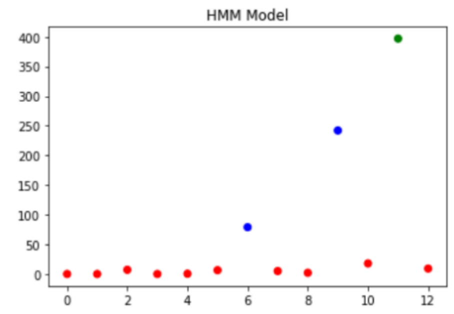
An example below is of a dog's life in Markov Model.



Let's assume the dog is sick. Given the current state, there is a 0.6 chance it will continue being sick the next hour, 0.4 that it is sleeping, 05 pooping, 0.1 eating and 0.4 that it will be healthy again. In an HMM, you provide how many states there may be inside the timeseries data for the model to compute. An example of the Boston house prices dataset is given below with 3 states.

Hint:

```
from hmmlearn import hmm
import numpy as np
%matplotlib inline

from sklearn import datasets#Data
boston = datasets.load_boston()
ts_data = boston.data[1,:]#HMM Model
gm = hmm.GaussianHMM(n_components=3)
gm.fit(ts_data.reshape(-1, 1))
states = gm.predict(ts_data.reshape(-1, 1))#Plot
color_dict = {0:"r",1:"g",2:"b"}
color_array = [color_dict[i] for i in states]
plt.scatter(range(len(ts_data)), ts_data, c=color_array)
plt.title("HMM Model")
```



Try: Let's say that we want to model a word BOOK. We investigate data set and notice that there are three time series corresponding to BOOK. We'll base our HMM on those three examples. One important step to take before training is deciding on the number of states. We can do that empirically. If we look at the plot of one of data points of the word BOOK we can conclude that there are three sequences through which speaker's hands are transitioning.

8.5 Hierarchical Clustering

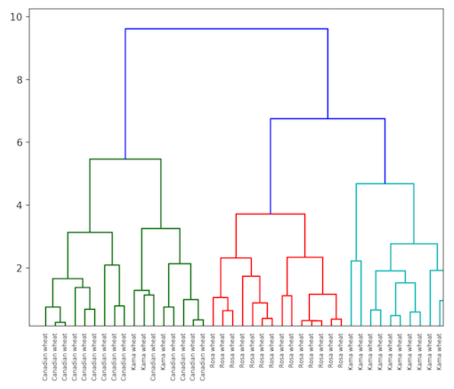
Suppose Walmart has collected customer data based on past transactions such as customer, gender, age, annual income, spending score, and shopped item category. With the availability of all these parameters, Walmart's marketing team has sufficient information to explain customers' spending habits. Now image Walmart is launching a campaign targeting customers interested in luxurious items. They have special offers to attract them to the store but extending them to all customers will not make sense since not all customers are interested in luxurious items. The goal of this exercise is to explore the potential of clustering algorithms to accomplish the above task.

Input: seed_less_rows dataset

Output: Visualize the clustering process using a dendrogram.

Explanation:
As its name implies, hierarchical clustering is an algorithm that builds a hierarchy of clusters. This algorithm begins with all the data assigned to a cluster, then the two closest clusters are joined into the same cluster. The algorithm ends when only a single cluster is left. The completion of hierarchical clustering can be shown using dendrogram. Now let's look at an example of hierarchical clustering using grain data. The dataset can be found here**.**

Hint:
```
# Import the Modules like cluster hierarchy, linkage, and dendrogram

import matplotlib.pyplot as plt
import pandas as pd

# Reading the DataFrame

seeds_df = pd.read_csv(
    "https://raw.githubusercontent.com/vihar/unsupervised-learning-with-
python/master/seeds-less-rows.csv")

# Remove the grain species from the DataFrame, save for later

varieties = list(seeds_df.pop('grain_variety'))

# Extract the measurements as a NumPy array

samples = seeds_df.values

"""

Perform hierarchical clustering on samples using the linkage() function with the
method='complete' keyword argument. Assign the result to mergings.
"""
mergings = linkage(samples, method='complete')

"""
Plot a dendrogram using the dendrogram() function on mergings,
specifying the keyword arguments labels=varieties, leaf_rotation=90,
and leaf_font_size=6.
"""

dendrogram(mergings,
           labels=varieties,
           leaf_rotation=90,
           leaf_font_size=6,
           )

plt.show()
```



Try: Hierarchical clustering requires both a distance and linkage method. Make use of euclidean distance and the Ward linkage method and attempts to minimize the variance between clusters.

8.6 DBSCAN Clustering

Suppose we have an e-commerce, and we want to improve our sales by recommending relevant products to our customers. We don't know exactly what our customers are looking for but based on a data set we can predict and recommend a relevant product to a specific customer. The goal of this exercise is to apply the DBSCAN to our data set (based on the e-commerce database) and find clusters based on the products that the users have bought.

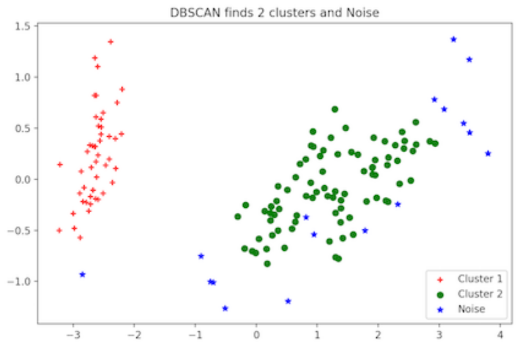
Input: Customer/Iris Dataset

Output: Visualization showing the implementation of DBSCAN Clustering

Explanation:
Density-based spatial clustering of applications with noise, or DBSCAN, is a popular clustering algorithm used as a replacement for k-means in predictive analytics. To run it doesn't require an input for the number of clusters, but it does need to tune two other parameters.

Hint:
```
# Importing Modules

from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA

# Load Dataset

iris = load_iris()

# Declaring Model

dbscan = DBSCAN()

# Fitting
```

```
dbscan.fit(iris.data)

# Perform the transformation using PCA


# Plot based on Class
for i in range(0, pca_2d.shape[0]):
    if dbscan.labels_[i] == 0:
        c1 = plt.scatter(pca_2d[i, 0], pca_2d[i, 1], c='r', marker='+')
    elif dbscan.labels_[i] == 1:
        c2 = plt.scatter(pca_2d[i, 0], pca_2d[i, 1], c='g', marker='o')
    elif dbscan.labels_[i] == -1:
        c3 = plt.scatter(pca_2d[i, 0], pca_2d[i, 1], c='b', marker='*')


plt.legend([c1, c2, c3], ['Cluster 1', 'Cluster 2', 'Noise'])
plt.title('DBSCAN finds 2 clusters and Noise')
plt.show()
```



Try: use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high.

9. Dimensionality Reduction Techniques

9.1 Algorithm Optimization using PCA

While you can speed up the fitting of a machine learning algorithm by changing the optimization algorithm, a more common way to speed up the algorithm is to use principal component analysis (PCA). If you're learning algorithm is too slow because the input dimension is too high, then using PCA to speed it up can be a reasonable choice. The goal of this exercise is to use PCA technique and improve the performance of an algorithm.

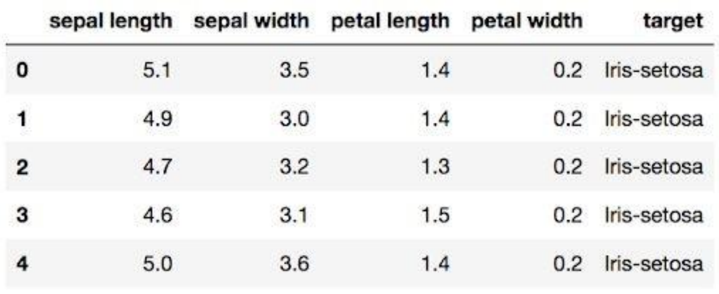
Input: Iris Dataset

Output: Visualize the 2D projection with two components of PCA graph.

Explanation:
Principal component analysis (PCA) is a method of reducing the dimensionality of data and is used to improve data visualization and speed up machine learning model training.
To understand the value of using PCA for data visualization, the first part of this tutorial post goes over a basic visualization of the Iris data set after applying PCA. The second part explores how to use PCA to speed up a machine learning algorithm (logistic regression) on the Modified National Institute of Standards and Technology (MNIST) data set.

STEP 1: LOAD THE IRIS DATA SET

Hint:
```
import pandas as pd

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

# load dataset into Pandas DataFrame

df = pd.read_csv(url, names=['sepal length','sepal width','petal length','petal
width','target'])
```

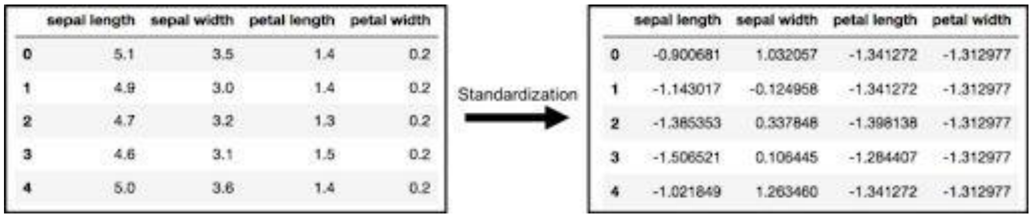


	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

STEP 2: STANDARDIZE THE DATA

Hint:
```
from sklearn.preprocessing import StandardScaler
features = ['sepal length', 'sepal width', 'petal length', 'petal width']
# Separating out the features
```

```python
x = df.loc[:, features].values

# Separate the target and standardize the features
```


Standardization

	sepal length	sepal width	petal length	petal width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

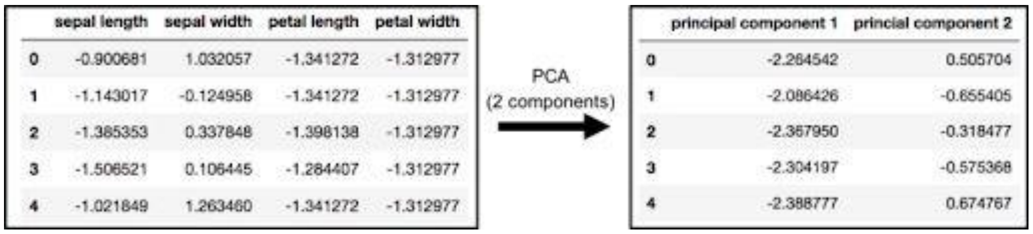
	sepal length	sepal width	petal length	petal width
0	-0.900681	1.032057	-1.341272	-1.312977
1	-1.143017	-0.124958	-1.341272	-1.312977
2	-1.385353	0.337848	-1.398138	-1.312977
3	-1.506521	0.106445	-1.284407	-1.312977
4	-1.021849	1.263460	-1.341272	-1.312977

The array x (visualized by a pandas dataframe) before and after standardization. | Image: Michael Galarnyk.

STEP 3: PCA PROJECTION TO 2D

Hint:

```python
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents
             , columns = ['principal component 1', 'principal component 2'])
```


PCA
(2 components)

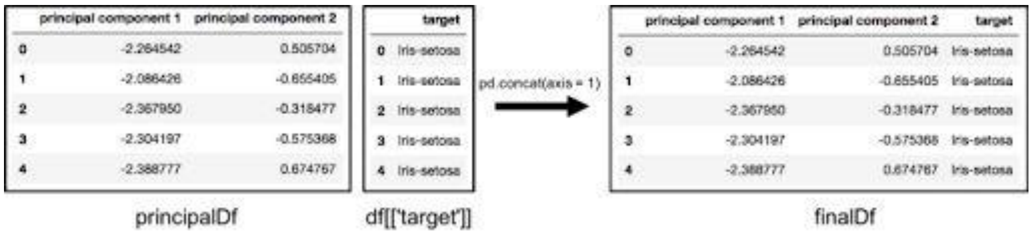
	sepal length	sepal width	petal length	petal width
0	-0.900681	1.032057	-1.341272	-1.312977
1	-1.143017	-0.124958	-1.341272	-1.312977
2	-1.385353	0.337848	-1.398138	-1.312977
3	-1.506521	0.106445	-1.284407	-1.312977
4	-1.021849	1.263460	-1.341272	-1.312977

	principal component 1	princial component 2
0	-2.264542	0.505704
1	-2.086426	-0.655405
2	-2.367950	-0.318477
3	-2.304197	-0.575368
4	-2.388777	0.674767

PCA and keeping the top two principal components

```python
finalDf = pd.concat([principalDf, df[['target']]], axis = 1)
```

Concatenating DataFrame along axis = 1. finalDf is the final DataFrame before plotting the data.


pd.concat(axis = 1)
principalDf
df[['target']]
finalDf

	principal component 1	principal component 2
0	-2.264542	0.505704
1	-2.086426	-0.655405
2	-2.367950	-0.318477
3	-2.304197	-0.575368
4	-2.388777	0.674767

	target
0	Iris-setosa
1	Iris-setosa
2	Iris-setosa
3	Iris-setosa
4	Iris-setosa

	principal component 1	principal component 2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa

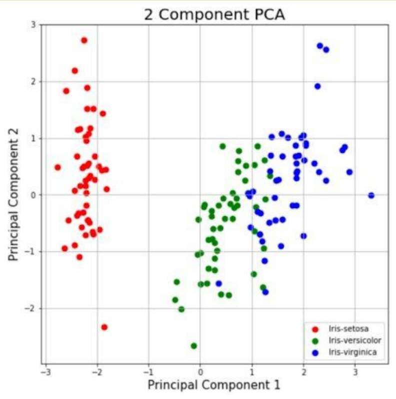
Concatenating DataFrames along columns to make finalDf before graphing. | Image: Michael Galarnyk

STEP 4: VISUALIZE 2D PROJECTION

Hint:

```python
fig = plt.figure(figsize = (8,8))
```

```
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 component PCA', fontsize = 20)

targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(targets,colors):
    indicesToKeep = finalDf['target'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(targets)
ax.grid()
```



A two component PCA graph. | Image: Michael Galarnyk

Try: Apply the same technique on MNIST dataset and compare the results.

9.2 T-SNE Clustering

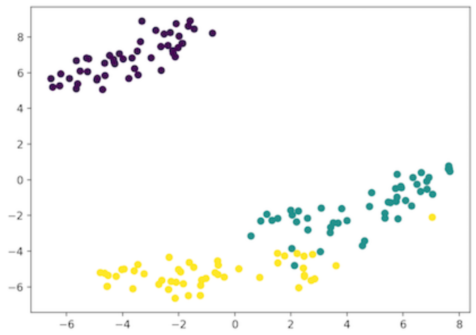
Imagine you get a dataset with hundreds of features (variables) and have little understanding about the domain the data belongs to. You are expected to identify hidden patterns in the data, explore and analyze the dataset. And not just that, you must find out if there is a pattern in the data – is it signal or is it just noise? The goal of this exercise is to prove that t-SNE algorithm is a good fit for dimensionality reduction.

Input: Iris Dataset
Output: Visualization showing the four features of Iris dataset by applying t-SNE model

Explanation:
One of the unsupervised learning methods for visualization is **t-distributed stochastic neighbor embedding, or t-SNE.** It maps high-dimensional space into a two or three-dimensional space which can then be visualized. Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability.

Hint:
```
# Importing Modules
```

```
from sklearn import datasets
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Loading dataset
iris_df = datasets.load_iris()

# Define and Fit the model here

# Plotting 2d t-Sne
x_axis = transformed[:, 0]
y_axis = transformed[:, 1]

plt.scatter(x_axis, y_axis, c=iris_df.target)
plt.show()
```



Violet: Setosa, Green: Versicolor, Yellow: Virginica

Here, the Iris dataset has four features (4d) and is transformed and represented in the two-dimensional figure. Similarly, t-SNE model can be applied to a dataset which has n-features.

Try: Make use of Scikit-learn implementation of PCA and perform the dimensionality reduction.

9.3 Linear Discriminant Analysis for Supervised Dimensionality Reduction

Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique that maximizes class separability by projecting data in a lower-dimensional space based on label information.

Input: Iris Dataset

Output: 2D scatter plot with maximized class separability

Hint:

```
# import libraries

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.datasets import load_iris

import matplotlib.pyplot as pltfrom sklearn.svm import SVC
```

```
# Load data

data = load_iris()

X, y = data.data, data.target

# Apply LDA

# Visualize
plt.figure(figsize=(8,6))
scatter = plt.scatter(X_lda[:,0], X_lda[:,1], c=y, cmap='rainbow')
```

Try: Compare PCA vs LDA projections on the same Iris dataset. Which method shows clearer class separation, and why might that be?

10. Semi supervised Learning Techniques

10.1 Label Propagation in Semi Supervised Learning

Predict customer interests based on the information about other customers. Here, you can apply the variation of continuity assumption — if two people are connected on social media, for example, it's highly likely that they will share similar interests. The goal of this exercise is to apply the label propagation algorithm and represent the labeled and unlabeled data in the form of graphs.

Input: Iris Dataset

Output: Improve the accuracy of a machine learning model by leveraging both labeled and unlabeled data.

Explanation:

Semi-supervised learning is a type of machine learning that combines both labeled and unlabeled data to improve the accuracy of a model. In traditional supervised learning, a large amount of labeled data is required for training a model, whereas in unsupervised learning, only unlabeled data is used.

In this example, we will use a small subset of the iris dataset for labeled data and the remaining data as unlabeled data.

Hint:

```
# Import necessary modules
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import accuracy_score

# Load the iris dataset
iris = load_iris()

# Split the data into labeled and unlabeled subset. Later create the
Label Spreading model
X_labeled, X_unlabeled, y_labeled, y_unlabeled =
train_test_split(iris.data, iris.target, test_size=0.8,
stratify=iris.target)


# Fit the model using both labeled and unlabeled data
model.fit(X_labeled, y_labeled)

# Predict labels for the unlabeled data
y_pred = model.predict(X_unlabeled)

# Compute the accuracy of the model and print the same
```

Yields below output.

```
# Output:
Accuracy: 0.975
```

Try: Improve the accuracy of a machine learning model by leveraging both labeled and unlabeled data for different split ratios.

10.2 Label Propagation Algorithm

A popular approach to semi-supervised learning is to create a graph that connects examples in the training dataset and propagate known labels through the edges of the graph to label unlabeled examples. The goal of this exercise is to implement semi-supervised learning, the label propagation algorithm for classification predictive modeling.

Input: Define your own dataset
Output: Achieve an optimal classification accuracy

Explanation:
Label Propagation is a semi-supervised learning algorithm. The algorithm was proposed in the 2002 technical report by Xiaojin Zhu and Zoubin Ghahramani titled "Learning from Labeled and Unlabeled Data with Label Propagation." The intuition for the algorithm is that a graph is created that connects all examples (rows) in the dataset based on their distance, such as Euclidean distance. Nodes in the graph then have label soft labels or label distribution based on the labels or label distributions of examples connected nearby in the graph.

Hint:
```
# define dataset

X, y = make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0, random_state=1)
```

Next, we will split the dataset into train and test datasets with an equal 50-50 split (e.g. 500 rows in each).

Hint:
```
# split into train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50,
random_state=1, stratify=y)
```

Finally, we will split the training dataset in half again into a portion that will have labels and a portion that we will pretend is unlabeled.

Hint:
```
...

# split train into labeled and unlabeled
```

Tying this together, the complete example of preparing the semi-supervised learning dataset is listed below.

Hint:

```
# prepare semi-supervised learning dataset

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# define dataset

X, y = make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0, random_state=1)

# split into train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50,
random_state=1, stratify=y)

# split train into labeled and unlabeled

X_train_lab, X_test_unlab, y_train_lab, y_test_unlab = train_test_split(X_train,
y_train, test_size=0.50, random_state=1, stratify=y_train)

# Summarize training set and testing set size
```

Running the example prepares the dataset and then summarizes the shape of each of the three portions.
The results confirm that we have a test dataset of 500 rows, a labeled training dataset of 250 rows, and 250 rows of unlabeled data.

```
Labeled Train Set: (250, 2) (250,)
Unlabeled Train Set: (250, 2) (250,)
Test Set: (500, 2) (500,)
```

In this case, we will use a logistic regression algorithm fit on the labeled portion of the training dataset.

Hint:

```
...

# define model
model = LogisticRegression()

# fit model on labeled dataset
model.fit(X_train_lab, y_train_lab)
```

The model can then be used to make predictions on the entire hold out test dataset and evaluated using classification accuracy.

Hint:

```
...

# make predictions on hold out test set
```

```
yhat = model.predict(X_test)

# calculate score for test set
score = accuracy_score(y_test, yhat)

# summarize score
print('Accuracy: %.3f' % (score*100))
```

Tying this together, the complete example of evaluating a supervised learning algorithm on the semi-supervised learning dataset is listed below.

Hint:

```
# baseline performance on the semi-supervised learning dataset

from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression

# define dataset

X,   y   =   make_classification(n_samples=1000,   n_features=2,   n_informative=2,
n_redundant=0, random_state=1)

# split into train and test

X_train,  X_test,  y_train,  y_test  =  train_test_split(X,  y,  test_size=0.50,
random_state=1, stratify=y)

# split train into labeled and unlabeled

X_train_lab,  X_test_unlab,  y_train_lab,  y_test_unlab  =  train_test_split(X_train,
y_train, test_size=0.50, random_state=1, stratify=y_train)

# define model

model = LogisticRegression()

# fit model on labeled dataset

model.fit(X_train_lab, y_train_lab)

# make predictions on hold out test set
yhat = model.predict(X_test)

# calculate score for test set
score = accuracy_score(y_test, yhat)

# summarize score
print('Accuracy: %.3f' % (score*100))
```

Running the algorithm fits the model on the labeled training dataset and evaluates it on the holdout dataset and prints the classification accuracy.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
In this case, we can see that the algorithm achieved a classification accuracy of about 84.8 percent.
We would expect an effective semi-supervised learning algorithm to achieve better accuracy than this.

```
Accuracy: 84.800
```

Label Propagation for Semi-Supervised Learning

Hint:

```
...

# define model
model = LabelPropagation()

# fit model on training dataset

model.fit(..., ...)

# make predictions on hold out test set

yhat = model.predict(...)
```

```
...

# get labels for entire training dataset data

tran_labels = model.transduction_
```

```
...

# create the training dataset input

X_train_mixed = concatenate((X_train_lab, X_test_unlab))
```

We can then create a list of -1 valued (unlabeled) for each row in the unlabeled portion of the training dataset.

```
...

# create "no label" for unlabeled data

nolabel = [-1 for _ in range(len(y_test_unlab))]
```

This list can then be concatenated with the labels from the labeled portion of the training dataset to correspond with the input array for the training dataset.

```
...
```

```
# recombine training dataset labels

y_train_mixed = concatenate((y_train_lab, nolabel))
```

We can now train the *LabelPropagation* model on the entire training dataset.

Hint:

```
...

# define model

model = LabelPropagation()

# fit model on training dataset

model.fit(X_train_mixed, y_train_mixed)
```

Next, we can use the model to make predictions on the holdout dataset and evaluate the model using classification accuracy.

Hint:

```
# make predictions on hold out test set
yhat = model.predict(X_test)

# calculate score for test set
score = accuracy_score(y_test, yhat)

# summarize score
print('Accuracy: %.3f' % (score*100))
```

Tying this together, the complete example of evaluating label propagation on the semi-supervised learning dataset is listed below.

Hint:

```
# evaluate label propagation on the semi-supervised learning dataset

from numpy import concatenate
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.semi_supervised import LabelPropagation

# define dataset

X, y = make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0, random_state=1)

# split into train and test
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50,
random_state=1, stratify=y)

# split train into labeled and unlabeled

X_train_lab, X_test_unlab, y_train_lab, y_test_unlab = train_test_split(X_train,
y_train, test_size=0.50, random_state=1, stratify=y_train)

# create the training dataset input

X_train_mixed = concatenate((X_train_lab, X_test_unlab))

# create "no label" for unlabeled data

nolabel = [-1 for _ in range(len(y_test_unlab))]

# recombine training dataset labels

y_train_mixed = concatenate((y_train_lab, nolabel))

# Define the model, fit it to the training dataset and make the predictions on hold
out test set.


# calculate score for test set
score = accuracy_score(y_test, yhat)

# summarize score
print('Accuracy: %.3f' % (score*100))
```

Running the algorithm fits the model on the entire training dataset and evaluates it on the holdout dataset and prints the classification accuracy.

Try: Apply the label propagation technique on a different dataset and compare the results to interpret the difference.

10.3 Estimate labels for the training dataset

Consider the following problem of given sets of unlabeled observations, each set with known label proportions, predict the labels of another set of observations, possibly with known label proportions. This problem occurs in areas like e-commerce, politics, spam filtering and improper content detection. We present consistent estimators which can reconstruct the correct labels with high probability in a uniform convergence sense. The goal of this exercise is to generate the estimate labels for the training set using consistent estimators.

Input: Any unlabeled dataset

Output: Labeled dataset

Explanation:
We can then use these labels along with all the input data to train and evaluate a supervised learning algorithm, such as a logistic regression model. The hope is that the supervised learning model fit on the

entire training dataset would achieve even better performance than the semi-supervised learning model alone.

Hint:

```
...

# define supervised learning model

model2 = LogisticRegression()

# fit supervised learning model on entire training dataset

model2.fit(X_train_mixed, tran_labels)

# make predictions on hold out test set

yhat = model2.predict(X_test)

# calculate score for test set

score = accuracy_score(y_test, yhat)

# summarize score

print('Accuracy: %.3f' % (score*100))
```

Tying this together, the complete example of using the estimated training set labels to train and evaluate a supervised learning model is listed below.

Hint:

```
# Evaluate logistic regression fit on label propagation for semi-supervised
learning

from numpy import concatenate
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.semi_supervised import LabelPropagation
from sklearn.linear_model import LogisticRegression

# define dataset

X, y = make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0, random_state=1)

# split into train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.50,
random_state=1, stratify=y)

# split train into labeled and unlabeled
```

```
X_train_lab, X_test_unlab, y_train_lab, y_test_unlab = train_test_split(X_train, 
y_train, test_size=0.50, random_state=1, stratify=y_train)

# create the training dataset input

X_train_mixed = concatenate((X_train_lab, X_test_unlab))

# create "no label" for unlabeled data

nolabel = [-1 for _ in range(len(y_test_unlab))]

# Recombine training dataset labels

y_train_mixed = concatenate((y_train_lab, nolabel))

# define model

model = LabelPropagation()

# fit model on training dataset

model.fit(X_train_mixed, y_train_mixed)

# Get labels for entire training dataset data and define the supervised learning 
model.

# Fit supervised learning model on entire training dataset

model2.fit(X_train_mixed, tran_labels)

# Make predictions on hold out test set

yhat = model2.predict(X_test)

# Calculate score for test set

score = accuracy_score(y_test, yhat)

# Summarize score

print('Accuracy: %.3f' % (score*100))
```

Running the algorithm fits the semi-supervised model on the entire training dataset, then fits a supervised learning model on the entire training dataset with inferred labels and evaluates it on the holdout dataset, printing the classification accuracy.

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.
In this case, we can see that this hierarchical approach of the semi-supervised model followed by supervised model achieves a classification accuracy of about 86.2 percent on the holdout dataset, even better than the semi-supervised learning used alone that achieved an accuracy of about 85.6 percent.

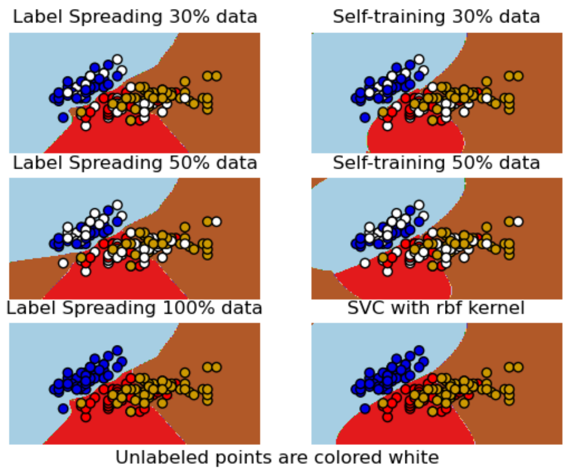
```
Accuracy: 86.200
```

Try: Achieve better results by tuning the hyperparameters of the LabelPropogation Model.

10.4 Decision Boundary of Semi-Supervised Classifier Vs SVM

The decision boundaries generated by SVM, Label Spreading, and Self-training on a dataset of iris were compared. The results show that these methods can learn good boundaries even if there are only a few labeled pieces of information. Note that training with 100% of the data is not included since it is functionally identical to doing so with the SVC. The goal of this exercise is to demonstrate the label spreading and self-training can learn good boundaries.

Input: Iris Dataset
Output: Obtain 100% with self-trained data and SVC
Explanation:
A comparison for the decision boundaries generated on the iris dataset by Label Spreading, Self-training and SVM. This example demonstrates that Label Spreading and Self-training can learn good boundaries even when small amounts of labeled data are available. Note that Self-training with 100% of the data is omitted as it is functionally identical to training the SVC on 100% of the data.



Hint:
```
import matplotlib.pyplot as plt
import numpy as np

from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading, SelfTrainingClassifier
from sklearn.svm import SVC

iris = datasets.load_iris()

X = iris.data[:, :2]
y = iris.target

# step size in the mesh
h = 0.02
```

```python
rng = np.random.RandomState(0)
y_rand = rng.rand(y.shape[0])
y_30 = np.copy(y)
y_30[y_rand < 0.3] = -1  # set random samples to be unlabeled
y_50 = np.copy(y)
y_50[y_rand < 0.5] = -1
# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
ls30 = (LabelSpreading().fit(X, y_30), y_30, "Label Spreading 30% data")
ls50 = (LabelSpreading().fit(X, y_50), y_50, "Label Spreading 50% data")
ls100 = (LabelSpreading().fit(X, y), y, "Label Spreading 100% data")

# Define the base classifier for self-training is identical to the SVC

# create a mesh to plot in
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

color_map = {-1: (1, 1, 1), 0: (0, 0, 0.9), 1: (1, 0, 0), 2: (0.8, 0.6, 0)}

classifiers = (ls30, st30, ls50, st50, ls100, rbf_svc)
for i, (clf, y_train, title) in enumerate(classifiers):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    plt.subplot(3, 2, i + 1)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    plt.axis("off")

    # Plot also the training points
    colors = [color_map[y] for y in y_train]
    plt.scatter(X[:, 0], X[:, 1], c=colors, edgecolors="black")

    plt.title(title)

plt.suptitle("Unlabeled points are colored white", y=0.1)
plt.show()
```

Try: Make use of at least three other datasets and do the comparative study on the results obtained.

10.5 Label Propagation Learning a Complex Structure

The decision boundaries generated by SVM, Label Spreading, and Self-training on a dataset of iris were compared. The results show that these methods can learn good boundaries even if there are only a few labeled pieces of information. Note that training with 100% of the data is not included since it is functionally identical to doing so with the SVC. The goal of this exercise is to demonstrate the label spreading and self-training can learn good boundaries.

Input: make_circles dataset

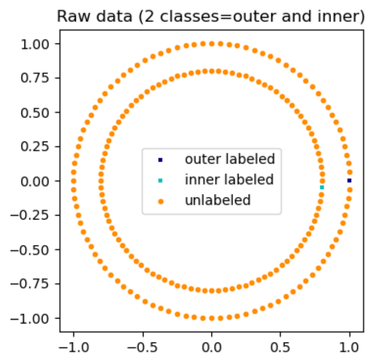
Output: Obtain 100% with self-trained data

Explanation:
Example of LabelPropagation learning a complex internal structure to demonstrate "manifold learning". The outer circle should be labeled "red" and the inner circle "blue". Because both label groups lie inside their own distinct shape, we can see that the labels propagate correctly around the circle.

We generate a dataset with two concentric circles. In addition, a label is associated with each sample of the dataset that is: 0 (belonging to the outer circle), 1 (belonging to the inner circle), and -1 (unknown). Here, all labels but two are tagged as unknown.

Hint:
```
import numpy as np

from sklearn.datasets import make_circles

n_samples = 200
X, y = make_circles(n_samples=n_samples, shuffle=False)
outer, inner = 0, 1
labels = np.full(n_samples, -1.0)
labels[0] = outer
labels[-1] = inner
```

Hint:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 4))
plt.scatter(
    X[labels == outer, 0],
    X[labels == outer, 1],
    color="navy",
    marker="s",
    lw=0,
    label="outer labeled",
    s=10,
)
plt.scatter(
    X[labels == inner, 0],
    X[labels == inner, 1],
    color="c",
    marker="s",
    lw=0,
    label="inner labeled",
    s=10,
)
plt.scatter(
    X[labels == -1, 0],
    X[labels == -1, 1],
    color="darkorange",
    marker=".",
    label="unlabeled",
)
plt.legend(scatterpoints=1, shadow=False, loc="center")
_ = plt.title("Raw data (2 classes=outer and inner)")
```



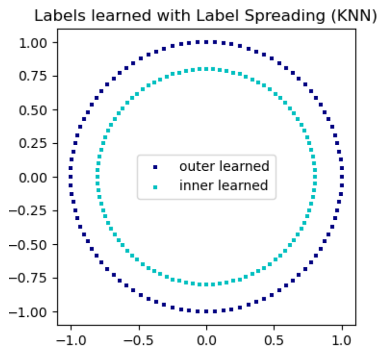
The aim of **LabelSpreading** is to associate a label to sample where the label is initially unknown.

```
from sklearn.semi_supervised import LabelSpreading

label_spread = LabelSpreading(kernel="knn", alpha=0.8)
label_spread.fit(X, labels)
LabelSpreading(alpha=0.8, kernel='knn')
```

Now, we can check which labels have been associated with each sample when the label was unknown.

Hint:

```
output_labels = label_spread.transduction_
output_label_array = np.asarray(output_labels)
outer_numbers = np.where(output_label_array == outer)[0]
inner_numbers = np.where(output_label_array == inner)[0]

plt.figure(figsize=(4, 4))
plt.scatter(
    X[outer_numbers, 0],
    X[outer_numbers, 1],
    color="navy",
    marker="s",
    lw=0,
    s=10,
    label="outer learned",
)
plt.scatter(
    X[inner_numbers, 0],
    X[inner_numbers, 1],
    color="c",
    marker="s",
    lw=0,
    s=10,
    label="inner learned",
)
plt.legend(scatterpoints=1, shadow=False, loc="center")
plt.title("Labels learned with Label Spreading (KNN)")
plt.show()
```



Try: Implement the code to perform the efficient non-parametric function induction in semi-supervised learning.

11. Reinforcement Learning Techniques and Its Application

11.1 Working of RL Algorithm

In some situations, there is a lot of data available out there. However, algorithms aren't available to teach machines the logic to arrive at the desired output. This is where machine learning comes to the rescue. Machine learning is the technology that, given the inputs and the desired outputs, will arrive at the logic or the algorithm to predict the output for an unforeseen or new input. The goal of this exercise is to develop the code to implement the RL algorithm.

Input: The state of the agent, environment, and the actions to be performed.

Output: Rewards accumulated by the agent in each step up to 20 steps which is the upper limit defined.

Explanation:

In a nutshell, RL is the branch of machine learning in which a machine learns from experience and takes proper decisions to maximize its reward or, in other words, to get the best reward possible. The machine is called the agent here. For every action it takes, it receives an award if it was the right action, failing which, it receives a punishment if it was the wrong action.

The best and the most common example of RL is how pet dogs are trained to get the stick and come back to their master! Every time the dog fails to get the stick or gets the wrong stick, it will not get its treat otherwise it will be rewarded with its delicious treats. The dog, quite obviously, aims to maximize the number of treats it gets because it loves enjoying its food! In this case, the dog is referred to as the agent.

Agent:

To check if the action taken by the agent was correct or wrong, logic will be involved. But, here, let's choose one of the rewards randomly using the random package. Let's begin by importing it:

```
import random
```

With the above understanding, let us define the environment class as follows:

Hint:

```
#create Environment class

class MyEnvironment:
def __init__(self):
self.remaining_steps=20

def get_observation(self):
return [1.0,2.0,1.0]

def get_actions(self):
return [-1,1]

def check_is_done(self):
return self.remaining_steps==0

def action(self,int):
if self.check_is_done():
```

```
raise Exception("Game over")
self.remaining_steps-=1

return random.random()
```

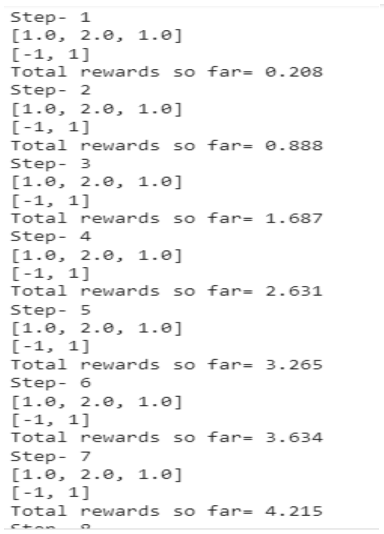
myAgent:
With this knowledge, the agent class can be defined as follows:

Hint:
```
class myAgent:
   def __init__(self):
     self.total_rewards=0.0
   def step(self,ob:MyEnvironment):
     curr_obs=ob.get_observation()
     print(curr_obs)
     curr_action=ob.get_actions()
     print(curr_action)
     curr_reward=ob.action(random.choice(curr_action))
     self.total_rewards+=curr_reward
     print("Total rewards so far= %.3f "%self.total_rewards)
```

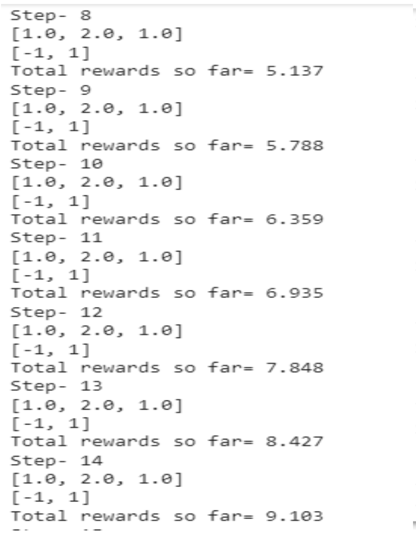
Finally, create objects of the above classes and execute as follows:

```
if __name__=='__main__':
        obj=MyEnvironment()
        agent=myAgent()
        step_number=0

while not obj.check_is_done():
        step_number+=1
        print("Step-",step_number)
        agent.step(obj)
print("Total reward is %.3f "%agent.total_rewards)
```

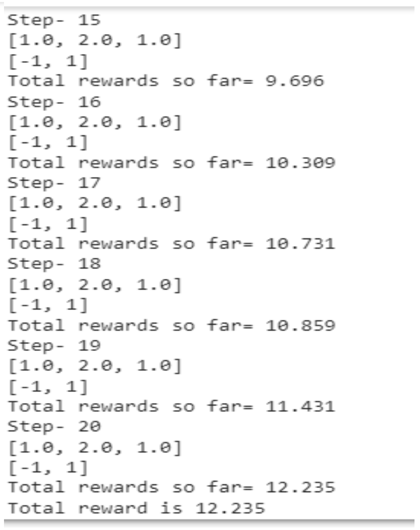
Results
Running the above code, we will get the rewards accumulated by the agent in each step up to 20 steps which is the upper limit defined by us. Here is a snapshot of what I got:



```
Step- 1
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 0.208
Step- 2
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 0.888
Step- 3
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 1.687
Step- 4
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 2.631
Step- 5
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 3.265
Step- 6
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 3.634
Step- 7
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 4.215
```



```
Step- 8
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 5.137
Step- 9
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 5.788
Step- 10
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 6.359
Step- 11
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 6.935
Step- 12
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 7.848
Step- 13
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 8.427
Step- 14
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 9.103
```



```
Step- 15
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 9.696
Step- 16
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 10.309
Step- 17
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 10.731
Step- 18
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 10.859
Step- 19
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 11.431
Step- 20
[1.0, 2.0, 1.0]
[-1, 1]
Total rewards so far= 12.235
Total reward is 12.235
```

Try: Implement the reinforcement learning technique with an OpenAI's gym, specially with MountainCar-v0 environment.

11.2 Q-Learning Technique

Most of the machine learning algorithms are trained based on the training dataset and show their efficiency by understanding the unseen data. These algorithms are touted as the future of Machine Learning as these eliminate the cost of collecting and cleaning the data. Reinforcement Learning is a type of Machine Learning paradigm in which a learning algorithm is trained not on preset data but rather based on a feedback system. The goal of this exercise is to implement a basic Reinforcement Learning algorithm which is called the Q-Learning technique. In this exercise, we attempt to teach a bot to reach its destination using the Q-Learning technique.

Input: A state or an input state

Output: The most efficient path to reach its destination by a bot

Explanation:

Training an RL model is an iterative process because the agent keeps on learning from its experience. It keeps exploring the environment. Here, the agent faces a trade-off between experience and exploration: At a given time, should the agent explore the environment and decide its next action, or should it decide its next action based on its previous experience?

While training an RL model, firstly, scores are assigned to all the grids in the environment. The agent explores all the possible paths and learns from experience, again, aiming to maximize this total score it achieves by choosing among the grids.
The agent keeps exploring until it gets a negative reward. It stops at this point, realizing,"Oh! I was not supposed to go this way. I was wrong."

Step 1: Importing the required libraries.

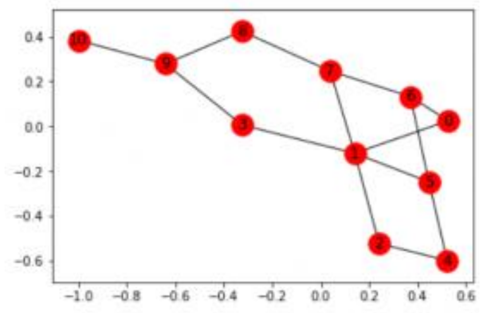
Hint:

```
import numpy as np
import pylab as pl
import networkx as nx
```

Step 2: Defining and visualizing the graph.

Hint:

```
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
         (8, 9), (7, 8), (1, 7), (3, 9)]
goal = 10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
pl.show()
```

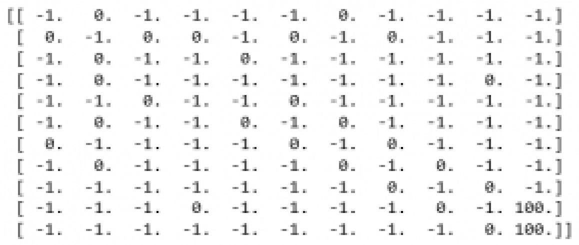


Note: The above graph may not look the same on reproduction of the code because the networkx library in python produces a random graph from the given edges.

Step 3: Defining the reward the system for the bot

Hint:

```
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1
for point in edges:
    print(point)
    if point[1] == goal:
        M[point] = 100
    else:
        M[point] = 0

    if point[0] == goal:
        M[point[::-1]] = 100
    else:
        M[point[::-1]]= 0
        # reverse of point
M[goal, goal]= 100
print(M)
# add goal point round trip
```



```
[[ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.   0.   0.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.]
 [ -1.  -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.]
 [ -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.   0.  -1. 100.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0. 100.]]
```

Step 4: Defining some utility functions to be used in the training.

Hint:

```
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
gamma = 0.75
# learning parameter
initial_state = 1
# Determines the available actions for a given state
def available_actions(state):
    current_state_row = M[state, ]
    available_action = np.where(current_state_row >= 0)[1]
    return available_action
 available_action = available_actions(initial_state)
# Chooses one of the available actions at random

# Updates the Q-Matrix according to the path chosen
 update(initial_state, action, gamma)
```

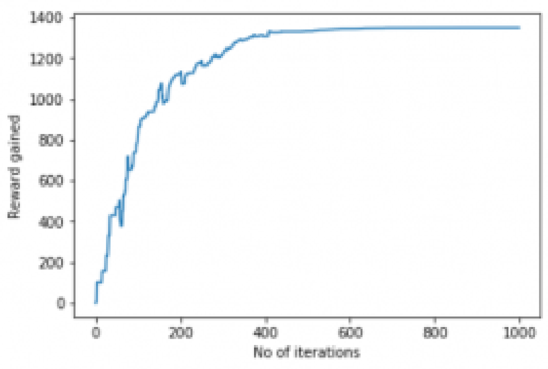
Step 5: Training and evaluating the bot using the Q-Matrix

Hint:

```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)
# print("Trained Q matrix:")
# print(Q / np.max(Q)*100)
# You can uncomment the above two lines to view the trained Q matrix
# Testing
```

```
current_state = 0
steps = [current_state]
while current_state != 10:
     next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state,
]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index
print("Most efficient path:")
print(steps)
pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()
```

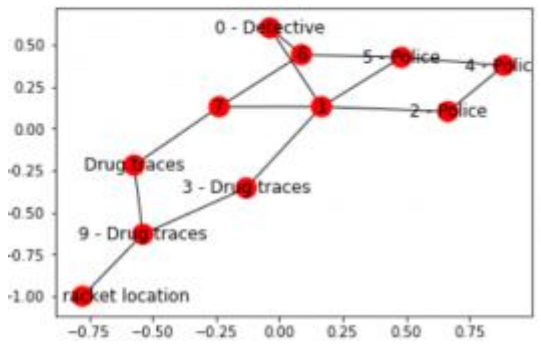
```
Most efficient path:
[0, 1, 3, 9, 10]
```



Step 6: Defining and visualizing the new graph with the environmental clues.

Hint:

```
# Defining the locations of the police and the drug traces
```



Note: The above graph may look a bit different from the previous graph but they, in fact, are the same graphs. This is due to the random placement of nodes by the networkx library.

Step 7: Defining some utility functions for the training process

Hint:

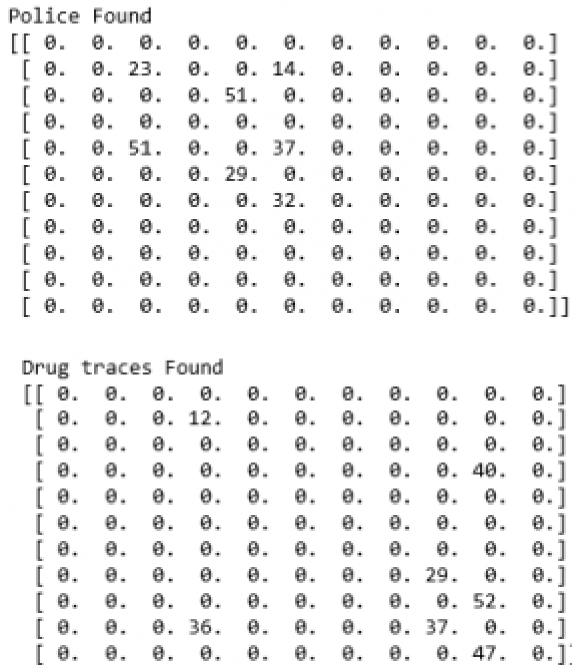
```
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_police = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_drugs = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
initial_state = 1
# Same as above
def available_actions(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]
    return av_action
# Same as above
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action
# Exploring the environment
def collect_environmental_data(action):
    found = []
    if action in police:
        found.append('p')
    if action in drug_traces:
        found.append('d')
    return (found)
available_action = available_actions(initial_state)
action = sample_next_action(available_action)
def update(current_state, action, gamma):
  max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  environment = collect_environmental_data(action)
  if 'p' in environment:
    env_police[current_state, action] += 1
  if 'd' in environment:
    env_drugs[current_state, action] += 1
  if (np.max(Q) > 0):
    return(np.sum(Q / np.max(Q)*100))
  else:
    return (0)
# Same as above
update(initial_state, action, gamma)
def available_actions_with_env_help(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]
# if there are multiple routes, dis-favor anything negative
```

```
    env_pos_row = env_matrix_snap[state, av_action]
    if (np.sum(env_pos_row < 0)):
# can we remove the negative directions from av_act?
        temp_av_action = av_action[np.array(env_pos_row)[0]>= 0]
        if len(temp_av_action) > 0:
            av_action = temp_av_action
    return av_action
# Determines the available actions according to the environment
```

Step 8: Visualizing the Environmental matrices

Hint:

```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
# Print environmental matrices
```

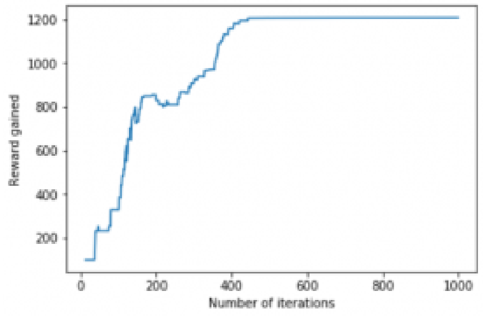


```
Police Found
[[ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.  23.   0.   0.  14.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.  51.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.  51.   0.   0.  37.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.  29.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.  32.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]

Drug traces Found
[[ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.  12.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.  40.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.  29.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.  52.   0.]
 [ 0.   0.   0.  36.   0.   0.   0.   0.  37.   0.   0.]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.  47.   0.]]
```

Step 9: Training and evaluating the model

Hint:

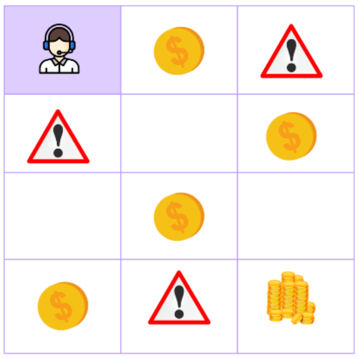
```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions_with_env_help(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)
pl.plot(scores)
pl.xlabel('Number of iterations')
pl.ylabel('Reward gained')
pl.show()
```



Try: Implement the Q Algorithm and Agent (Q Learning) and build Q Table by including all the possible discrete states.

11.3 Deep Q-Networks

Consider, that there is an AI agent present within a maze environment, and its goal is to find a reward. The agent interacts with the environment by performing some actions, and based on those actions, the state of the agent gets changed, and it also receives a reward or penalty as feedback.



Input: 1. Install the random package and choose the rewards

2. Environment and Action classes

Output: Perform exactly 10 steps and make the agent to again as many rewards as possible.

Explanation:

The field of reinforcement learning is made up of several algorithms that each take different approaches. The differences are mainly due to their strategies for exploring their environments. Some of the important Reinforcement learning algorithms are listed as follows.

- **Q-learning**
- **Deep Q-Networks**

```
#Importing the random package
import random
```

We use two classes, Environment and Agent in our model.

The environment class represents the agent's environment. The class must have member functions to get the current observation or state where the agent is, what are the points for reward and punishment, and keep track of how many more steps are left that the agent can take before the game is over. In this example, consider a game that the agent must finish in at most ten steps.

Hint:

```
#Creating the Environment class

class Environment:
  def init(self):
    self.steps_left=10
  def get_observation(self):
    return [1.0,2.0,1.0]
  def get_actions(self):
    return [-1,1]
  def check_is_done(self):
    return self.steps_left==0
  def action(self,int):
    if self.check_is_done():
      raise Exception("Game over")
    self.steps_left-=1
    return random.random()
```

The agent class is simpler compared to the environment class. The agent collects rewards given to it by its environment and makes an action. For this, we will need a data member and a member function.

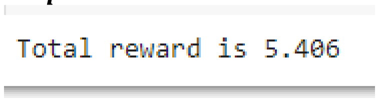
Hint:

```
#Creating the Agent class
class Agent:
  def init(self):
  self.total_rewards=0.0
  def step(self,ob:Environment):
    curr_obs=ob.get_observation()
    #print(curr_obs,end=" ")
    curr_action=ob.get_actions()
    #print(curr_action)
    curr_reward=ob.action(random.choice(curr_action))
    self.total_rewards+=curr_reward
    #print("Total rewards so far= %.3f "%self.total_rewards)
```

Until the game is not over, which is checked by the while loop, the agent takes an action by invoking the step function of the Agent class by passing obj which refers to the agent's environment. The reward here can be positive(in case of 1) or negative(in case of -1) and will be added to the total rewards of the agent.

```
if name=='main':
  obj=Environment()
  agent=Agent()
  step_number=0
  while not obj.check_is_done():
    step_number+=1

#print("Step-",step_number, end=" ")
    agent.step(obj)
  print("Total reward is %.3f "%agent.total_rewards)
```

Output:



```
Total reward is 5.406
```

On executing the code, we will get the rewards accumulated by the agent in each and every step up to 10 steps. The output differs with each time we play the game.

Try: Perform the sequence of actions that will eventually generate the maximum total reward using Markov Decision Process.

11.4 Q-Learning with Discount Factor

The concept of reinforcement learning is a type of machine learning that allows agents to learn to behave in an environment by giving them rewards. The agent can interact with the environment and act based on its current condition. The goal of this exercise is to implement reinforcement learning is to help agents develop a policy that will allow them to maximize their reward over time. This type of machine learning is commonly used in various fields such as gaming and robotics. It can help improve an agent's decision-making skills.

Input: Environment with OpenAI Gym Library

Output: Q-Table Matrix with maximum number of rewards

Explanation:

Reinforcement Learning is a subfield of machine learning where an agent learns to act in an environment by receiving feedback in the form of rewards. The agent interacts with the environment, takes actions based on its current state, and receives a reward for the action it takes. The agent’s objective is to learn a policy that maximizes the cumulative reward over time. Reinforcement Learning is used in various domains such as robotics, gaming, and recommendation systems. It is advantageous because it enables agents to improve their decision-making abilities through experience.

Here’s code example of how RL works, implemented in Python using the OpenAI Gym library:

1 Import the necessary libraries:

Hint:

```
# pip install gym
import gym
import numpy as np
```

2 Create an environment:

```
# Creating the env
env = gym.make('CartPole-v1')
```

3 Define the parameters:

```
# Extracting the number of dimensions
```

4 Initialize the Q-table:

```
# Initialize the Q-table

Q = np.zeros((n_states, n_actions))
```

5 Set the hyperparameters:

Hint:

```
# Learning rate
alpha = 0.1

# Discount factor
gamma = 0.99

# Exploration rate
epsilon = 0.1
```

6 Train the agent using Q-learning:

Hint:

```
# Setting the number of episodes
n_episodes = 10000

# Train the agent using Q-learning
```

7 Test the agent:

```
# Test the agent
n_episodes = 100
total_reward = 0

for episode in range(n_episodes):
    state = env.reset()
    done = False

    while not done:
        #chooses the action with the highest Q-value
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        #total_reward
        total_reward += reward
```

```
print("Average reward over {} episodes: {}".format(n_episodes,
total_reward/n_episodes))
```

After training, the agent is tested on 100 episodes and the average reward is computed. The output should be a number between 0 and 500, with higher values indicating better performance.

Output:

```
# Output

Average reward over 100 episodes: 487.28
```

Try: Make an agent that can play a game called CartPole. We can also use an Atari game but training an agent to play that takes a while (from a few hours to a day). The idea behind our approach will remain the same so you can try this on an Atari game on your machine.

11.5 Anomaly Detection

The goal of this exercise is to use DBSCAN to analyze a classification problem involving credit card transactions and customers' history.

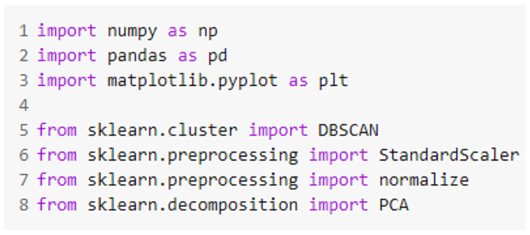
Input: Credit Card Dataset

Output: Implement the DBSCAN algorithm to detect the outliers or anomaly in the data.

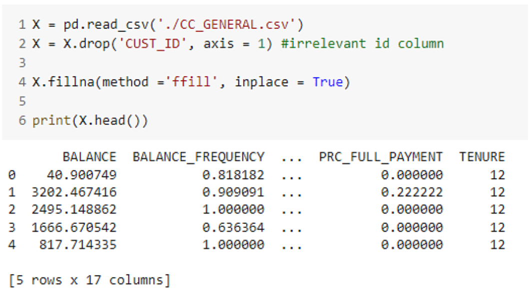
Explanation:

This example will use scikit-learn to implement one of the many algorithms we discovered today in Python. Let's look at a classification problem of segmenting customers based on their credit card activity and history and using DBSCAN to identify outliers or anomalies in the data.

First, fetch the data from Kaggle at Credit Card Dataset for Clustering. Next, we import the necessary libraries and explore the data.

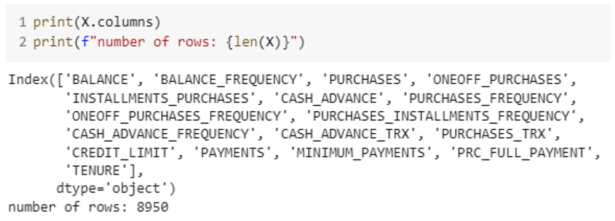


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.decomposition import PCA
```



```
X = pd.read_csv('./CC_GENERAL.csv')
X = X.drop('CUST_ID', axis = 1) #irrelevant id column

X.fillna(method ='ffill', inplace = True)

print(X.head())
```

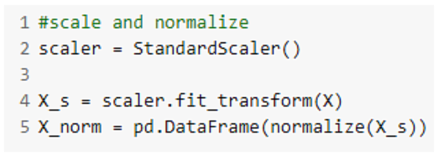
```
       BALANCE  BALANCE_FREQUENCY  ...  PRC_FULL_PAYMENT  TENURE
0    40.900749           0.818182  ...          0.000000      12
1  3202.467416           0.909091  ...          0.222222      12
2  2495.148862           1.000000  ...          0.000000      12
3  1666.670542           0.636364  ...          0.000000      12
4   817.714335           1.000000  ...          0.000000      12

[5 rows x 17 columns]
```



```
print(X.columns)
print(f"number of rows: {len(X)}")
```

```
Index(['BALANCE', 'BALANCE_FREQUENCY', 'PURCHASES', 'ONEOFF_PURCHASES',
       'INSTALLMENTS_PURCHASES', 'CASH_ADVANCE', 'PURCHASES_FREQUENCY',
       'ONEOFF_PURCHASES_FREQUENCY', 'PURCHASES_INSTALLMENTS_FREQUENCY',
       'CASH_ADVANCE_FREQUENCY', 'CASH_ADVANCE_TRX', 'PURCHASES_TRX',
       'CREDIT_LIMIT', 'PAYMENTS', 'MINIMUM_PAYMENTS', 'PRC_FULL_PAYMENT',
       'TENURE'],
      dtype='object')
number of rows: 8950
```

Normalize and scale to preprocess the data as unsupervised algorithms are greatly sensitive to distance measures.



```
#scale and normalize
scaler = StandardScaler()

X_s = scaler.fit_transform(X)
X_norm = pd.DataFrame(normalize(X_s))
```

Before moving on to fit the DBSCAN model, for the sake of visualization, efficiency, and simplicity, we perform dimensionality reduction to reduce the 17 columns to 2.

Hint:

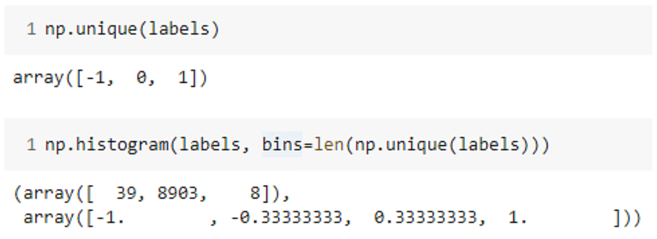
```
pca = PCA(n_components = 2)
X_principal = pca.fit_transform(X_norm)
X_principal = pd.DataFrame(X_principal)
X_principal.columns = [‘P1’, ‘P2’]
```

Let’s fit the DBSCAN model now using eps 0.05 and minPts as 10.

```
db_model = DBSCAN(eps = 0.05, min_samples = 10).fit(X_reduce)
labels = db_model.labels_
```

“labels” is a vector of the same length as the number of training samples. It contains the class index for each sample, indicating the class it was assigned to. Anomalies have ‘-1’ as their class index. Below we can see how the two clusters and anomalies are distributed in the 8950 samples.

```
np.unique(labels)
np.histogram(labels, bins = len(np.unique(labels)))
```



```
np.unique(labels)
```

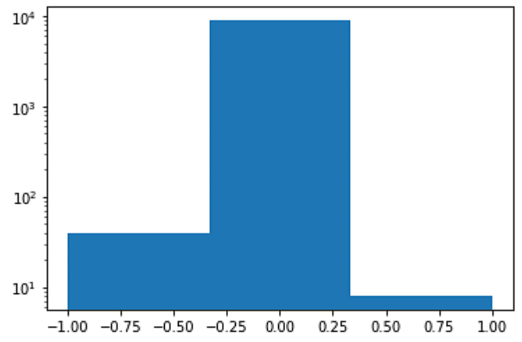
```
array([-1,  0,  1])
```

```
np.histogram(labels, bins=len(np.unique(labels)))
```

```
(array([  39, 8903,    8]),
 array([-1.        , -0.33333333,  0.33333333,  1.        ]))
```

We can also visualize a similar logarithmic histogram for visual intuition:

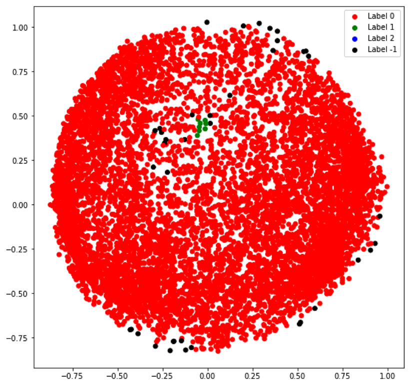
```
plt.hist(labels, bins=len(np.unique(labels)), log=True)
plt.show()
```



```
n_clusters = len(np.unique(labels))-1
anomaly = list(labels).count(-1)
print(f'Clusters: {n_clusters}')
print(f'Abnormal points: {anomaly}')
```

```
Clusters: 2
Abnormal points: 39
```

Finally, since we chose two feature columns purposefully to visualize the anomalies and clusters together, let's plot a scatter plot of the results.



As expected, anomalies lie in the regions with less density – typically around the edges and then towards the center, where the points are relatively scant. These abnormal samples can be highlighted for manual review by bank officials.

```
X_anomaly = X.iloc[np.argwhere(labels==-1).reshape((-1,))]
print(X_anomaly.head())
```

```
         BALANCE  BALANCE_FREQUENCY  ...  PRC_FULL_PAYMENT  TENURE
86   7069.950386                1.0  ...          0.000000      12
87   8181.251131                1.0  ...          0.000000      12
109  6644.201651                1.0  ...          0.083333      12
120  8504.876253                1.0  ...          0.000000      12
468  6426.639738                1.0  ...          0.000000      12

[5 rows x 17 columns]
```

Thus, we have implemented an unsupervised anomaly detection algorithm called DBSCAN using scikit-learn in Python to detect possible credit card fraud. Before concluding, let's look at some other popular projects in anomaly detection that you can implement for practice.

Try: Implement DBSCAN Clustering using Mall Customer Segmentation Data from Kaggle and visualize the distribution of clusters.

11.6 Game Playing

Consider you are teaching the dog to catch a ball, but you cannot teach the dog explicitly to catch a ball, instead, you will just throw a ball, every time the dog catches a ball, you will give a cookie. If it fails to catch a dog, you will not give a cookie. So, the dog will figure out what actions it does that made it receive a cookie and repeat that action. Similarly in an RL environment, you will not teach the agent what to do or how to do it, instead, you will give feedback to the agent for each action it does. The feedback may be positive (reward) or negative (punishment). The learning system which receives the punishment will improve itself. Thus, it is a trial-and-error process. The reinforcement learning algorithm retains outputs that maximize the received reward over time. In the above analogy, the dog represents the agent, giving a cookie to the dog on catching a ball is a reward and not giving a cookie is punishment. The goal of this exercise is to implement the reinforcement learning algorithm to retain the maximum number of awards.

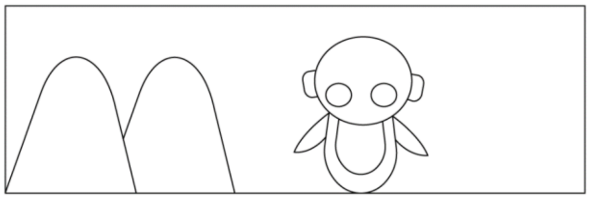
Input: Environment and an Agent

Output: The maximum received reward points overtime.

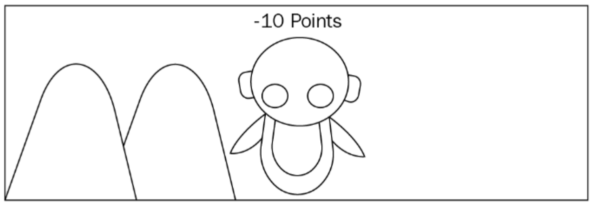
Explanation:

An RL agent can explore for different actions which might give a good reward, or it can (exploit) use the previous action which resulted in a good reward. If the RL agent explores different actions, there is a great possibility to get a poor reward. If the RL agent exploits past action, there is also a great possibility of missing out on the best action which might give a good reward. There is always a trade-off between exploration and exploitation. We cannot perform both exploration and exploitation at the same time. We will discuss exploration exploitation dilemma detail in the upcoming chapters.

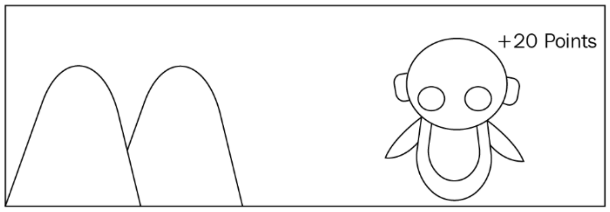
Say, If you want to teach a robot to walk, without getting stuck by hitting at the mountain, you will not explicitly teach the robot not to go in the direction of mountain,



Instead, if the robot hits and get stuck on the mountain you will reduce 10 points so that robot will understand that hitting mountain will give it a negative reward so it will not go in that direction again.



And you will give 20 points to the robot when it walks in the right direction without getting stuck. So robots will understand which is the right path to rewards and try to maximize the rewards by going in a right direction.



Hint:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## What is Reinforcement Learning?"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "Consider you are teaching the dog to catch a ball, but you cannot teach the dog explicitly to\n",
    "catch a ball, instead, you will just throw a ball, every time the dog catches a ball, you will\n",
    "give a cookie. If it fails to catch a dog, you will not give a cookie. So the dog will figure out\n",
    "what actions it does that made it receive a cookie and repeat that action."
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "Similarly in an RL environment, you will not teach the agent what to do or how
to do,\n",
    "instead, you will give feedback to the agent for each action it does. The
feedback may be\n",
    "positive (reward) or negative (punishment). The learning system which receives
the\n",
    "punishment will improve itself. Thus it is a trial and error process. The
reinforcement\n",
    "learning algorithm retains outputs that maximize the received reward over time.
In the\n",
    "above analogy, the dog represents the agent, giving a cookie to the dog on
catching a ball is\n",
    "a reward and not giving a cookie is punishment.\n",
    "\n",
    "There might be delayed rewards. You may not get a reward at each step. A reward
may be\n",
    "given only after the completion of the whole task. In some cases, you get a
reward at each\n",
    "step to find out that whether you are making any mistake.\n",
    "\n",
    "An RL agent can explore for different actions which might give a good reward or
it can\n",
    "(exploit) use the previous action which resulted in a good reward. If the RL
agent explores\n",
    "different actions, there is a great possibility to get a poor reward. If the RL
agent exploits\n",
    "past action, there is also a great possibility of missing out the best action
which might give a\n",
    "good reward. There is always a trade-off between exploration and exploitation.
We cannot\n",
    "perform both exploration and exploitation at the same time. We will discuss
exploration exploitation\n",
    "dilemma detail in the upcoming chapters.\n",
    "\n",
    "Say, If you want to teach a robot to walk, without getting stuck by hitting at
the mountain,\n",
    "you will not explicitly teach the robot not to go in the direction of mountain,"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "![title](images/B09792_01_01.png)"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "Instead, if the robot hits and get stuck on the mountain you will reduce 10
points so that\n",
    "robot will understand that hitting mountain will give it a negative reward so
it will not go\n",
    "in that direction again."
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "![title](images/B09792_01_02.png)"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "And you will give 20 points to the robot when it walks in the right direction
without getting\n",
    "stuck. So robot will understand which is the right path to rewards and try to
maximize the\n",
    "rewards by going in a right direction."
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {
    "collapsed": true
   },
   "source": [
    "![title](images/B09792_01_03.png)"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python [conda env:anaconda]",
   "language": "python",
   "name": "conda-env-anaconda-py"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 2
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython2",
   "version": "2.7.11"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```

Try: Implement the code to demonstrate how Deep Q-Learning algorithm learns to play game. Optimally, show the code how to optimize the artificial neural network using Bayesian optimization.

12. Evolutionary Learning Techniques

12.1 Evolution Strategies – Stochastic Global Optimization Algorithm

Methods for stochastic optimization provide a means of coping with inherent system noise and coping with models or systems that are highly nonlinear, high dimensional, or otherwise inappropriate for classical deterministic methods of optimization. The goal of this exercise is to use the randomness in an optimization algorithm that allows the search procedure to perform well on challenging optimization problems that may have a nonlinear response surface.

Input: Algorithm

Output: Visualization representing the outcome of ackley multimodal function

Explanation:

Here, we will develop a *(mu, lambda)-ES*, that is, a version of the algorithm where children replace parents. First, let's define a challenging optimization problem as the basis for implementing the algorithm. The Ackley function is an example of a multimodal objective function that has a single global optima and multiple local optima in which a local search might get stuck.

As such, a global optimization technique is required. It is a two-dimensional objective function that has a global optimum at [0,0], which evaluates to 0.0. The example below implements Ackley and creates a three-dimensional surface plot showing the global optima and multiple local optima.

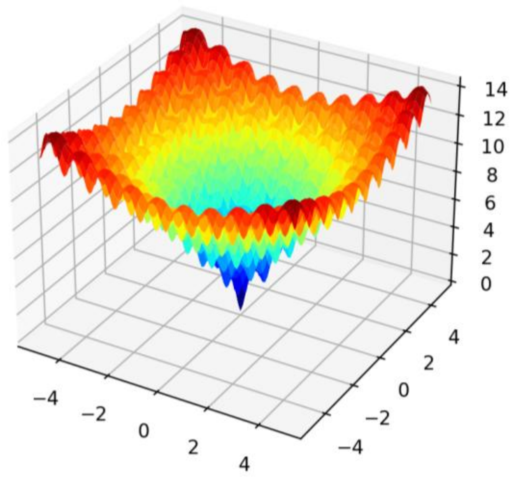
Hint:

```
# ackley multimodal function
from numpy import arange
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import meshgrid
from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D
```

```

# objective function

def objective(x, y):

      return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
* x) + cos(2 * pi * y))) + e + 20

# define range for input

r_min, r_max = -5.0, 5.0

# sample input range uniformly at 0.1 increments and create a mesh from the axis

# compute targets

```

```
results = objective(x, y)

# create a surface plot with the jet color scheme and show the same

```

Running the example creates the surface plot of the Ackley function showing the vast number of local optima.



```
# check if a point is within the bounds of the search

def in_bounds(point, bounds):

       # enumerate all dimensions of the point

       for d in range(len(bounds)):

              # check if out of bounds for this dimension

              if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
                     return False
       return True
```

We can then use this function when generating the initial population of "*lam*" (e.g. *lambda*) random candidate solutions. For example:

```

# initial population

population = list()
for _ in range(lam):
       candidate = None
       while candidate is None or not in_bounds(candidate, bounds):
              candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:,
0])
       population.append(candidate)
```

```
# Evaluate fitness for the population and select the indexes for the top ranked solutions
```

We can then create children for each selected parent. First, we must calculate the total number of children created per parent.

```
# calculate the number of children per parent

n_children = int(lam / mu)
```

We can then iterate over each parent and create modified versions of each.
We will create children using a similar technique used in stochastic hill climbing. Specifically, each variable will be sampled using a Gaussian distribution with the current value as the mean and the standard deviation provided as a "*step size*" hyperparameter.

```
# create children for parent

for _ in range(n_children):
      child = None
      while child is None or not in_bounds(child, bounds):
            child = population[i] + randn(len(bounds)) * step_size
```

We can also check if each selected parent is better than the best solution seen so far so that we can return the best solution at the end of the search.
...

```
# check if this parent is the best solution ever seen

if scores[i] < best_eval:
      best, best_eval = population[i], scores[i]
      print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
```

The created children can be added to a list and we can replace the population with the list of children at the end of the algorithm iteration.

```
# replace population with children

population = children
```

We can tie all this together into a function named *es_comma()* that performs the comma version of the Evolution Strategy algorithm.

Hint:

```
# evolution strategy (mu, lambda) algorithm
```

```
def es_comma(objective, bounds, n_iter, step_size, mu, lam):
	best, best_eval = None, 1e+10

	# calculate the number of children per parent

	n_children = int(lam / mu)

	# initial population

	population = list()

	for _ in range(lam):
		candidate = None
		while candidate is None or not in_bounds(candidate, bounds):
			candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
bounds[:, 0])
		population.append(candidate)

	# Perform the search

	for epoch in range(n_iter):

		# Evaluate fitness for the population

		scores = [objective(c) for c in population]

		# rank scores in ascending order

		ranks = argsort(argsort(scores))

		# select the indexes for the top mu ranked solutions

		selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]

		# create children from parents

		children = list()
		for i in selected:

			# check if this parent is the best solution ever seen

			if scores[i] < best_eval:
				best, best_eval = population[i], scores[i]
				print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))

			# create children for parent

			for _ in range(n_children):
				child = None
				while child is None or not in_bounds(child, bounds):
```

```
                                 child    =   population[i]   +   randn(len(bounds))   *
step_size
                          children.append(child)

             # Replace population with children

             population = children
       return [best, best_eval]
```

At the end of the search, we will report the best candidate solution found during the search.

```

...

# Seed the pseudorandom number generator

# Define range for input

bounds = asarray([[-5.0, 5.0], [-5.0, 5.0]])

# Define the total iterations

n_iter = 5000

# Define the maximum step size

step_size = 0.15

# Number of parents selected

mu = 20

# The number of children generated by parents

lam = 100

# Write the code to perform the evolution strategy (mu, lambda) search

best, score = es_comma(objective, bounds, n_iter, step_size, mu, lam)
print('Done!')
print('f(%s) = %f' % (best, score))
```

Tying this together, the complete example of applying the comma version of the Evolution Strategies algorithm to the Ackley objective function is listed below.

Hint:

```
# evolution strategy (mu, lambda) of the ackley objective function

from numpy import asarray
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import e
from numpy import pi
from numpy import argsort
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

# objective function

def objective(v):
	x, y = v
	return -20.0 * exp(-0.2 * sqrt(0.5 * (x**2 + y**2))) - exp(0.5 * (cos(2 * pi
* x) + cos(2 * pi * y))) + e + 20
```

Hint:

```
# check if a point is within the bounds of the search

def in_bounds(point, bounds):

	# enumerate all dimensions of the point

	for d in range(len(bounds)):

		# check if out of bounds for this dimension

		if point[d] < bounds[d, 0] or point[d] > bounds[d, 1]:
			return False
	return True
```

Hint:

```
# evolution strategy (mu, lambda) algorithm

def es_comma(objective, bounds, n_iter, step_size, mu, lam):
	best, best_eval = None, 1e+10
	# calculate the number of children per parent
	n_children = int(lam / mu)
	# initial population
	population = list()
	for _ in range(lam):
		candidate = None
		while candidate is None or not in_bounds(candidate, bounds):
			candidate = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
bounds[:, 0])
		population.append(candidate)

	# perform the search

	for epoch in range(n_iter):
		# evaluate fitness for the population
		scores = [objective(c) for c in population]
```

```
            # rank scores in ascending order
            ranks = argsort(argsort(scores))
            # select the indexes for the top mu ranked solutions
            selected = [i for i,_ in enumerate(ranks) if ranks[i] < mu]
            # create children from parents
            children = list()
            for i in selected:
                  # check if this parent is the best solution ever seen
                  if scores[i] < best_eval:
                        best, best_eval = population[i], scores[i]
                        print('%d, Best: f(%s) = %.5f' % (epoch, best, best_eval))
                  # create children for parent
                  for _ in range(n_children):
                        child = None
                        while child is None or not in_bounds(child, bounds):
                              child  =  population[i]  +  randn(len(bounds))  *
step_size
                        children.append(child)
            # replace population with children
            population = children
      return [best, best_eval]
# Seed the pseudorandom number generator and define the range for input

# define the total iterations and define the maximum step size

# Number of parents selected
mu = 20

# The number of children generated by parents
lam = 100

# Perform the evolution strategy (mu, lambda) search
```

Running the example reports the candidate solution and scores each time a better solution is found, then reports the best solution found at the end of the search.

No doubt, this solution can be provided as a starting point to a local search algorithm to be further refined, a common practice when using a global optimization algorithm like ES.

```
0, Best: f([-0.82977995 2.20324493]) = 6.91249
0, Best: f([-1.03232526 0.38816734]) = 4.49240
1, Best: f([-1.02971385 0.21986453]) = 3.68954
2, Best: f([-0.98361735 0.19391181]) = 3.40796
2, Best: f([-0.98189724 0.17665892]) = 3.29747
2, Best: f([-0.07254927 0.67931431]) = 3.29641
3, Best: f([-0.78716147 0.02066442]) = 2.98279
3, Best: f([-1.01026218 -0.03265665]) = 2.69516
3, Best: f([-0.08851828 0.26066485]) = 2.00325
4, Best: f([-0.23270782 0.04191618]) = 1.66518
4, Best: f([-0.01436704 0.03653578]) = 0.15161
7, Best: f([0.01247004 0.01582657]) = 0.06777
9, Best: f([0.00368129 0.00889718]) = 0.02970
```

```
25, Best: f([ 0.00666975 -0.0045051 ]) = 0.02449
33, Best: f([-0.00072633 -0.00169092]) = 0.00530
211, Best: f([2.05200123e-05 1.51343187e-03]) = 0.00434
315, Best: f([ 0.00113528 -0.00096415]) = 0.00427
418, Best: f([ 0.00113735 -0.00030554]) = 0.00337
491, Best: f([ 0.00048582 -0.00059587]) = 0.00219
704, Best: f([-6.91643854e-04 -4.51583644e-05]) = 0.00197
1504, Best: f([ 2.83063223e-05 -4.60893027e-04]) = 0.00131
3725, Best: f([ 0.00032757 -0.00023643]) = 0.00115
Done!
f([ 0.00032757 -0.00023643]) = 0.001147
```

12.2 Genetic Algorithm for Function Optimization

Genetic Algorithms (GAs) are a type of Evolutionary Algorithm that simulate the process of natural selection to solve optimization problems. The goal is to find the maximum (or minimum) of a non-linear objective function using operations such as selection, crossover, and mutation. This method is particularly useful for complex functions with multiple local optima.

Input: A non-linear objective (fitness) function, for example: $f(x)=x \cdot \sin(10\pi x)+1.0$ where $0 \le x \le 1$

Output:

Optimal value of x after N generations.

Corresponding maximum value of the fitness function f(x).

Plot showing convergence of fitness values across generations.

Hint:

```
# import libraries
Import numpy as np
import matplotlib.pyplot as plt

# Define the fitness function
def fitness(x):
    return x * np.sin(10 * np.pi * x) + 1.0

# Initialize population
population_size = 50
generations = 100
mutation_rate = 0.1
population = np.random.rand(population_size)

# Evolution process
# Selection - select top 50% individuals
# Crossover - generate children
# Mutation
# New population
# Final output
# Plotting fitness over generations
```

12.3 Genetic Algorithm for Solving the Travelling Salesman Problem (TSP)

The Travelling Salesman Problem (TSP) is a classical combinatorial optimization problem where the goal is to find the shortest possible route that visits each city exactly once and returns to the starting point. Genetic Algorithms provide a heuristic approach to tackle TSP by evolving a population of city sequences (routes) using selection, crossover, and mutation techniques.

Input: A list of coordinates representing cities: cities = [(x1, y1), (x2, y2), ..., (xn, yn)]

Output: A near-optimal route covering all cities, Minimum tour distance, A plot of the best route, Convergence graph showing the distance improvement over generations.

Hint:

```
# import libraries
Import numpy as np
import matplotlib.pyplot as plt
import random

# Create random coordinates for cities
num_cities = 10
cities = np.random.rand(num_cities, 2)

# Calculate distance between two cities
def distance(c1, c2):
    return np.linalg.norm(c1 - c2)

# Total distance of the tour
def tour_length(tour):
    return sum(distance(cities[tour[i]], cities[tour[(i+1) % num_cities]]) for i in 
range(num_cities))

# Initialize population
# Create a population of random tours
# GA process
# Best tour
# Plotting the best route
# Plot convergence
```

12.4 Evolving Neural Network Weights using Genetic Algorithms

This experiment uses a Genetic Algorithm (GA) to optimize the weights of a simple neural network without using traditional backpropagation. The idea is to evolve the weights and biases for a fixed network architecture to minimize prediction error on a small dataset. This demonstrates how GAs can be used in neuroevolutionary, especially when gradient information is unavailable or not ideal.

Input:

A simple dataset like XOR: X = [[0,0], [0,1], [1,0], [1,1]] y = [0, 1, 1, 0]

Neural network architecture: 2 input neurons → 2 hidden neurons → 1 output neuron

Output:

Evolved weights and biases.

Accuracy or mean squared error (MSE) of the best individual.

A graph showing the error vs. generations.

Hint:

```
# import libraries
Import numpy as np
import matplotlib.pyplot as plt

# XOR dataset
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0, 1, 1, 0])

# Neural network architecture: 2-2-1 (input-hidden-output)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def forward_pass(individual, X):
       # Decode weights
       # Forward pass
# Fitness function: MSE
# Genetic algorithm parameters
# Initialize population
population    =    [np.random.uniform(-1,    1,    chromosome_length)    for    _    in
range(population_size)]

best_errors = []
       # Selection (top 50%)
       # Crossover
       # Mutation
       # Create new population
# Get best individual and evaluate
# Plot error over generations
```

Try: Change the network architecture (e.g., 3 hidden neurons), observe how it impacts learning. You can also compare the result with backpropagation for the same problem.

13. Natural Language Processing

13.1 Sentiment Analysis

We have a dataset with tweets. Some of them are annotated with positive, negative, or neutral sentiment. Unfortunately annotating is time and cost intensive — we need to pay annotators for doing so and cross-check their answers for correctness. Therefore, most of the tweets are not labelled as it is relatively cheap and easy to download them, but not so cheap to annotate them. The goal of this exercise is to extract some useful features from these images which could help us in other tasks.

Input: piece of text
Output: The sentiment (positive, negative, or neutral) based on its polarity score.

Explanation:

defines a function analyze_sentiment() that takes a piece of text as input and returns the sentiment (positive, negative, or neutral) based on its polarity score.

```
pip install textblob
```

Hint:

```
from textblob import TextBlob
def analyze_sentiment(text):
    # Create a TextBlob object
    blob = TextBlob(text)

    # Get the sentiment polarity
    polarity = blob.sentiment.polarity

    # Classify sentiment
    if polarity > 0:
        return "Positive"
    elif polarity < 0:
        return "Negative"
    else:
        return "Neutral"

# Complete the code
```

Try: Explore what sentiment analysis encompasses and the various ways to implement it in Python.

13.2 Text Classification Using Naive Bayes

Text classification is the process of assigning predefined categories to textual data. In this experiment, you'll build a classifier (e.g., Spam vs. Ham) using the Naive Bayes algorithm, which works well with text due to its probabilistic nature and speed.

Input: SMS Spam dataset (e.g., from UCI repository or Kaggle), Format: label, text

Output:

Classification of messages as "spam" or "ham".

Accuracy, precision, recall, and confusion matrix.

Hint:

```
# import libraries
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Load dataset
df = pd.read_csv("spam.csv", encoding='latin-1')[['v1', 'v2']]
df.columns = ['label', 'text']

# Preprocessing

# Split & Train

# Predict and evaluate
```

Try: Modify the existing experiment to use TF-IDF vectorization instead of CountVectorizer. Then, compare the performance (accuracy, precision, recall) of both models using a bar plot. Which vectorization method works better for spam detection in your dataset, and why?

13.3 Named Entity Recognition using spaCy

NER is a sub-task of information extraction that seeks to locate and classify named entities in text into predefined categories such as person names, organizations, locations, etc.

Input: Any sample news article or paragraph.

Output:

List of named entities with labels.

Visual representation using spaCy's displacy.

Hint:

```
# import libraries
import spacy
from spacy import displacy

# Load English model
nlp = spacy.load("en_core_web_sm")

# Input text

# Process and visualize
```

Try: Modify the input to a longer news article or Wikipedia excerpt containing multiple types of entities (e.g., people, dates, locations, organizations). Extract and count the number of entities for each type (e.g., how many PERSON, ORG, GPE, etc.). Then, visualize the counts using a bar chart.

13.4 Text Summarization using Transformers (BART or T5)

Text summarization is the process of creating a shorter version of a long document while preserving its key information. Transformer-based models like BART and T5 are pre-trained for this task and can generate high-quality summaries using deep learning.

Input: A long news article or paragraph

Output: A concise summary of the input text

Hint:

```
# import libraries
from transformers import pipeline

# Load summarization pipeline
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")

# Input Long text
text = """              """

# Generate summary
```

Try: Compare the results of BART and T5 for the same text. Adjust min_length and max_length to control the level of summarization.

14. Deep Learning

14.1 Deep Neural Network for Classification

Deep Learning has seen significant advancements with companies looking to build intelligent systems using vast amounts of unstructured data. Deep Learning works on the theory of artificial neural networks. Develop a sequential neural network and apply Adam Optimization algorithm. Choose a model configuration and training configuration that achieves the lowest loss and highest accuracy possible for a given dataset.

Input: Pima Indian Diabetes.csv dataset

Output: Classification with highest accuracy and lowest loss.

Explanation:

1. Load Data
The first step is to define the functions and classes you intend to use in this tutorial.
You will use the NumPy library to load your dataset and two classes from the Keras library to define your model. The imports required are listed below.

Hint:
```
# Write the code to design first neural network with keras
```

You can now load our dataset. The dataset is available here:

- Dataset CSV File (pima-indians-diabetes.csv)

Download the dataset and place it in your local working directory, the same location as your Python file. Save it with the filename:

```
pima-indians-diabetes.csv
```

Look inside the file; you should see rows of data like the following:

```
6,148,72,35,0,33.6,0.627,50,1
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
1,89,66,23,94,28.1,0.167,21,0
0,137,40,35,168,43.1,2.288,33,1
...
```

You can now load the file as a matrix of numbers using the NumPy function loadtxt().
There are eight input variables and one output variable (the last column). You will be learning a model to map rows of input variables (X) to an output variable (y), which is often summarized as *y* = *f(X)*.
The variables can be summarized as follows:
Input Variables (X):

1. Number of times pregnant
2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-hour serum insulin (mu U/ml)
6. Body mass index (weight in kg/(height in m)^2)
7. Diabetes pedigree function
8. Age (years)

Output Variables (y):

1. Class variable (0 or 1)

Hint:

```
...
# Load the dataset
dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')

# Split into input (X) and output (y) variables
X = dataset[:,0:8]
y = dataset[:,8]
...
```

You are now ready to define your neural network model.

2. Define Keras Model

Hint:

```
# Write the code to define the keras model
```

3. Compile Keras Model

Hint:

```
...
# Write the code to compile the keras model
```

4. Fit Keras Model

Hint:

```
...
# Fit the keras model on the dataset
model.fit(X, y, epochs=150, batch_size=10)
...
```

This is where the work happens on your CPU or GPU.

5. Evaluate Keras Model

Hint:

```
...
# Evaluate the keras model
_, accuracy = model.evaluate(X, y)
print('Accuracy: %.2f' % (accuracy*100))
```

6. Tie It All Together

Hint:

```
# first neural network with keras tutorial
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load the dataset, Split into input (X) and output (y) variables
# Define the keras model and compile the same

# Fit the keras model on the dataset

# Evaluate the keras model
```

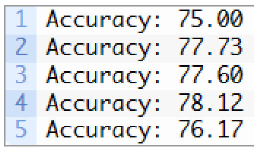
You can copy all the code into your Python file and save it as "**keras_first_network.py**" in the same directory as your data file "**pima-indians-diabetes.csv**". You can then run the Python file as a script from your command line (command prompt) as follows:

```
python keras_first_network.py
```

```
# Write the code to fit the keras model on the dataset without progress bars

# Write the code to evaluate the keras model
```

...

Note: Your results may vary given the stochastic nature of the algorithm or evaluation procedure, or differences in numerical precision. Consider running the example a few times and compare the average outcome.



```
Accuracy: 75.00
Accuracy: 77.73
Accuracy: 77.60
Accuracy: 78.12
Accuracy: 76.17
```

7. Make Predictions

Hint:

```
# make probability predictions with the model
predictions = model.predict(X)

# round predictions
rounded = [round(x[0]) for x in predictions]
```

Alternately, you can convert the probability into 0 or 1 to predict crisp classes directly; for example:

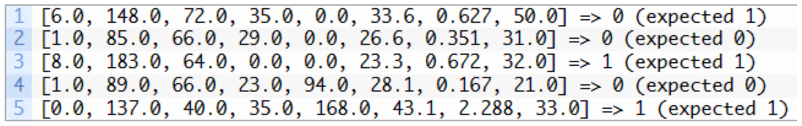
```
# make class predictions with the model
predictions = (model.predict(X) > 0.5).astype(int)
```

The complete example below makes predictions for each example in the dataset, then prints the input data, predicted class, and expected class for the first five examples in the dataset.

Hint:

```
# first neural network with keras make predictions

from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
# Load the dataset

dataset = loadtxt('pima-indians-diabetes.csv', delimiter=',')

# Split into input (X) and output (y) variables

X = dataset[:,0:8]
y = dataset[:,8]


# define the keras model

model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compile the keras model and fit the same on the training dataset

# Make class predictions with the model and summarize the first 5 cases
```



```
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 0 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)
```

14.2 Feed Forward Neural Network

Feed-forward neural networks are used to learn the relationship between independent variables, which serve as inputs to the network, and dependent variables that are designated as outputs of the network. The goal of this exercise is to design and train a feed-forward neural network, justify the multi-class classification, and obtain higher accuracy levels.

Input: MNIST Dataset

Output: Multi-Class Classification Testing Accuracy and Loss. Train the model to obtain > 90% accuracy on the dataset.

Explanation:

Today, we'll be using the *full* MNIST dataset, consisting of 70,000 data points (7,000 examples per digit). Each data point is represented by a 784-d vector, corresponding to the (flattened) *28×28* images in the MNIST dataset. Our goal is to train a neural network (using Keras) to obtain > *90%* accuracy on this dataset.

To get started, open a new file, name it keras_mnist.py, and insert the following code:

Hint:

```
# Import the necessary packages
```

```
# Implementing feedforward neural networks with Keras and TensorFlow
# Construct the argument parse and parse the arguments
```

```
# Implementing feedforward neural networks with Keras and TensorFlow
# Grab the MNIST dataset (if this is your first time using this
# dataset then the 11MB download may take a minute)
print("[INFO] accessing MNIST...")
((trainX, trainY), (testX, testY)) = mnist.load_data()
# each image in the MNIST dataset is represented as a 28x28x1
# image, but to apply a standard neural network we must
# first "flatten" the image to be simple list of 28x28=784 pixels
trainX = trainX.reshape((trainX.shape[0], 28 * 28 * 1))
testX = testX.reshape((testX.shape[0], 28 * 28 * 1))
# scale data to the range of [0, 1]
```

```
# Implementing feedforward neural networks with Keras and TensorFlow
# Convert the labels from integers to vectors

```

Here is a second example, this time with the label 1 binarized:

```
Implementing feedforward neural networks with Keras and TensorFlow
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
Implementing feedforward neural networks with Keras and TensorFlow
0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

Next, let's define our network architecture:

Hint:

```
# Define the 784-256-128-10 architecture using Keras

```

Let's go ahead and train our network:

Hint:

```
# train the model using SGD
print("[INFO] training network...")
sgd = SGD(0.01)
```

```
model.compile(loss="categorical_crossentropy", optimizer=sgd,
metrics=["accuracy"])
H = model.fit(trainX, trainY, validation_data=(testX, testY),
epochs=100, batch_size=128)
```

Once the network has finished training, we'll want to evaluate it on the testing data to obtain our final classifications:

Hint:

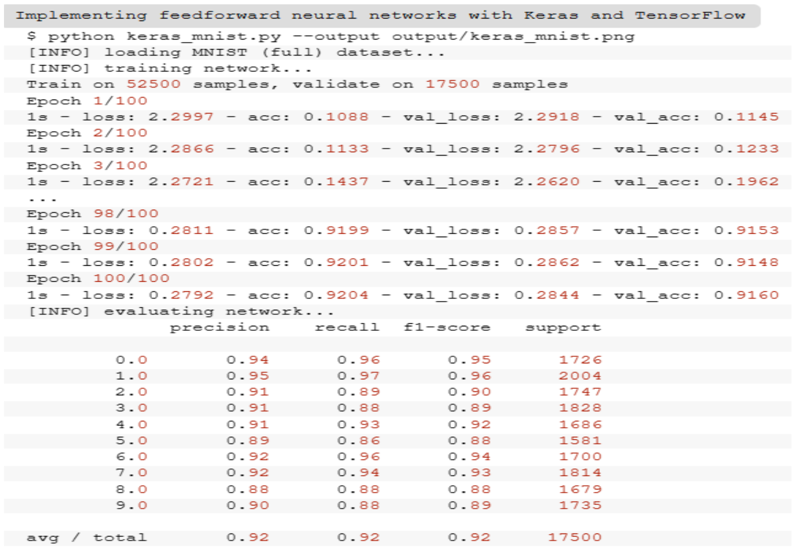
```
# Implementing feedforward neural networks with Keras and TensorFlow
# Evaluate the network
```

Our final code block handles plotting the training loss, training accuracy, validation loss, and validation accuracy over time:

Hint:

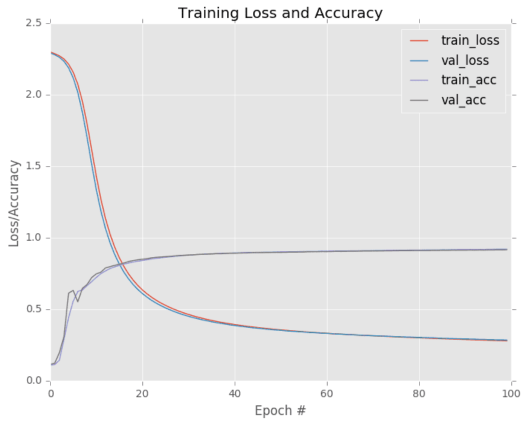
```
Implementing feedforward neural networks with Keras and TensorFlow
# Plot the training loss and accuracy
```

This plot is then saved to disk based on the –output command line argument. To train our network of fully connected layers on MNIST, just execute the following command:



```
Implementing feedforward neural networks with Keras and TensorFlow
 $ python keras_mnist.py --output output/keras_mnist.png
 [INFO] loading MNIST (full) dataset...
 [INFO] training network...
 Train on 52500 samples, validate on 17500 samples
 Epoch 1/100
 1s - loss: 2.2997 - acc: 0.1088 - val_loss: 2.2918 - val_acc: 0.1145
 Epoch 2/100
 1s - loss: 2.2866 - acc: 0.1133 - val_loss: 2.2796 - val_acc: 0.1233
 Epoch 3/100
 1s - loss: 2.2721 - acc: 0.1437 - val_loss: 2.2620 - val_acc: 0.1962
 ...
 Epoch 98/100
 1s - loss: 0.2811 - acc: 0.9199 - val_loss: 0.2857 - val_acc: 0.9153
 Epoch 99/100
 1s - loss: 0.2802 - acc: 0.9201 - val_loss: 0.2862 - val_acc: 0.9148
 Epoch 100/100
 1s - loss: 0.2792 - acc: 0.9204 - val_loss: 0.2844 - val_acc: 0.9160
 [INFO] evaluating network...
              precision    recall  f1-score   support

         0.0       0.94      0.96      0.95      1726
         1.0       0.95      0.97      0.96      2004
         2.0       0.91      0.89      0.90      1747
         3.0       0.91      0.88      0.89      1828
         4.0       0.91      0.93      0.92      1686
         5.0       0.89      0.86      0.88      1581
         6.0       0.92      0.96      0.94      1700
         7.0       0.92      0.94      0.93      1814
         8.0       0.88      0.88      0.88      1679
         9.0       0.90      0.88      0.89      1735

 avg / total       0.92      0.92      0.92     17500
```

As the results demonstrate, we are obtaining ≈*92%* accuracy. Furthermore, the training and validation curves match each other *nearly identically*, indicating there is no overfitting or issues with the training process.



In fact, if you are unfamiliar with the MNIST dataset, you might think 92% accuracy is *excellent* — and it was, perhaps 20 years ago. Using Convolutional Neural Networks, we can easily obtain > *98%* accuracy. Current state-of-the-art approaches can even break 99% accuracy.

While on the surface it may appear that our (strictly) fully connected network is performing well, we can do much better.

Try: Implement the same code on CIFAR-10 dataset and get the same or higher accuracy.

14.3 Back Propagation

Sometimes you need to improve the accuracy of your neural network model, and backpropagation helps you achieve the desired accuracy. The backpropagation algorithm helps you to get a good prediction of your neural network model. The goal of this exercise is to apply the backpropagation technique and observe how it fine-tune the weight function and improve the accuracy of the model.

Input: Iris Dataset

Output: Improve the accuracy of the model more than 90%.

Explanation:

The backpropagation algorithm is a type of supervised learning algorithm for artificial neural networks where we fine-tune the weight functions and improve the accuracy of the model. It employs the gradient descent method to reduce the cost function. It reduces the mean-squared distance between the predicted and the actual data. This type of algorithm is generally used for training feed-forward neural networks for a given data whose classifications are known to us.

You can also think of backward propagation as the backward spread of errors to achieve more accuracy. If we have received a prediction from a neural network model which has a huge difference from the actual output, we need to apply the backpropagation algorithm to achieve higher accuracy.

Note: Feed-forward neural networks are generally multi-layered neural networks (MLN). The data travels from the input layer to the hidden layer to the output layer.
Now let's get the intuition about how the algorithm works. There are mainly three layers in a backpropagation model i.e. input layer, hidden layer, and output layer. Following are the main steps of the algorithm:

- **Step 1**: The input layer receives the input.
- **Step 2:** The input is then averaged overweight's.
- **Step 3**: Each hidden layer processes the output. Each output is referred to as "Error" here which is the difference between the actual output and the desired output.
- **Step 4**: In this step, the algorithm moves back to the hidden layers again to optimize the weights and reduce the error.

Implementing Backpropagation in Python

Hint:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Now let's look at what dataset we will be working with.

Hint:

```
# Loading dataset
data = load_iris()

# Dividing the dataset into target variable and features
X=data.data
y=data.target
```

```
# Split dataset into training and test sets
```

```
learning_rate = 0.1
iterations = 5000
N = y_train.size

# Define the Input features, Hidden layers, and Output layer as well.
```

Initialize Weights

Hint:

```
np.random.seed(10)

# Hidden layer

W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))

# Output layer
```

```
W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
```

Now we will create helper functions such as mean squared error, accuracy and sigmoid.

```
# Helper functions

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
    # One-hot encode y_true (i.e., convert [0, 1, 2] into [[1, 0, 0], [0, 1, 0],
[0, 0, 1]])
    y_true_one_hot = np.eye(output_size)[y_true]

    # Reshape y_true_one_hot to match y_pred shape
    y_true_reshaped = y_true_one_hot.reshape(y_pred.shape)

    # Compute the mean squared error between y_pred and y_true_reshaped
    error = ((y_pred - y_true_reshaped)**2).sum() / (2*y_pred.size)

    return error

def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) ==  y_true.argmax(axis=1)
    return acc.mean()

results = pd.DataFrame(columns=["mse", "accuracy"])
```

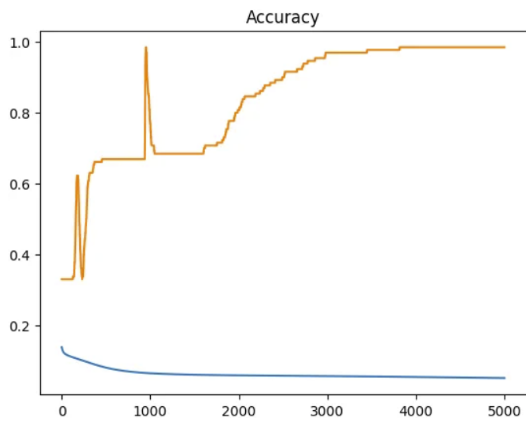
Now we will start building our backpropagation model.

Building the Backpropagation Model in Python

Now we will plot the mean squared error and accuracy using the pandas plot() function.

Hint:

```
results.mse.plot(title="Mean Squared Error")
plt.show()
results.accuracy.plot(title="Accuracy")
plt.show()
```



Now we will calculate the accuracy of the model.

```
# Write the code to test the model and display the accuracy
```

Output:

```
Accuracy: 0.95
```

You can see the accuracy of the model have been significantly increased to 80%.

14.4 Deep Neural Networks

Artificial Intelligence is a technique which enables machines to mimic human behavior. The idea behind AI is simple yet fascinating, which is to make intelligent machines that can take decisions on its own. For years, it was thought that computers would never match the power of the human brain. The goal of this exercise is to develop a deep neural network.

Input: MNIST Dataset

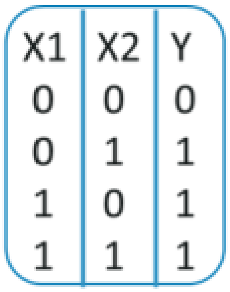
Class 1: Inputs having output as 0 that lies below the decision line.
Class 2: Inputs having output as 1 that lies above the decision line or separator.

Output: Accuracy in identifying the digits present on the image.

Explanation:

Deep Learning with Python: Perceptron Example

Now I'm sure you guys must be familiar with the working of the "**OR"** gate. The output is **1** if any of the inputs is also **1.**

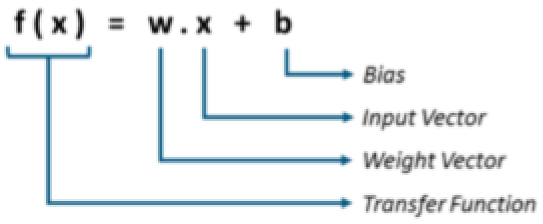


X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Therefore, a Perceptron can be used as a separator or a decision line that divides the input set of OR Gate, into two classes:

Class 1: Inputs having output as 0 that lies below the decision line.
Class 2: Inputs having output as 1 that lies above the decision line or separator.

Till now, we understood that a linear perceptron can be used to classify the input data set into two classes. But, how does it actually classify the data?



Mathematically a perceptron can be thought of like an equation of Weights, Inputs, and Bias.

Step 1: Import all the required library

Hint:

```
import tensorflow as tf
```

Step 2: Define Vector Variables for Input and Output

Hint:

```
train_in = [

[0,0,1],

[0,1,1],

[1,0,1],

[1,1,1]]

train_out = [
[0],

[1],

[1],

[1]]
```

Step 3: Define Weight Variable

Hint:

```
w = tf.Variable(tf.random_normal([3, 1], seed=15))
```

Step 4: Define placeholders for Input and Output

We need to define placeholders so that they can accept external inputs on the run.

Hint:

```
x = tf.placeholder(tf.float32,[None,3])

y = tf.placeholder(tf.float32,[None,1])
```

Step 5: Calculate Output and Activation Function

Hint:

```
output = tf.nn.relu(tf.matmul(x, w))
```

Step 6: Calculate the Cost or Error

Hint:

```
# Write the code to calculate the cost or error
```

Step 7: Minimize Error

Hint:

```
optimizer = tf.train.GradientDescentOptimizer(0.01)

train = optimizer.minimize(loss)
```

Step 8: Initialize all the variables

Hint:

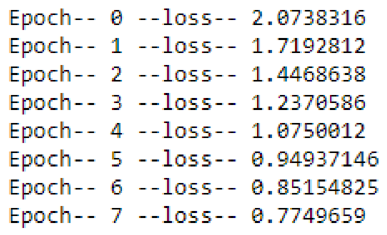
```
# Write the code to initialize all the variables
```

Step 9: Training Perceptron in Iterations

Hint:

```
# Write the code to perform the training perception in iterations
```

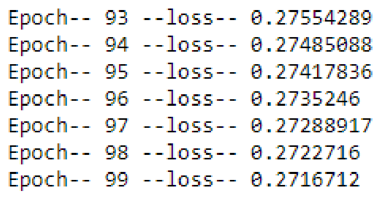
Step 10: Output



```
Epoch-- 0 --loss-- 2.0738316
Epoch-- 1 --loss-- 1.7192812
Epoch-- 2 --loss-- 1.4468638
Epoch-- 3 --loss-- 1.2370586
Epoch-- 4 --loss-- 1.0750012
Epoch-- 5 --loss-- 0.94937146
Epoch-- 6 --loss-- 0.85154825
Epoch-- 7 --loss-- 0.7749659
```

……

……



```
Epoch-- 93 --loss-- 0.27554289
Epoch-- 94 --loss-- 0.27485088
Epoch-- 95 --loss-- 0.27417836
Epoch-- 96 --loss-- 0.2735246
Epoch-- 97 --loss-- 0.27288917
Epoch-- 98 --loss-- 0.2722716
Epoch-- 99 --loss-- 0.2716712
```

As you can see here, the loss started at **2.07** and ended at **0.27**

Deep Learning with Python: Creating a Deep Neural Network

```
from __future__ import print_function
```

Following is the code with comments at every step:

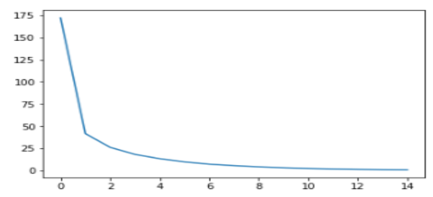
Hint:

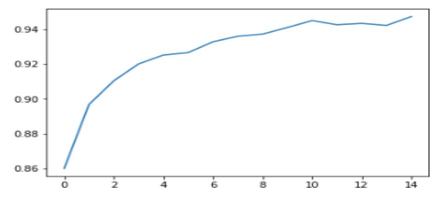
```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
import tensorflow as tf
import matplotlib.pyplot as plt
# Parameters
learning_rate = 0.001
training_epochs = 15
batch_size = 100
display_step = 1
# Network Parameters
```

```
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])
# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with RELU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
# Construct model
pred = multilayer_perceptron(x, weights, biases)
# Define loss, optimizer and initialize the variables
# Create an empty list to store the cost history and accuracy history
cost_history = []
accuracy_history = []
# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size)

            # Run optimization op (backprop) and cost op (to get loss value)
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_x,y: batch_y})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:

            correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
            # Calculate accuracy
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
            acu_temp = accuracy.eval({x: mnist.test.images, y: mnist.test.labels})
            #Append the accuracy to the list
            accuracy_history.append(acu_temp)
            #Append the cost history
            cost_history.append(avg_cost)
            print ("Epoch:", '%04d' % (epoch + 1), "- cost=",
"{:.9f}".format(avg_cost), "- Accuracy=",acu_temp)
     print ("Optimization Finished!")
    # Write the code to plot the cost history and accuracy history


    # Write the code to test the model and calculate the accuracy
```

Output:

```
Epoch: 0001 - cost= 171.935939952 - Accuracy= 0.86
Epoch: 0002 - cost= 41.690122643 - Accuracy= 0.8967
Epoch: 0003 - cost= 26.223513353 - Accuracy= 0.9103
Epoch: 0004 - cost= 18.315846960 - Accuracy= 0.92
Epoch: 0005 - cost= 13.400080771 - Accuracy= 0.9251
Epoch: 0006 - cost= 9.863584718 - Accuracy= 0.9265
Epoch: 0007 - cost= 7.281175789 - Accuracy= 0.9327
Epoch: 0008 - cost= 5.611238315 - Accuracy= 0.9359
Epoch: 0009 - cost= 4.224623066 - Accuracy= 0.9371
Epoch: 0010 - cost= 3.170401931 - Accuracy= 0.9409
Epoch: 0011 - cost= 2.357616258 - Accuracy= 0.9449
Epoch: 0012 - cost= 1.807826444 - Accuracy= 0.9425
Epoch: 0013 - cost= 1.375618323 - Accuracy= 0.9433
Epoch: 0014 - cost= 1.113478136 - Accuracy= 0.9421
Epoch: 0015 - cost= 0.935187823 - Accuracy= 0.9472
Optimization Finished!
```





```
Accuracy: 0.9472
```

Try: Apply the code for at least three different datasets and compare the results.

14.5 Non-Linear Activation Functions

Activation functions play an integral role in neural networks by introducing nonlinearity. This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear regression model. Many different nonlinear activation functions have been proposed throughout the history of neural networks. The goal of this exercise is to explore three popular ones: sigmoid, tanh, and ReLU implement at least three activation functions and perform the comparative analysis.

Input: Non-linear activation functions like ReLU, Sigmoid, and Hyperbolic Tangent.

Output: Comparative Analysis of activation functions

Explanation:

You might be wondering, why all this hype about nonlinear activation functions? Or why can't we just use an identity function after the weighted linear combination of activations from the previous layer? Using multiple linear layers is basically the same as using a single linear layer. This can be seen through a simple example.

Sigmoid Function and Vanishing Gradient

In TensorFlow, you can call the sigmoid function from the Keras library as follows:

Hint:

```
import tensorflow as tf
from tensorflow.keras.activations import sigmoid
 input_array = tf.constant([-1, 0, 1], dtype=tf.float32)
print (sigmoid(input_array))
```

This gives the following output:

```
tf.Tensor([0.26894143 0.5 0.7310586 ], shape=(3,), dtype=float32)
```

Hyperbolic Tangent Function

In TensorFlow, you can implement the tanh activation on a tensor using the tanh function in Keras's activations module:

Hint:

```
# Write the code to implement tanh activation function and display the output
```

Rectified Linear Unit (ReLU)

Next up, you can also look at the gradient of the ReLU function:

To use the ReLU activation in TensorFlow:

```
# Write the code to implement ReLU activation function and display the output
```

This gives the following output:

```
tf.Tensor([0. 0. 1.], shape=(3,), dtype=float32)
```

Using Activation Functions in Practice

```
x = Dense(units=10)(input_layer)

x = relu(x)
```

However, for many Keras layers, you can also use a more compact representation to add the activation on top of the layer:

```
x = Dense(units=10, activation=”relu”)(input_layer)
```

Using this more compact representation, let’s build our LeNet5 model using Keras:

Hint:

```
import tensorflow as tf
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense, Input, Flatten, Conv2D,
BatchNormalization, MaxPool2D
from tensorflow.keras.models import Model
 (trainX, trainY), (testX, testY) = keras.datasets.cifar10.load_data()
 input_layer = Input(shape=(32,32,3,))
x = Conv2D(filters=6, kernel_size=(5,5), padding="same",
activation="relu")(input_layer)
x = MaxPool2D(pool_size=(2,2))(x)
x = Conv2D(filters=16, kernel_size=(5,5), padding="same", activation="relu")(x)
x = MaxPool2D(pool_size=(2, 2))(x)
x = Conv2D(filters=120, kernel_size=(5,5), padding="same", activation="relu")(x)
x = Flatten()(x)
x = Dense(units=84, activation="relu")(x)
x = Dense(units=10, activation="softmax")(x)
model = Model(inputs=input_layer, outputs=x)

# Write the code to display the model summary

# Write the code to compile, fit, and validate the model
```

14.6 Generative Adversarial Networks (GAN)

Generative models can also be used with labeled datasets. When they are, they’re trained to learn the probability P(x|y) of the input x given the output y. They can also be used for classification tasks, but in general, discriminative models perform better when it comes to classification. The goal of this exercise is to generate a high dimensional sample space using generative adversarial networks.

Input: MNIST Handwritten Dataset
Output: Generate High Dimensional Samples
Explanation:

Generative adversarial networks can also generate high-dimensional samples such as images. In this example, you're going to use a GAN to generate images of handwritten digits. For that, you'll train the models using the MNIST dataset of handwritten digits, which is included in the torchvision package.

To begin, you need to install torchvision in the activated gan conda environment:

```
$ conda install -c pytorch torchvision=0.5.0
```

As in the previous example, you start by importing the necessary libraries:

Hint:

```
import torch
from torch import nn
import math
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
```

Besides the libraries you've imported before, you're going to need torchvision and transforms to obtain the training data and perform image conversions.

Again, set up the random generator seed to be able to replicate the experiment:

```
torch.manual_seed(111)
```

```
device = ""
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")
```

Preparing the Training Data

```
transform = transforms.Compose(

    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]

)
```

The function has two parts:

1. **transforms.ToTensor()** converts the data to a PyTorch tensor.
2. **transforms.Normalize()** converts the range of the tensor coefficients.

Now you can load the training data using torchvision.datasets.MNIST and perform the conversions using transform:

```
train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)
```

Now that you've created train_set, you can create the data loader as you did before:

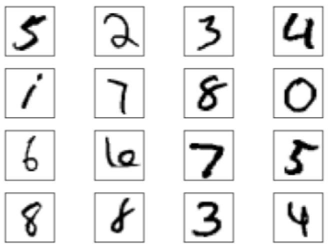
```
batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

To improve the visualization, you can use cmap=gray_r to reverse the color map and plot the digits in black over a white background:

Hint:

```
real_samples, mnist_labels = next(iter(train_loader))
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(real_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])
```

The output should be something similar to the following:



Implementing the Discriminator and the Generator

In this case, the discriminator is an MLP neural network that receives a 28 × 28 pixel image and provides the probability of the image belonging to the real training data. You can define the model with the following code:

```
 class Discriminator(nn.Module):
     def __init__(self):
         super().__init__()
         self.model = nn.Sequential(
             nn.Linear(784, 1024),
             nn.ReLU(),
             nn.Dropout(0.3),
             nn.Linear(1024, 512),
             nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output
```

To run the discriminator model using the GPU, you must instantiate it and send it to the GPU with .to(). To use a GPU when there’s one available, you can send the model to the device object you created earlier:

```
discriminator = Discriminator().to(device=device)
```

Hint:

```
 class Generator(nn.Module):
     def __init__(self):
         super().__init__()
         self.model = nn.Sequential(
             nn.Linear(100, 256),
             nn.ReLU(),
             nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output

generator = Generator().to(device=device)
```

Training the Models

To train the models, you need to define the training parameters and optimizers like you did in the previous example:

```
lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()

optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)
```

To obtain a better result, you decrease the learning rate from the previous example. You also set the number of epochs to 50 to reduce the training time. The training loop is very similar to the one you used in the previous example. In the highlighted lines, you send the training data to device to use the GPU if available:

Hint:

```
 for epoch in range(num_epochs):
    for n, (real_samples, mnist_labels) in enumerate(train_loader):
        # Data for training the discriminator
        real_samples = real_samples.to(device=device)
        real_samples_labels = torch.ones((batch_size, 1)).to(
            device=device
        )
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )
        generated_samples = generator(latent_space_samples)
        generated_samples_labels = torch.zeros((batch_size, 1)).to(
            device=device
        )
        all_samples = torch.cat((real_samples, generated_samples))
        all_samples_labels = torch.cat(
            (real_samples_labels, generated_samples_labels)
        )

        # Training the discriminator
        discriminator.zero_grad()
        output_discriminator = discriminator(all_samples)
        loss_discriminator = loss_function(
            output_discriminator, all_samples_labels
        )
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Data for training the generator
    latent_space_samples = torch.randn((batch_size, 100)).to(
      device=device
    )
        # Training the generator
        generator.zero_grad()
        generated_samples = generator(latent_space_samples)
        output_discriminator_generated = discriminator(generated_samples)
        loss_generator = loss_function(
            output_discriminator_generated, real_samples_labels
        )
        loss_generator.backward()
        optimizer_generator.step()

        # Show loss
        if n == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")
```

Since this example features more complex models, the training may take a bit more time. After it finishes, you can check the results by generating some samples of handwritten digits.

Checking the Samples Generated by the GAN

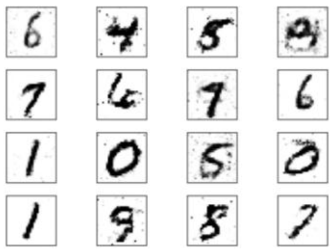
To generate handwritten digits, you have to take some random samples from the latent space and feed them to the generator:

```
latent_space_samples = torch.randn(batch_size, 100).to(device=device)
generated_samples = generator(latent_space_samples)
```

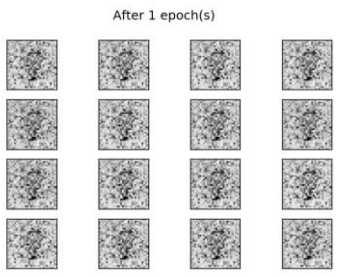
To plot generated_samples, you need to move the data back to the CPU in case it's running on the GPU. For that, you can simply call .cpu(). As you did previously, you also need to call .detach() before using Matplotlib to plot the data:

```
generated_samples = generated_samples.cpu().detach()
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(generated_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])
```

The output should be digits resembling the training data, as in the following figure:



After fifty epochs of training, there are several generated digits that resemble the real ones. You can improve the results by considering more training epochs. As with the previous example, by using a fixed latent space samples tensor and feeding it to the generator at the end of each epoch during the training process, you can visualize the evolution of the training:



You can see that at the beginning of the training process, the generated images are completely random. As the training progresses, the generator learns the distribution of the real data, and at about twenty epochs, some generated digits already resemble real data.

Try: Define the training parameters and optimizers and run the model for at least 100 epochs by decreasing the learning rate.

15. Final Notes

The only way to learn programming is program, program, and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests. Check out these sites:

- Industry-Curated Hackathon: Machine Hack (AI Hackathons | MachineHack Generative AI)
- Kaggle Competition for Data Science (Kaggle: Your Machine Learning and Data Science Community)
- Brainstorming Data Science Challenges (Contests | Analytics Vidhya).
- Code Lab Online Data Science Challenges (CodaLab - Competitions).
- Coding Challenges Platform (https://www.topcoder.com/challenges).
- Data Driven Science Online Competition (Competitions (drivendata.org))
- New Science Competition (https://www.icfpconference.org/)
- New Algorithmic Competition (AIcrowd)
- Tianchi Big Data Science Competition (4 Data Science Competition Platforms Other Than Kaggle | by Edwin Tan | Towards Data Science)
- Data Visualization Competition (Iron Viz | Win or learn—you can't lose (tableau.com))
- Other Competitions (ML Contests)

Student must have any one of the following certifications:

1. LearnBay – Machine Learning Course for Professions
2. AWS Certified Machine Learning - Specialty Certification
3. Andrew Ng's Machine Learning Specialization
4. IBM Machine Learning Professional Certification
5. Google Professional Machine Learning Engineer Certification
6. University of Washington Machine Learning Specialization

V. TEXT BOOKS:

1. Introduction to Machine Learning with Python: A Guide for Data Scientists" by Andreas C. Müller and Sarah Guido
2. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron

VI. REFERENCE BOOKS:

1. Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar, "Foundations of Machine Learning", MIT Press,2nd edition, 2018.
2. Trevor Hastie, Robert Tibshirani, Jerome Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction", Springer, 2nd edition, 2009.
3. Avrim Blum, John Hopcroft, Ravindran Kannan, "Foundations of Data Science", Cambridge UniversityPress, 2020.
4. Tom M. Mitchell, "Machine Learning", Mc Graw Hill, Indian edition, 2017.
5. Gareth James, Daniela Witten, Trevor Hastie and Rob Tibshirani, "An Introduction to Statistical Learning: with applications in R", Springer Texts in Statistics, 2017.

VII. ELECTRONIC RESOURCES

1. https://onlinecourses.nptel.ac.in/noc19_cs52/preview
2. https://ece.iisc.ac.in/~parimal/2019/ml.html
3. https://www.springer.com/gp/book/9780387848570
4. https://www.cse.iitb.ac.in/~sunita/cs725/calendar.html
5. https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/
6. https://cs.nyu.edu/~mohri/mlu11/

VIII. ELECTRONIC RESOURCES

1. Course Content
2. Lab Manual