A PRACTICAL REPORT

ON

MACHINE LEARNING

SUBMITTED BY

Mr. Malpura Mohd Azharuddin Mohd Rafiq

Submitted in fulfillment of the requirements for qualifying
M.Sc.IT Part-2 Semester-3 Examination 2025-2026

University of Mumbai

Department of Information Technology

University of Mumbai

Institute of Distance and Open Learning (IDOL)

Dr. Shankardayal Sharma bhavan, Vidyanagari, Santacruz(E)

PCP CENTER: RIZVI COLLEGE, BANDRA (W)

Certificate

*This is to certify that **MACHINE LEARNING** performed at RIZVI COLLEGE, BANDRA (W) by Mr. **Malpura Mohd Azharuddin Mohd Rafiq** holding Seat No/Application No. 8163020 /6780 studying Masters of Science in Information Technology Semester – 3 has been satisfactorily completed as prescribed by the University of Mumbai, during the year 2025 – 2026.*

Subject In-Charge **Coordinator In-Charge** **External Examiner**

INDEX

Sr. No	Practical List	Date	Sign
1	Data Preprocessing and Exploration. a) Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers. b) Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note: Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization. c) Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.		
2	Testing Hypothesis. a) Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset)		
3	Linear Models. a) Simple Linear Regression Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE. b) Multiple Linear Regression Extend linear regression to multiple features. Handle feature selection and potential multicollinearity. c) Regularized Linear Models (Ridge, Lasso, Elastic Net). ❖ Implement regression variants like LASSO and Ridge on any generated dataset. o Linear Regression. The Normal Equation. o Regularised Linear Model. Ridge Regression.		

	○ Lasso Regression. Elastic Net.		
4	Discriminative Models. a) Logistic Regression Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve. b) Implement and demonstrate k-nearest Neighbour algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions. c) Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree. d) Implement a Support Vector Machine for any relevant dataset. e) Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree. f) Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.		
5	Generative Models. a) Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample. b) Implement Hidden Markov Models using hmmlearn.		
6	Probabilistic Models. a) Implement Bayesian Linear Regression to explore prior and posterior distribution. b) Implement Gaussian Mixture Models for density estimation and unsupervised clustering.		
7	Model Evaluation and Hyperparameter Tuning. a) Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation.		

	b) Systematically explore combinations of hyperparameters to optimize model performance. (use grid and randomized search) (Refer Logistic Regression)		
8	Bayesian Learning. a) Bayesian Inference Learning.		
9	Deep Generative Models. a) Set up a generator network to produce samples and a discriminator network to distinguish between real and generated data. (Use a simple small dataset)		

Practical 1

Aim: Data Preprocessing and Exploration.

a. Load a CSV dataset. Handle missing values, inconsistent formatting, and outliers.

Theory:

Load the data from the file using the techniques learned today. Examine it.

Determine the following:

- The number of data points (rows). (*Hint:* check out the dataframe, shape attribute.)
- The column names. (*Hint:* check out the dataframe. columns attribute.)
- The data types for each column. (*Hint:* check out the dataframe, dtypes attribute.)

Code:

```
data = pd.read_csv("iris_data.csv")

data.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
# Number of rows

print(data.shape[0])


# Column names

print(data.columns.tolist())
```

Data types

print(data.dtypes)

```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'species']
sepal_length    float64
sepal_width     float64
petal_length    float64
petal_width     float64
species          object
dtype: object
```

Examine the species names and note that they all begin with 'Iris-'. Remove this portion of the name so the species name is shorter.

(Hint: there are multiple ways to do this, but you could use either the string processing methods or the apply method.)

The str method maps the following function to each entry as a string

data['species'] = data.species.str.replace('Iris-', '')

Alternatively

data['species'] = data.species.apply(lambda r: r.replace('Iris-', ''))

data.head()

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Determine the following:

- The number of each species present. (Hint: check out the series .value_counts method.)
- The mean, median, and quantiles and ranges (max-min) for each petal and sepal measurement.

Hint: for the last question, the. describe method does have median, but it's not called median. It's the 50% quantile. describe does not have range though, and in order to get the range, you will need to create a new entry in the. describe table, which is max - min.

One way to count each species

```
data.species.value_counts()
```

```
versicolor    50
setosa        50
virginica     50
Name: species, dtype: int64
```

Select just the rows desired from the 'describe' method and add in the 'median'.

```
stats_df = data.describe()

stats_df = data.describe()

stats_df.loc['range'] = stats_df.loc['max'] - stats_df.loc['min']

out_fields = ['mean','25%','50%','75%', 'range']

stats_df = stats_df.loc[out_fields]

stats_df.rename({'50%': 'median'}, inplace=True)

stats_df
```

	sepal_length	sepal_width	petal_length	petal_width
mean	5.843333	3.054	3.758667	1.198667
25%	5.100000	2.800	1.600000	0.300000
median	5.800000	3.000	4.350000	1.300000
75%	6.400000	3.300	5.100000	1.800000
range	3.600000	2.400	5.900000	2.400000

If certain fields need to be aggregated differently, we can do:

```
from pprint import pprint
agg_dict = {field: ['mean', 'median'] for field in data.columns if field != 'species'}
agg_dict['petal_length'] = 'max'
pprint(agg_dict)
data.groupby('species').agg(agg_dict)
```

```
{'petal_length': 'max',
 'petal_width': ['mean', 'median'],
 'sepal_length': ['mean', 'median'],
 'sepal_width': ['mean', 'median']}
```

	sepal_length		sepal_width		petal_length	petal_width	
	mean	median	mean	median	max	mean	median
species							
setosa	5.006	5.0	3.418	3.4	1.9	0.244	0.2
versicolor	5.936	5.9	2.770	2.8	5.1	1.326	1.3
virginica	6.588	6.5	2.974	3.0	6.9	2.026	2.0

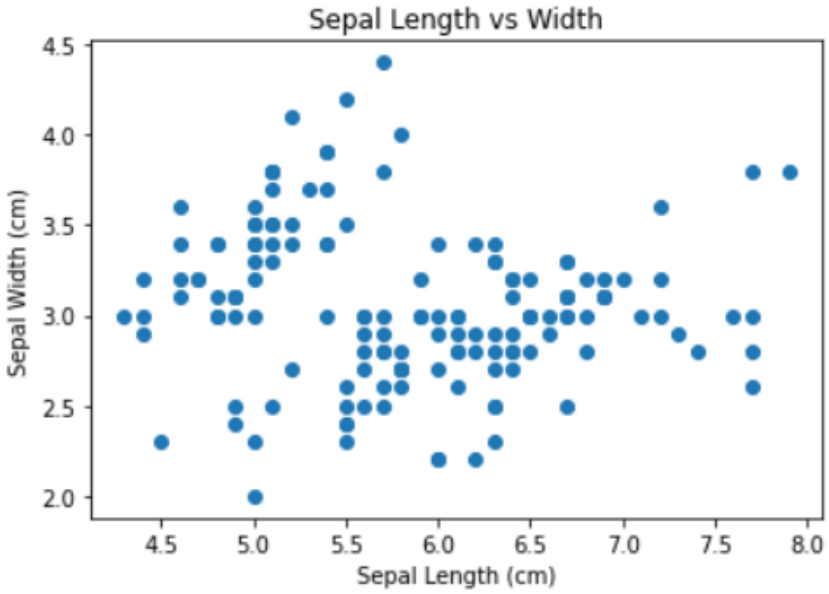
b. Load a dataset, calculate descriptive summary statistics, create visualizations using different graphs, and identify potential features and target variables Note: Explore Univariate and Bivariate graphs (Matplotlib) and Seaborn for visualization.

Theory: Make a scatter plot of sepal_length vs sepal_width using Matplotlib. Label the axes and give the plot a title.

Code:

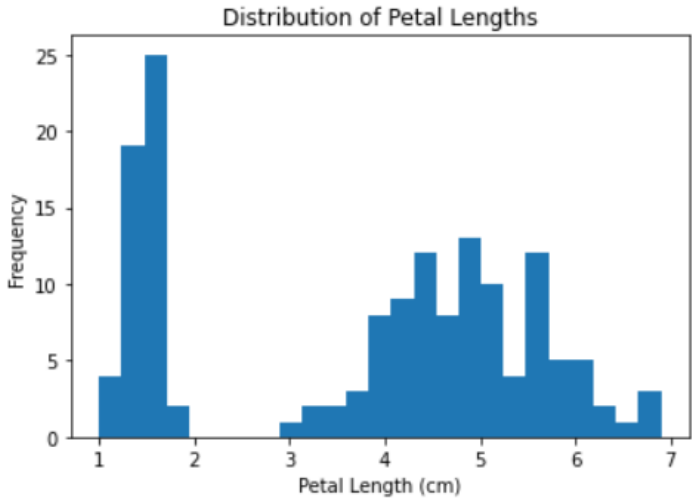
```
import matplotlib.pyplot as plt
%matplotlib inline

# A simple scatter plot with Matplotlib
ax = plt.axes()
```

```
ax.scatter(data.sepal_length, data.sepal_width)


# Label the axes
ax.set(xlabel='Sepal Length (cm)',
       ylabel='Sepal Width (cm)',
       title='Sepal Length vs Width');
```

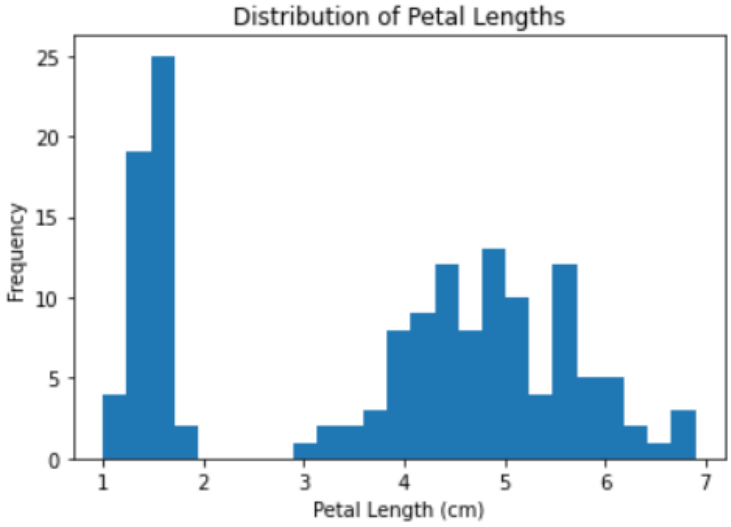


Make a histogram of any one of the four features. Label axes and title it as appropriate.

```
# Using Matplotlib's plotting functionality


ax = plt.axes()
ax.hist(data.petal_length, bins=25);
ax.set(xlabel='Petal Length (cm)',
       ylabel='Frequency',
       title='Distribution of Petal Lengths')
```



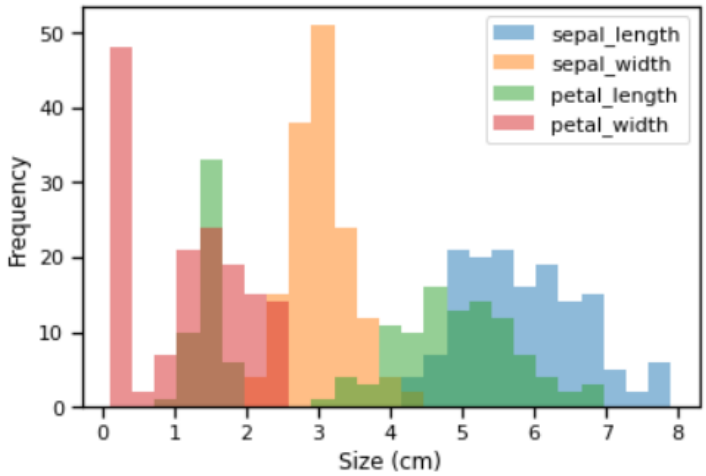

```
# Alternatively using Pandas plotting functionality
ax = data.petal_length.plot.hist(bins=25)

ax.set(xlabel='Petal Length (cm)',
       ylabel='Frequency',
       title='Distribution of Petal Lengths');
```



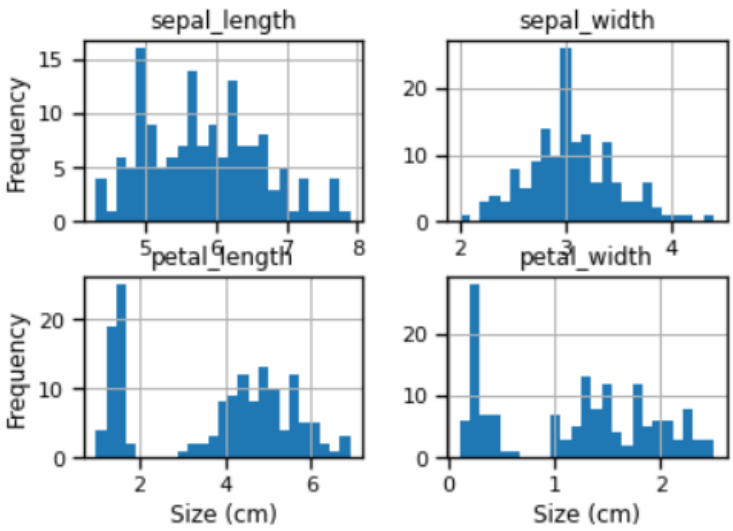
Now create a single plot with histograms for each feature (petal_width, petal_length, sepal_width, sepal_length) overlayed. If you have time, next try to create four individual histogram plots in a single figure, where each plot contains one feature.

For some hints on how to do this with Pandas plotting methods, check out the visualization guide for Pandas.

```
import seaborn as sns

sns.set_context('notebook')

### BEGIN SOLUTION

# This uses the `.plot.hist` method

ax = data.plot.hist(bins=25, alpha=0.5)

ax.set_xlabel('Size (cm)');
```



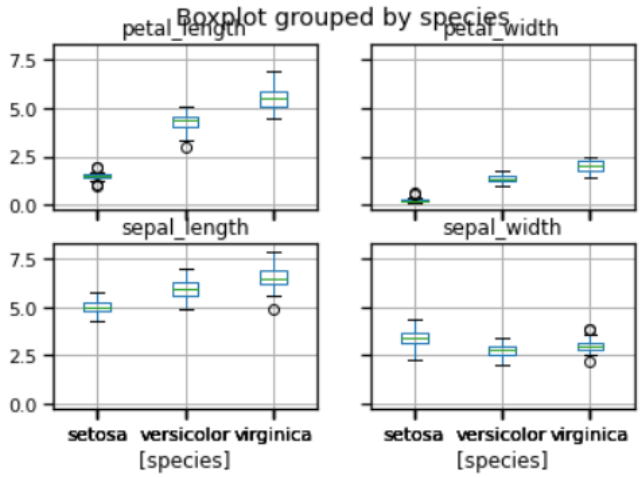

```
# To create four separate plots, use Pandas `.hist` method

axList = data.hist(bins=25)


# Add some x- and y- labels to first column and last row

for ax in axList.flatten():

    if ax.is_last_row():

        ax.set_xlabel('Size (cm)')


    if ax.is_first_col():

        ax.set_ylabel('Frequency')
```



Using Pandas, make a boxplot of each petal and sepal measurement. Here is the documentation for Pandas boxplot method.

Here we have four separate plots

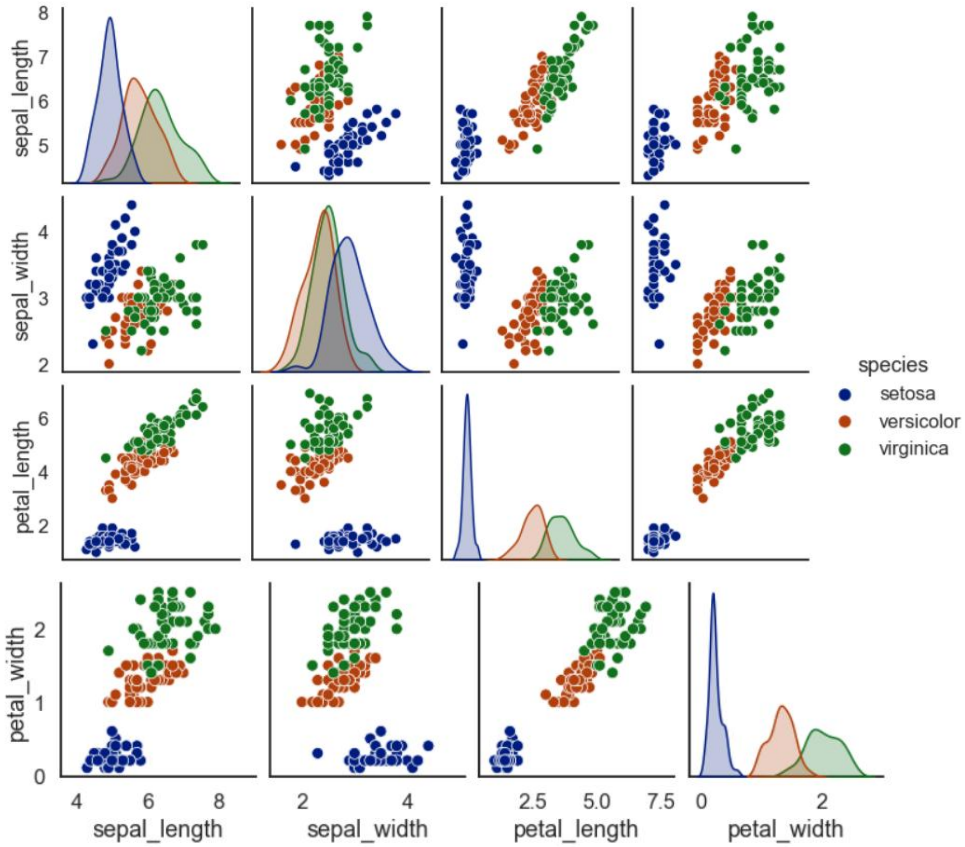
data.boxplot(by='species');



Make a pair plot with Seaborn to examine the correlation between each of the measurements.

(Hint: this plot may look complicated, but it is actually only a single line of code. This is the power of Seaborn and dataframe-aware plotting! See the lecture notes for reference.)

```
sns.set_context('talk')
sns.pairplot(data, hue='species');
```



c. Create or Explore datasets to use all pre-processing routines like label encoding, scaling, and binarization.

Theory:

Feature Engineering

```
# Load in the Ames Housing Data.
URL = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML0232EN-SkillsNetwork/asset/Ames_Housing_Data.tsv'

#await skillsnetwork.download_dataset(URL)
df = pd.read_csv('Ames_Housing_Data.tsv', sep='\t')
## Examine the columns, look at missing data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2930 entries, 0 to 2929
Data columns (total 82 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Order            2930 non-null   int64
 1   PID              2930 non-null   int64
 2   MS SubClass      2930 non-null   int64
 3   MS Zoning        2930 non-null   object
 4   Lot Frontage     2440 non-null   float64
 5   Lot Area         2930 non-null   int64
 6   Street           2930 non-null   object
 7   Alley            198 non-null    object
 8   Lot Shape        2930 non-null   object
 9   Land Contour     2930 non-null   object
 10  Utilities        2930 non-null   object
 11  Lot Config       2930 non-null   object
 12  Land Slope       2930 non-null   object
 13  Neighborhood     2930 non-null   object
 14  Condition 1      2930 non-null   object
 15  Condition 2      2930 non-null   object
 16  Bldg Type        2930 non-null   object
 17  House Style      2930 non-null   object
 18  Overall Qual     2930 non-null   int64
 19  Overall Cond     2930 non-null   int64
 20  Year Built       2930 non-null   int64
 21  Year Remod/Add   2930 non-null   int64
 22  Roof Style       2930 non-null   object
 23  Roof Matl        2930 non-null   object
 24  Exterior 1st     2930 non-null   object
 25  Exterior 2nd     2930 non-null   object
 26  Mas Vnr Type     2907 non-null   object
 27  Mas Vnr Area     2907 non-null   float64
 28  Exter Qual       2930 non-null   object
 29  Exter Cond       2930 non-null   object
 30  Foundation       2930 non-null   object
 31  Bsmt Qual        2850 non-null   object
 32  Bsmt Cond        2850 non-null   object
 33  Bsmt Exposure    2847 non-null   object
 34  BsmtFin Type 1   2850 non-null   object
 35  BsmtFin SF 1     2929 non-null   float64
 36  BsmtFin Type 2   2849 non-null   object
 37  BsmtFin SF 2     2929 non-null   float64
 38  Bsmt Unf SF      2929 non-null   float64
 39  Total Bsmt SF    2929 non-null   float64
 40  Heating          2930 non-null   object
 41  Heating QC       2930 non-null   object
 42  Central Air      2930 non-null   object
 43  Electrical       2929 non-null   object
 44  1st Flr SF       2930 non-null   int64
 45  2nd Flr SF       2930 non-null   int64
 46  Low Qual Fin SF  2930 non-null   int64
 47  Gr Liv Area      2930 non-null   int64
 48  Bsmt Full Bath   2928 non-null   float64
 49  Bsmt Half Bath   2928 non-null   float64
 50  Full Bath        2930 non-null   int64
 51  Half Bath        2930 non-null   int64
 52  Bedroom AbvGr    2930 non-null   int64
 53  Kitchen AbvGr    2930 non-null   int64
 54  Kitchen Qual     2930 non-null   object
 55  TotRms AbvGrd    2930 non-null   int64
 56  Functional       2930 non-null   object
 57  Fireplaces       2930 non-null   int64
 58  Fireplace Qu     1508 non-null   object
 59  Garage Type      2773 non-null   object
 60  Garage Yr Blt    2771 non-null   float64
```

```
 61  Garage Finish    2771 non-null   object
 62  Garage Cars      2929 non-null   float64
 63  Garage Area      2929 non-null   float64
 64  Garage Qual      2771 non-null   object
 65  Garage Cond      2771 non-null   object
 66  Paved Drive      2930 non-null   object
 67  Wood Deck SF     2930 non-null   int64
 68  Open Porch SF    2930 non-null   int64
 69  Enclosed Porch   2930 non-null   int64
 70  3Ssn Porch       2930 non-null   int64
 71  Screen Porch     2930 non-null   int64
 72  Pool Area        2930 non-null   int64
 73  Pool QC          13 non-null     object
 74  Fence            572 non-null    object
 75  Misc Feature     106 non-null    object
 76  Misc Val         2930 non-null   int64
 77  Mo Sold          2930 non-null   int64
 78  Yr Sold          2930 non-null   int64
 79  Sale Type        2930 non-null   object
 80  Sale Condition   2930 non-null   object
 81  SalePrice        2930 non-null   int64
dtypes: float64(11), int64(28), object(43)
memory usage: 1.8+ MB
```

This is recommended by the data set author to remove a few outliers.

```
df = df.loc[df['Gr Liv Area'] <= 4000,:]
print("Number of rows in the data:", df.shape[0])
print("Number of columns in the data:", df.shape[1])
data = df.copy()
```

Keep a copy our original data

```
Number of rows in the data: 2925
Number of columns in the data: 82
```

A quick look at the data:

```
df.head()
```

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Normal
1	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Normal
2	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Normal
3	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Normal
4	5	527105010	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	...	0	NaN	MnPrv	NaN	0	3	2010	WD	Normal

5 rows × 82 columns

Get a Pd.Series consisting of all the string categoricals

```
one_hot_encode_cols = df.dtypes[df.dtypes == object]  # filtering by string categoricals
```

```
one_hot_encode_cols = one_hot_encode_cols.index.tolist()  # list of categorical fields
df[one_hot_encode_cols].head().T
```

	0	1	2	3	4
MS Zoning	RL	RH	RL	RL	RL
Street	Pave	Pave	Pave	Pave	Pave
Alley	NaN	NaN	NaN	NaN	NaN
Lot Shape	IR1	Reg	IR1	Reg	IR1
Land Contour	Lvl	Lvl	Lvl	Lvl	Lvl
Utilities	AllPub	AllPub	AllPub	AllPub	AllPub
Lot Config	Corner	Inside	Corner	Corner	Inside
Land Slope	Gtl	Gtl	Gtl	Gtl	Gtl
Neighborhood	NAmes	NAmes	NAmes	NAmes	Gilbert
Condition 1	Norm	Feedr	Norm	Norm	Norm
Condition 2	Norm	Norm	Norm	Norm	Norm
Bldg Type	1Fam	1Fam	1Fam	1Fam	1Fam
House Style	1Story	1Story	1Story	1Story	2Story
Roof Style	Hip	Gable	Hip	Hip	Gable
Roof Matl	CompShg	CompShg	CompShg	CompShg	CompShg
Exterior 1st	BrkFace	VinylSd	Wd Sdng	BrkFace	VinylSd
Exterior 2nd	Plywood	VinylSd	Wd Sdng	BrkFace	VinylSd
Mas Vnr Type	Stone	None	BrkFace	None	None
Exter Qual	TA	TA	TA	Gd	TA
Exter Cond	TA	TA	TA	TA	TA
Foundation	CBlock	CBlock	CBlock	CBlock	PConc

Foundation	CBlock	CBlock	CBlock	CBlock	PConc
Bsmt Qual	TA	TA	TA	TA	Gd
Bsmt Cond	Gd	TA	TA	TA	TA
Bsmt Exposure	Gd	No	No	No	No
BsmtFin Type 1	BLQ	Rec	ALQ	ALQ	GLQ
BsmtFin Type 2	Unf	LwQ	Unf	Unf	Unf
Heating	GasA	GasA	GasA	GasA	GasA
Heating QC	Fa	TA	TA	Ex	Gd
Central Air	Y	Y	Y	Y	Y
Electrical	SBrkr	SBrkr	SBrkr	SBrkr	SBrkr
Kitchen Qual	TA	TA	Gd	Ex	TA
Functional	Typ	Typ	Typ	Typ	Typ
Fireplace Qu	Gd	NaN	NaN	TA	TA
Garage Type	Attchd	Attchd	Attchd	Attchd	Attchd
Garage Finish	Fin	Unf	Unf	Fin	Fin
Garage Qual	TA	TA	TA	TA	TA
Garage Cond	TA	TA	TA	TA	TA
Paved Drive	P	Y	Y	Y	Y
Pool QC	NaN	NaN	NaN	NaN	NaN
Fence	NaN	MnPrv	NaN	NaN	MnPrv

We're going to first do some basic data cleaning on this data:

- Converting categorical variables to dummies
- Making skew variables symmetric

One-hot encoding the dummy variables:

Do the one hot encoding

df = pd.get_dummies(df, columns=one_hot_encode_cols, drop_first=True)

df.describe().T

Out[14]:

	count	mean	std	min	25%	50%	75%	max
Order	2925.0	1.464795e+03	8.464417e+02	1.0	732.0	1463.0	2199.0	2.930000e+03
PID	2925.0	7.143931e+08	1.887274e+08	526301100.0	528477030.0	535453210.0	907180130.0	1.007100e+09
MS SubClass	2925.0	5.739658e+01	4.266875e+01	20.0	20.0	50.0	70.0	1.900000e+02
Lot Frontage	2435.0	6.902382e+01	2.271092e+01	21.0	58.0	68.0	80.0	3.130000e+02
Lot Area	2925.0	1.010358e+04	7.781999e+03	1300.0	7438.0	9428.0	11515.0	2.152450e+05
...	...	...	...	...	...	...	...	...
Sale Condition_AdjLand	2925.0	4.102564e-03	6.393067e-02	0.0	0.0	0.0	0.0	1.000000e+00
Sale Condition_Alloca	2925.0	8.205128e-03	9.022520e-02	0.0	0.0	0.0	0.0	1.000000e+00
Sale Condition_Family	2925.0	1.572650e-02	1.244366e-01	0.0	0.0	0.0	0.0	1.000000e+00
Sale Condition_Normal	2925.0	8.246154e-01	3.803608e-01	0.0	1.0	1.0	1.0	1.000000e+00
Sale Condition_Partial	2925.0	8.273504e-02	2.755284e-01	0.0	0.0	0.0	0.0	1.000000e+00

262 rows × 8 columns

There are a *lot* of variables. Let's go back to our saved original data and look at how many values are missing for each variable.

df = data

data.isnull().sum().sort_values()

```
Out[20]: Order              0
         Sale Condition     0
         Heating QC         0
         Central Air        0
         1st Flr SF         0
                           ...
         Fireplace Qu    1422
         Fence           2354
         Alley           2727
         Misc Feature    2820
         Pool QC         2914
         Length: 82, dtype: int64
```

Let's pick out just a few numeric columns to illustrate basic feature transformations.

```
smaller_df= df.loc[:,['Lot Area', 'Overall Qual', 'Overall Cond',
                      'Year Built', 'Year Remod/Add', 'Gr Liv Area',
                      'Full Bath', 'Bedroom AbvGr', 'Fireplaces',
                      'Garage Cars','SalePrice']]
```

Now we can look at summary statistics of the subset data.

smaller_df.describe().T

	count	mean	std	min	25%	50%	75%	max
Lot Area	2925.0	10103.583590	7781.999124	1300.0	7438.0	9428.0	11515.0	215245.0
Overall Qual	2925.0	6.088205	1.402953	1.0	5.0	6.0	7.0	10.0
Overall Cond	2925.0	5.563761	1.112262	1.0	5.0	5.0	6.0	9.0
Year Built	2925.0	1971.302906	30.242474	1872.0	1954.0	1973.0	2001.0	2010.0
Year Remod/Add	2925.0	1984.234188	20.861774	1950.0	1965.0	1993.0	2004.0	2010.0
Gr Liv Area	2925.0	1493.978803	486.273646	334.0	1126.0	1441.0	1740.0	3820.0
Full Bath	2925.0	1.564786	0.551386	0.0	1.0	2.0	2.0	4.0
Bedroom AbvGr	2925.0	2.853675	0.827737	0.0	2.0	3.0	3.0	8.0
Fireplaces	2925.0	0.596923	0.645349	0.0	0.0	1.0	1.0	4.0
Garage Cars	2924.0	1.765048	0.759834	0.0	1.0	2.0	2.0	5.0
SalePrice	2925.0	180411.574701	78554.857286	12789.0	129500.0	160000.0	213500.0	625000.0

smaller_df.info()

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2925 entries, 0 to 2929
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Lot Area        2925 non-null   int64
 1   Overall Qual    2925 non-null   int64
 2   Overall Cond    2925 non-null   int64
 3   Year Built      2925 non-null   int64
 4   Year Remod/Add  2925 non-null   int64
 5   Gr Liv Area     2925 non-null   int64
 6   Full Bath       2925 non-null   int64
 7   Bedroom AbvGr   2925 non-null   int64
 8   Fireplaces      2925 non-null   int64
 9   Garage Cars     2924 non-null   float64
 10  SalePrice       2925 non-null   int64
dtypes: float64(1), int64(10)
memory usage: 274.2 KB
```

There appears to be one NA in Garage Cars - we will take a simple approach and fill it with 0.

smaller_df = smaller_df.fillna(0)

smaller_df.info()

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2925 entries, 0 to 2929
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Lot Area        2925 non-null   int64
 1   Overall Qual    2925 non-null   int64
 2   Overall Cond    2925 non-null   int64
 3   Year Built      2925 non-null   int64
 4   Year Remod/Add  2925 non-null   int64
 5   Gr Liv Area     2925 non-null   int64
 6   Full Bath       2925 non-null   int64
 7   Bedroom AbvGr   2925 non-null   int64
 8   Fireplaces      2925 non-null   int64
 9   Garage Cars     2925 non-null   float64
 10  SalePrice       2925 non-null   int64
dtypes: float64(1), int64(10)
memory usage: 274.2 KB
```

- Objectives

  After completing this lab you will be able to:

  - Use Log function to transform the data
  - Handle the duplicates
  - Handle the missing values
  - Standardize and normalize the data
  - Handle the outliers

- Setup

  For this lab, we will be using the following libraries:

  - pandas for managing the data.
  - numpy for mathematical operations.
  - seaborn for visualizing the data.
  - matplotlib for visualizing the data.
  - sklearn for machine learning and machine-learning-pipeline related functions.
  - scipy for statistical computations.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pylab as plt
%matplotlib inline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from scipy.stats import norm
from scipy import stats
```

Reading and understanding our data

For this lab, we will be using the Ames_Housing_Data.tsv file, hosted on IBM Cloud object storage. The Ames housing dataset examines features of houses sold in Ames (a small city in the state of Iowa in the United States) during the 2006–2010 timeframe.

Let's read the data into pandas data frame and look at the first 5 rows using the head() method.

In [4]:

```
housing = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML0232EN-SkillsNetwork/asset/Ames_Housing_Data1.tsv", sep='\t')
```

```
housing.head(10)
```

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Normal
1	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Normal
2	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Normal
3	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Normal
4	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Normal
5	5	527105010	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	...	0	NaN	MnPrv	NaN	0	3	2010	WD	Normal
6	6	527105030	60	RL	78.0	9978	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	6	2010	WD	Normal
7	7	527127150	120	RL	41.0	4920	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Normal
8	8	527145080	120	RL	43.0	5005	Pave	NaN	IR1	HLS	...	0	NaN	NaN	NaN	0	1	2010	WD	Normal
9	9	527146030	120	RL	39.0	5389	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	3	2010	WD	Normal

10 rows × 82 columns

```
housing["SalePrice"].describe()
```

```
count      2931.000000
mean     180807.729785
std       79875.557267
min       12789.000000
25%      129500.000000
50%      160000.000000
75%      213500.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

From the above analysis, it is important to note that the minimum value is greater than 0. Also, there is a big difference between the minimum value and the 25th percentile. It is bigger than the 75th percentile and the maximum value. This means that our data might not be normally distributed (an important assumption for linear regression analysis), so will check for normality in the Log Transform section.

Looking for Correlations¶

Before proceeding with the data cleaning, it is useful to establish a correlation between the response variable (in our case the sale price) and other predictor variables, as some of them might not have any major impact in determining the price of the house and will not be used in the analysis. There are many ways to discover correlation between the target variable and the rest of the features. Building pair plots, scatter plots, heat maps, and a correlation matrixes are the most common ones. Below, we will use the corr() function to list the top features based on the pearson correlation

coefficient (measures how closely two sequences of numbers are correlated). Correlation coefficient can only be calculated on the numerical attributes (floats and integers), therefore, only the numeric attributes will be selected.

```
hous_num = housing.select_dtypes(include = ['float64', 'int64'])

hous_num_corr = hous_num.corr()['SalePrice'][:-1] # -1 means that the latest row is SalePrice

top_features = hous_num_corr[abs(hous_num_corr) > 0.5].sort_values(ascending=False) #displays pearsons correlation coefficient greater than 0.5

print("There is {} strongly correlated values with SalePrice:\n{}".format(len(top_features), top_features))
```

```
There is 11 strongly correlated values with SalePrice:
Overall Qual      0.799226
Gr Liv Area       0.706791
Garage Cars       0.647891
Garage Area       0.640411
Total Bsmt SF     0.632270
1st Flr SF        0.621672
Year Built        0.558340
Full Bath         0.545339
Year Remod/Add    0.532664
Garage Yr Blt     0.526808
Mas Vnr Area      0.508277
Name: SalePrice, dtype: float64
```

Above, there are 11 features, with coefficients greater than 0.5, that are strongly correlated with the sale price.

Handling the Duplicates

As mentioned in the video, having duplicate values can effect our analysis, so it is good to check whether there are any duplicates in our data. We will use pandas duplicated() function and search by the 'PID' column, which contains a unique index number for each entry.

```
duplicate = housing[housing.duplicated(['PID'])]

duplicate
```

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition	S
1	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Normal	

1 rows × 82 columns

As we can see, there is one duplicate row in this dataset. To remove it, we can use pandas drop_duplicates() function. By default, it removes all duplicate rows based on all the columns.

dup_removed = housing.drop_duplicates()

dup_removed

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sa Conditio
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Norm
2	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Norm
3	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Norm
4	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Norm
5	5	527105010	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	...	0	NaN	MnPrv	NaN	0	3	2010	WD	Norm
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
2926	2926	923275080	80	RL	37.0	7937	Pave	NaN	IR1	Lvl	...	0	NaN	GdPrv	NaN	0	3	2006	WD	Norm
2927	2927	923276100	20	RL	NaN	8885	Pave	NaN	IR1	Low	...	0	NaN	MnPrv	NaN	0	6	2006	WD	Norm
2928	2928	923400125	85	RL	62.0	10441	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	Shed	700	7	2006	WD	Norm
2929	2929	924100070	20	RL	77.0	10010	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2006	WD	Norm
2930	2930	924151050	60	RL	74.0	9627	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	11	2006	WD	Norm

2930 rows × 82 columns

An alternative way to check if there are any duplicated Indexes in our dataset is using index.is_unique function.

housing.index.is_unique

True

In this exercise try to remove duplicates on a specific column by setting the subset equal to the column that contains the duplicate, such as 'Order'.

Enter your code and run the cell

removed_sub = housing.drop_duplicates(subset=['Order'])

Handling the Missing Values

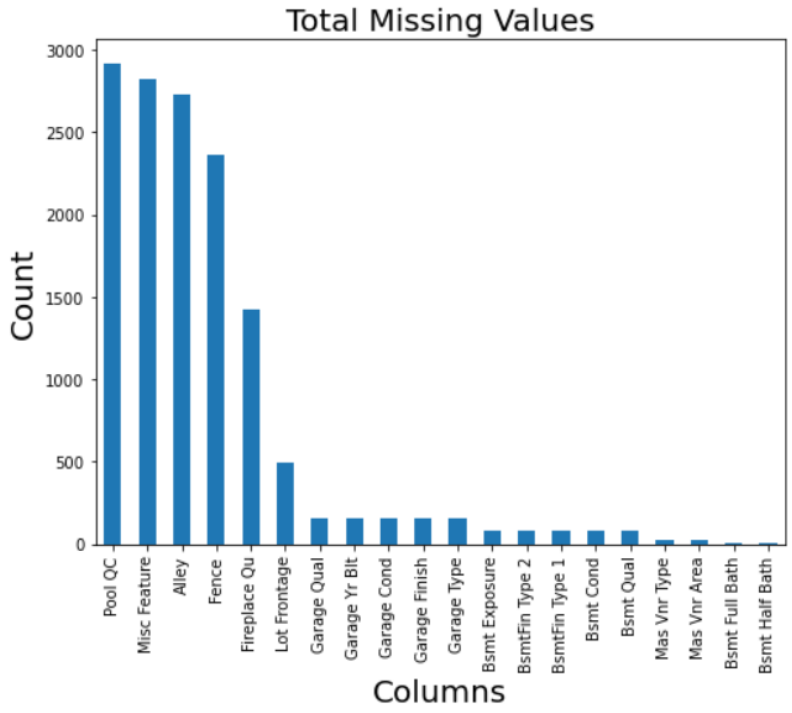
Finding the Missing Values

For easier detection of missing values, pandas provides the isna(), isnull(), and notna() functions. For more information on pandas missing values please check out this documentation.

To summarize all the missing values in our dataset, we will use isnull() function. Then, we will add them all up, by using sum() function, sort them with sort_values() function, and plot the first 20 columns (as the majority of our missing values fall within first 20 columns), using the bar plot function from the matplotlib library.

```
total = housing.isnull().sum().sort_values(ascending=False)
total_select = total.head(20)
total_select.plot(kind="bar", figsize = (8,6), fontsize = 10)

plt.xlabel("Columns", fontsize = 20)
plt.ylabel("Count", fontsize = 20)
plt.title("Total Missing Values", fontsize = 20)
```

```
Text(0.5, 1.0, 'Total Missing Values')
```



There are several options for dealing with missing values. We will use 'Lot Frontage' feature to analyze for missing values.

1. We can drop the missing values, using dropna() method.

housing.dropna(subset=["Lot Frontage"])

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sa Conditic
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Norm
1	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Norm
2	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Norm
3	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Norm
4	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Norm
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
2925	2925	923251180	20	RL	160.0	20000	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	9	2006	WD	Abnorı
2926	2926	923275080	80	RL	37.0	7937	Pave	NaN	IR1	Lvl	...	0	NaN	GdPrv	NaN	0	3	2006	WD	Norm
2928	2928	923400125	85	RL	62.0	10441	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	Shed	700	7	2006	WD	Norm
2929	2929	924100070	20	RL	77.0	10010	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2006	WD	Norm
2930	2930	924151050	60	RL	74.0	9627	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	11	2006	WD	Norm

Using this method, all the rows, containing null values in 'Lot Frontage' feature, for example, will be dropped.

2. We can drop the whole attribute (column), that contains missing values, using the drop() method.

housing.drop("Lot Frontage", axis=1)

	Order	PID	MS SubClass	MS Zoning	Lot Area	Street	Alley	Lot Shape	Land Contour	Utilities	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition
0	1	526301100	20	RL	31770	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	5	2010	WD	Norma
1	1	526301100	20	RL	31770	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	5	2010	WD	Norma
2	2	526350040	20	RH	11622	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Norma
3	3	526351010	20	RL	14267	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Norma
4	4	526353030	20	RL	11160	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	4	2010	WD	Norma
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2926	2926	923275080	80	RL	7937	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	GdPrv	NaN	0	3	2006	WD	Norma
2927	2927	923276100	20	RL	8885	Pave	NaN	IR1	Low	AllPub	...	0	NaN	MnPrv	NaN	0	6	2006	WD	Norma
2928	2928	923400125	85	RL	10441	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv	Shed	700	7	2006	WD	Norma
2929	2929	924100070	20	RL	10010	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	4	2006	WD	Norma
2930	2930	924151050	60	RL	9627	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	11	2006	WD	Norma

2931 rows × 81 columns

Using this method, the entire column containing the null values will be dropped.

3. We can replace the missing values (zero, the mean, the median, etc.), using fillna() method.

```
In [24]: median = housing["Lot Frontage"].median()
         median
Out[24]: 68.0

In [25]: housing["Lot Frontage"].fillna(median, inplace = True)

In [26]: housing.tail()
```

Out[26]:

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sale Condition
2926	2926	923275080	80	RL	37.0	7937	Pave	NaN	IR1	Lvl	...	0	NaN	GdPrv	NaN	0	3	2006	WD	Norma
2927	2927	923276100	20	RL	68.0	8885	Pave	NaN	IR1	Low	...	0	NaN	MnPrv	NaN	0	6	2006	WD	Norma
2928	2928	923400125	85	RL	62.0	10441	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	Shed	700	7	2006	WD	Norma
2929	2929	924100070	20	RL	77.0	10010	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2006	WD	Norma
2930	2930	924151050	60	RL	74.0	9627	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	11	2006	WD	Norma

5 rows × 82 columns

Index# 2927, containing a missing value in the "Lot Frontage", now has been replaced with the median value.

In this exercise, let's look at 'Mas Vnr Area' feature and replace the missing values with the mean value of that column.

```
mean = housing["Mas Vnr Area"].mean()

housing["Mas Vnr Area"].fillna(mean, inplace = True)
```

Feature Scaling

One of the most important transformations we need to apply to our data is feature scaling. There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

Min-max scaling (or normalization) is the simplest: values are shifted and rescaled so they end up ranging from 0 to 1. This is done by subtracting the min value and dividing by the max minus min.

Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation, so that the resulting distribution has unit variance.

Scikit-learn library provides MinMaxScaler for normalization and StandardScaler for standardization needs. For more information on scikit-learn MinMaxScaler and StandardScaler please visit their respective documentation websites.

First, we will normalize our data.

```
In [27]: norm_data = MinMaxScaler().fit_transform(hous_num)
         norm_data

Out[27]: array([[0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
                 3.63636364e-01, 1.00000000e+00, 2.72444089e-01],
                [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, ...,
                 3.63636364e-01, 1.00000000e+00, 2.72444089e-01],
                [3.41413452e-04, 1.01788895e-04, 0.00000000e+00, ...,
                 4.54545455e-01, 1.00000000e+00, 1.24238256e-01],
                ...,
                [9.99317173e-01, 8.25914814e-01, 3.82352941e-01, ...,
                 5.45454545e-01, 0.00000000e+00, 1.60616051e-01],
                [9.99658587e-01, 8.27370610e-01, 0.00000000e+00, ...,
                 2.72727273e-01, 0.00000000e+00, 2.11814430e-01],
                [1.00000000e+00, 8.27476641e-01, 2.35294118e-01, ...,
                 9.09090909e-01, 0.00000000e+00, 2.36066294e-01]])
```

Note the data is now and array.

we can also standardize our data.

```
In [28]: scaled_data = StandardScaler().fit_transform(hous_num)
         scaled_data
```

```
Out[28]: array([[-1.73027969, -0.99682434, -0.87674019, ..., -0.44796566,
                  1.67740664,  0.4281423 ],
                [-1.73027969, -0.99682434, -0.87674019, ..., -0.44796566,
                  1.67740664,  0.4281423 ],
                [-1.72909781, -0.99656498, -0.87674019, ..., -0.07945953,
                  1.67740664, -0.94923488],
                ...,
                [ 1.729097  ,  1.10758639,  0.64804102, ...,  0.2890466 ,
                 -1.36026952, -0.61115139],
                [ 1.73027889,  1.11129572, -0.87674019, ..., -0.81647179,
                 -1.36026952, -0.13533019],
                [ 1.73146077,  1.11156589,  0.06158671, ...,  1.76307112,
                 -1.36026952,  0.09005881]])
```

Handling the Outliers

Finding the Outliers

In statistics, an outlier is an observation point that is distant from other observations. An outlier can be due to some mistakes in data collection or recording, or due to natural high variability of data points. How to treat an outlier highly depends on our data or the type of analysis to be performed. Outliers can markedly affect our models and can be a valuable source of information, providing us insights about specific behaviours.

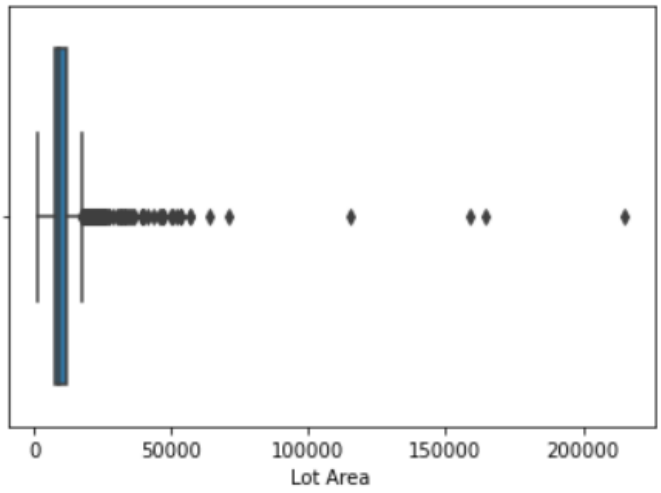
There are many ways to discover outliers in our data. We can do Uni-variate analysis (using one variable analysis) or Multi-variate analysis (using two or more variables). One of the simplest ways to detect an outlier is to inspect the data visually, by making box plots or scatter plots.

Uni-variate Analysis

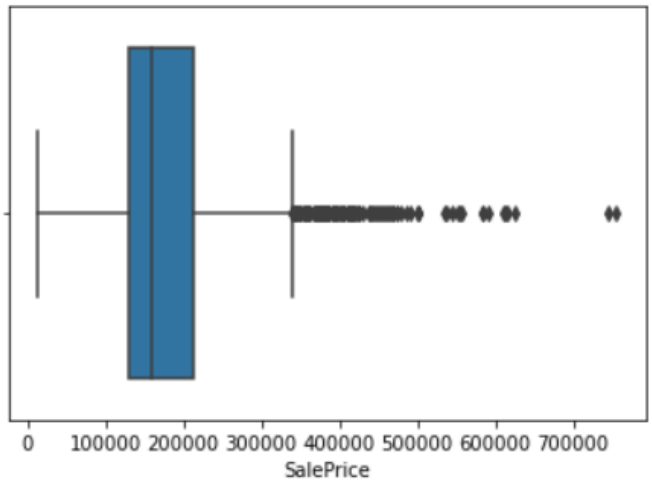
A box plot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles. Outliers may be plotted as individual points. To learn more about box plots please click here.

Here, we will use a box plot for the 'Lot Area' and the 'SalePrice' features.

```
In [30]: sns.boxplot(x=housing['Lot Area'])
Out[30]: <AxesSubplot:xlabel='Lot Area'>
```



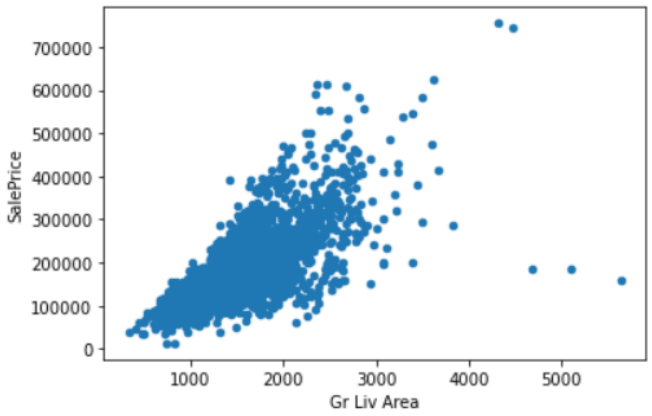

```
In [31]: sns.boxplot(x=housing['SalePrice'])
Out[31]: <AxesSubplot:xlabel='SalePrice'>
```



Bi-variate Analysis

Next, we will look at the bi-variate analysis of the two features, the sale price, 'SalePrice', and the ground living area, 'GrLivArea', and plot the scatter plot of the relationship between these two parameters.

```
price_area = housing.plot.scatter(x='Gr Liv Area',
                                  y='SalePrice')
```



From the above graph, there are two values above 5000 sq. ft. living area that deviate from the rest of the population and do not seem to follow the trend. It can be speculated why this is happening but for the purpose of this lab we can delete them.

The other two observations on the top are also deviating from the rest of the points but they also seem to be following the trend, so, perhaps, they can be kept.

Deleting the Outliers

First, we will sort all of our 'Gr Liv Area' values and select only the last two.

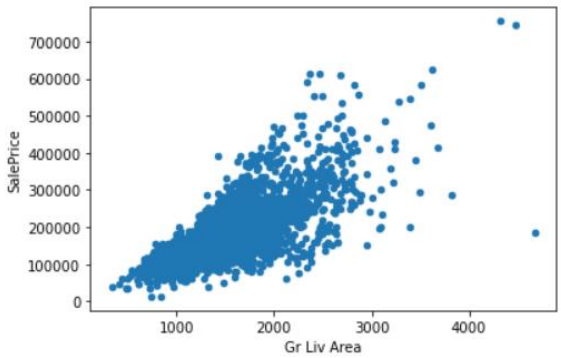
```
housing.sort_values(by = 'Gr Liv Area', ascending = False)[:2]
```

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sa Conditic
1499	1499	908154235	60	RL	313.0	63887	Pave	NaN	IR3	Bnk	...	480	Gd	NaN	NaN	0	1	2008	New	Parti
2181	2181	908154195	20	RL	128.0	39290	Pave	NaN	IR1	Bnk	...	0	NaN	NaN	Elev	17000	10	2007	New	Parti

2 rows × 82 columns

Now we will use the pandas drop() function to remove these two rows.

```
In [34]: outliers_dropped = housing.drop(housing.index[[1499,2181]])

In [35]: new_plot = outliers_dropped.plot.scatter(x='Gr Liv Area',
                                                  y='SalePrice')
```



As you can see, we do not have the last two points of the 'Gr Liv Area' anymore.

In this exercise, determine whether there are any outliers in the 'Lot Area' feature. You can either plot the box plot for the 'Lot Area', perform a bi-variate analysis by making a scatter plot between the 'SalePrice' and the 'Lot Area', or use the Z-score analysis. If there re any outliers, remove them from the dataset.

```
sns.boxplot(x=housing['Lot Area'])
price_lot = housing.plot.scatter(x='Lot Area', y='SalePrice')
housing['Lot_Area_Stats'] = stats.zscore(housing['Lot Area'])
housing[['Lot Area','Lot_Area_Stats']].describe().round(3)
housing.sort_values(by = 'Lot Area', ascending = False)[:1]
lot_area_rem = housing.drop(housing.index[[957]])
```

Z-score Analysis

Z-score is another way to identify outliers mathematically. Z-score is the signed number of standard deviations by which the value of an observation or data point is above the mean value of what is being observed or measured. In another words, Z-score is the value that quantifies relationship between a data point and a standard deviation and mean values of a group of points. Data points which are too far from zero will be treated as the outliers. In most of the cases, a threshold of 3 or -3 is used. For example, if the Z-score value is greater than or less than 3 or -3 standard deviations respectively, that data point will be identified as a outlier.

To learn more about Z-score, please visit this Wikipedia site.

Below, we are using Z-score function from scipy library to detect the outliers in our 'Low Qual Fin SF' parameter. To learn more about scipy.stats, please visit this link.

In [37]:
```
housing['LQFSF_Stats'] = stats.zscore(housing['Low Qual Fin SF'])
housing[['Low Qual Fin SF','LQFSF_Stats']].describe().round(3)
```

Out[37]:

	Low Qual Fin SF	LQFSF_Stats
count	2931.000	2931.000
mean	4.675	-0.000
std	46.303	1.000
min	0.000	-0.101
25%	0.000	-0.101
50%	0.000	-0.101
75%	0.000	-0.101
max	1064.000	22.882

The scaled results show a mean of 0.000 and a standard deviation of 1.000, indicating that the transformed values fit the z-scale model. The max value of 22.882 is further proof of the presence of outliers, as it falls well above the z-score limit of +3.

Practical 2

Aim: Testing Hypothesis.

Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a. CSV file and generate the final specific hypothesis. (Create your dataset).

Code:

```
import csv
hypo = ['%','%','%','%','%','%'];
with open('/content/drive/MyDrive/IDOL/Machine Learning/trainingdata.csv') as csv_file:
    readcsv = csv.reader(csv_file, delimiter=',')
    print(readcsv)
    data = []
    print("\nThe given training examples are:")
    for row in readcsv:
        print(row)
        if row[len(row)-1].upper() == "YES":
            data.append(row)
print("\nThe positive examples are:");
for x in data:
    print(x);
print("\n");
```

```
The positive examples are:
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes']
['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes']
['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
```

```
TotalExamples = len(data);
```

```
i=0;
j=0;
k=0;
print("The steps of the Find-s algorithm are :\n",hypo);
list = [];
p=0;
d=len(data[p])-1;
for j in range(d):
    list.append(data[i][j]);
hypo=list;
i=1;
for i in range(TotalExamples):
    for k in range(d):
        if hypo[k]!=data[i][k]:
            hypo[k]='?';
            k=k+1;
        else:
            hypo[k];
    print(hypo);
i=i+1;
```

```
The steps of the Find-s algorithm are :
 ['%', '%', '%', '%', '%', '%']
['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same']
['Sunny', 'Warm', '?', 'Strong', 'Warm', 'Same']
['Sunny', 'Warm', '?', 'Strong', '?', '?']
```

```
print("\nThe maximally specific Find-s hypothesis for the given training examples is :");
list=[];
for i in range(d):
    list.append(hypo[i]);
print(list);
```

```
The maximally specific Find-s hypothesis for the given training examples is :
['Sunny', 'Warm', '?', 'Strong', '?', '?']
```

Practical 3

Aim: Linear Models.

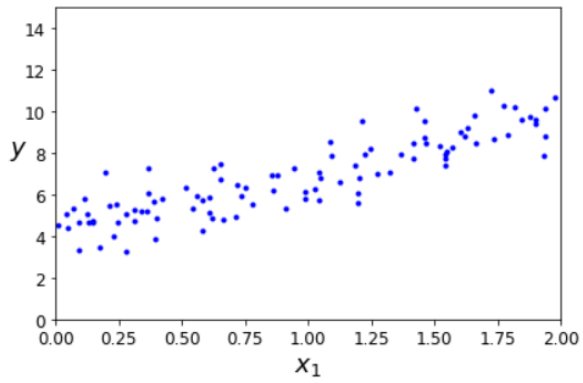
a. Simple Linear Regression Fit a linear regression model on a dataset. Interpret coefficients, make predictions, and evaluate performance using metrics like R-squared and MSE.
b. Multiple Linear Regression Extend linear regression to multiple features. Handle feature selection and potential multicollinearity.

Linear Regression

The Normal Equation

```
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15])
save_fig("generated_data_plot")
plt.show()
```

```
Saving figure generated_data_plot
```



In [4]:
```
X_b = np.c_[np.ones((100, 1)), X]  # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

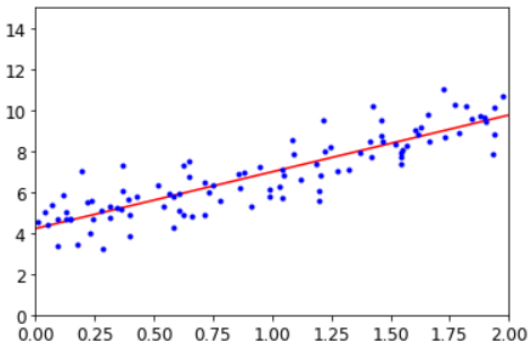
In [5]:
```
theta_best
```

Out[5]:
```
array([[4.21509616],
       [2.77011339]])
```

In [6]:
```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
y_predict
```

Out[6]:
```
array([[4.21509616],
       [9.75532293]])
```

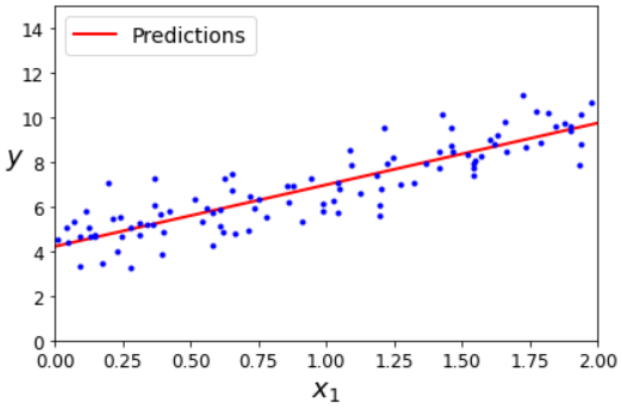
In [7]:
```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



The figure in the book actually corresponds to the following code, with a legend and axis labels:

```
plt.plot(X_new, y_predict, "r-", linewidth=2, label="Predictions")
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 2, 0, 15])
save_fig("linear_model_predictions_plot")
plt.show()
```

```
Saving figure linear_model_predictions_plot
```




```
In [9]: from sklearn.linear_model import LinearRegression

        lin_reg = LinearRegression()
        lin_reg.fit(X, y)
        lin_reg.intercept_, lin_reg.coef_

Out[9]: (array([4.21509616]), array([[2.77011339]]))

In [10]: lin_reg.predict(X_new)

Out[10]: array([[4.21509616],
                [9.75532293]])
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for "least squares"), which you could call directly:

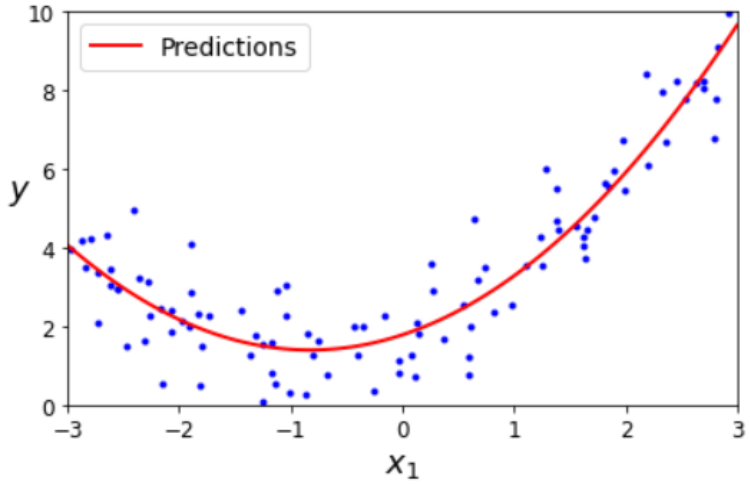
```
In [11]: theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
         theta_best_svd

Out[11]: array([[4.21509616],
                [2.77011339]])
```

This function computes $\mathbf{X}^{+}\mathbf{y}$, where $\mathbf{X}^{+}$ is the *pseudoinverse* of $\mathbf{X}$ (specifically the Moore-Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
In [12]: np.linalg.pinv(X_b).dot(y)

Out[12]: array([[4.21509616],
                [2.77011339]])
```

```
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
save_fig("quadratic_predictions_plot")
plt.show()
```

```
Saving figure quadratic_predictions_plot
```




```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0]
```

```
array([-0.75275929])
```

```
X_poly[0]
```

```
array([-0.75275929,  0.56664654])
```

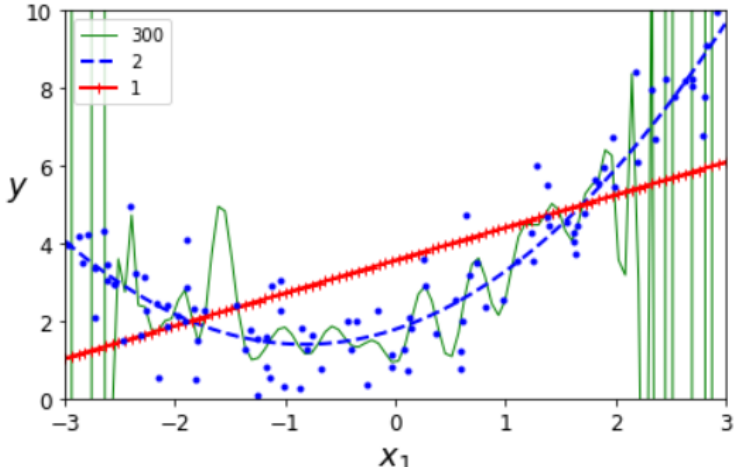
```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

```
X_new=np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([
            ("poly_features", polybig_features),
            ("std_scaler", std_scaler),
            ("lin_reg", lin_reg),
        ])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
save_fig("high_degree_polynomials_plot")
plt.show()
```

```
Saving figure high_degree_polynomials_plot
```

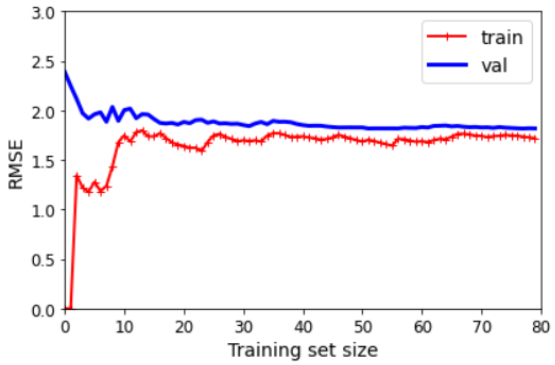


Learning Curves

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train) + 1):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))

    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
    plt.legend(loc="upper right", fontsize=14)   # not shown in the book
    plt.xlabel("Training set size", fontsize=14) # not shown
    plt.ylabel("RMSE", fontsize=14)              # not shown
```

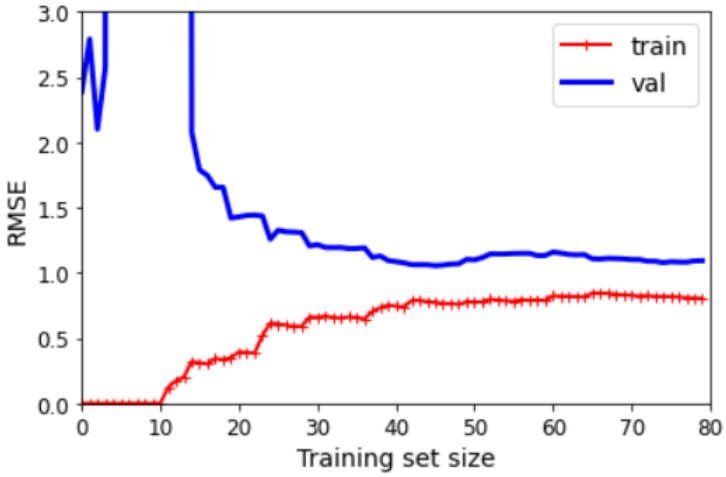
```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3])                          # not shown in the book
save_fig("underfitting_learning_curves_plot")    # not shown
plt.show()                                       # not shown
```

```
Saving figure underfitting_learning_curves_plot
```




```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ])

plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3])           # not shown
save_fig("learning_curves_plot")  # not shown
plt.show()                        # not shown
```

```
Saving figure learning_curves_plot
```



c. Regularized Linear Models (Ridge, Lasso, ElasticNet)

Implement regression variants like LASSO and Ridge on any generated dataset.

Regularised Linear Model

Ridge Regression

```
[ ]  np.random.seed(42)
     m = 20
     X = 3 * np.random.rand(m, 1)
     y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
     X_new = np.linspace(0, 3, 100).reshape(100, 1)
```

```
     from sklearn.linear_model import Ridge
     ridge_reg = Ridge(alpha=1, solver="cholesky", random_state=42)
     ridge_reg.fit(X, y)
     ridge_reg.predict([[1.5]])
```

```
array([[1.55071465]])
```

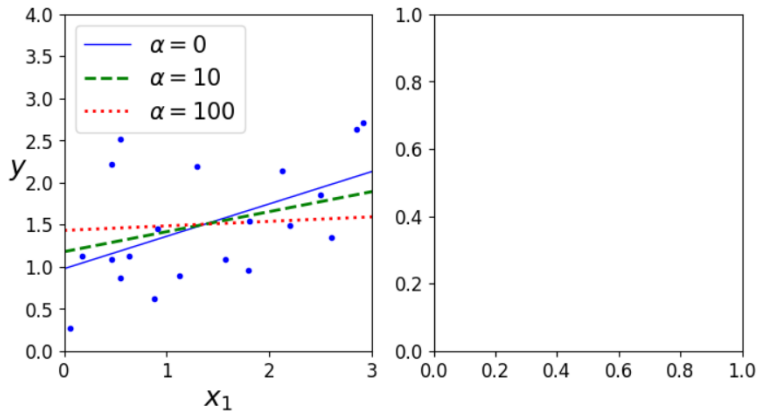
```
[ ]  ridge_reg = Ridge(alpha=1, solver="sag", random_state=42)
     ridge_reg.fit(X, y)
     ridge_reg.predict([[1.5]])
```

```
array([[1.55072189]])
```

```
from sklearn.linear_model import Ridge

def plot_model(model_class, polynomial, alphas, **model_kargs):
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline([
                    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ])
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
```

save_fig("ridge_regression_plot")

plt.show()



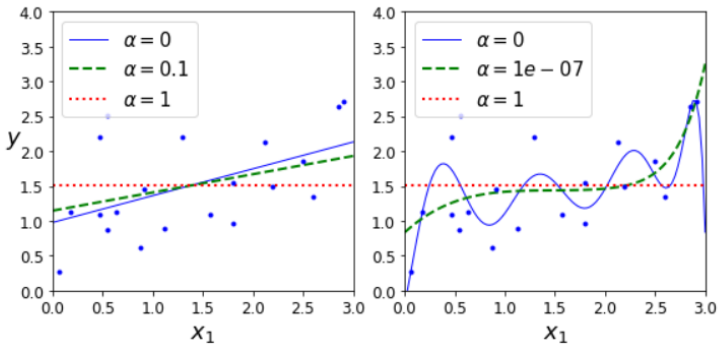

```
[ ] from sklearn.linear_model import SGDRegressor
    sgd_reg = SGDRegressor(penalty="l2", max_iter=1000, tol=1e-3, random_state=42)
    sgd_reg.fit(X, y.ravel())
    sgd_reg.predict([[1.5]])
```

```
array([1.47012588])
```

Lasso Regression

```
from sklearn.linear_model import Lasso

plt.figure(figsize=(8,4))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), random_state=42)

save_fig("lasso_regression_plot")
plt.show()
```

```
C:\Users\hina_\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:529: Conv
  model = cd_fast.enet_coordinate_descent(
Saving figure lasso_regression_plot
```



```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X, y)
lasso_reg.predict([[1.5]])
```

```
array([1.53788174])
```

Elastic Net

Elastic Net

```
[ ] from sklearn.linear_model import ElasticNet
    elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
    elastic_net.fit(X, y)
    elastic_net.predict([[1.5]])
```

```
array([1.54333232])
```

Practical 4

Aim: Discriminative Models

a. Logistic Regression Perform binary classification using logistic regression. Calculate accuracy, precision, recall, and understand the ROC curve.

Theory: Logistic Regression

We will be using the Human Activity Recognition with Smartphones database, which was built from the recordings of study participants who carried a smartphone with an embedded inertial sensor while performing activities of daily living (ADL). The objective is to classify the activities the participants performed into one of the six following categories: walking, walking upstairs, walking downstairs, sitting, standing, and laying.

- The following information is provided for each record in the dataset:
  - Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration
  - Triaxial Angular velocity from the gyroscope
  - A 561-feature vector with time and frequency domain variables
  - The activity label
- Import the data and do the following:
  - Examine the data types--there are many columns, so it might be wise to use value counts.
  - Determine if the floating-point values need to be scaled.
  - Determine the breakdown of each activity.
  - Encode the activity label as an integer.

data = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/Human_Activity_Recognition_Using_Smartphones_Data.csv", sep=',')
The data columns are all floats except for the activity label.

```
In [7]: data.dtypes.value_counts()
```

```
Out[7]: float64    561
        object       1
        dtype: int64
```

```
In [8]: data.dtypes.tail()
```

```
Out[8]: angle(tBodyGyroJerkMean,gravityMean)    float64
        angle(X,gravityMean)                    float64
        angle(Y,gravityMean)                    float64
        angle(Z,gravityMean)                    float64
        Activity                                 object
        dtype: object
```

The data are all scaled from -1 (minimum) to 1.0 (maximum).

```
In [9]: data.iloc[:, :-1].min().value_counts()
```

```
Out[9]: -1.0    561
        dtype: int64
```

```
In [10]: data.iloc[:, :-1].max().value_counts()
```

```
Out[10]: 1.0    561
         dtype: int64
```

Examine the breakdown of activities; they are relatively balanced.

```
In [11]: data.Activity.value_counts()
```

```
Out[11]: LAYING                1944
         STANDING              1906
         SITTING               1777
         WALKING               1722
         WALKING_UPSTAIRS      1544
         WALKING_DOWNSTAIRS    1406
         Name: Activity, dtype: int64
```

Scikit learn classifiers won't accept a sparse matrix for the prediction column. Thus, either LabelEncoder needs to be used to convert the activity labels to integers, or if DictVectorizer is used, the resulting matrix must be converted to a non-sparse array. Use LabelEncoder to fit_transform the "Activity" column, and look at 5 random values.

```
In [12]: from sklearn.preprocessing import LabelEncoder

         le = LabelEncoder()
         data['Activity'] = le.fit_transform(data.Activity)
         data['Activity'].sample(5)
         ### END SOLUTION
```

```
Out[12]: 3980     2
         8734     2
         1960     0
         10082    5
         6159     0
         Name: Activity, dtype: int32
```

- Split the data into train and test data sets. This can be done using any method, but consider using Scikit-learn's StratifiedShuffleSplit to maintain the same ratio of predictor classes.
- Regardless of the method used to split the data, compare the ratio of classes in both the train and test splits.

In [17]:
```
### BEGIN SOLUTION
from sklearn.model_selection import StratifiedShuffleSplit

# Get the split indexes
strat_shuf_split = StratifiedShuffleSplit(n_splits=1,
                                          test_size=0.3,
                                          random_state=42)

train_idx, test_idx = next(strat_shuf_split.split(data[feature_cols], data.Activity))

# Create the dataframes
X_train = data.loc[train_idx, feature_cols]
y_train = data.loc[train_idx, 'Activity']

X_test  = data.loc[test_idx, feature_cols]
y_test  = data.loc[test_idx, 'Activity']
```

In [18]:
```
y_train.value_counts(normalize=True)
```

Out[18]:
```
0    0.188792
2    0.185046
1    0.172562
3    0.167152
5    0.149951
4    0.136496
Name: Activity, dtype: float64
```

Out[19]:
```
0    0.188673
2    0.185113
1    0.172492
3    0.167314
5    0.149838
4    0.136570
Name: Activity, dtype: float64
```

Fit a logistic regression model without any regularization using all of the features. Be sure to read the documentation about fitting a multi-class model so you understand the coefficient output. Store the model.

Using cross validation to determine the hyperparameters and fit models using L1 and L2 regularization. Store each of these models as well. Note the limitations on multi-class models, solvers, and regularizations. The regularized models, in particular the L1 model, will probably take a while to fit.

In [20]:
```
### BEGIN SOLUTION
from sklearn.linear_model import LogisticRegression

# Standard logistic regression
lr = LogisticRegression(solver='liblinear').fit(X_train, y_train)
```

In [21]:
```
from sklearn.linear_model import LogisticRegressionCV

# L1 regularized logistic regression
lr_l1 = LogisticRegressionCV(Cs=10, cv=4, penalty='l1', solver='liblinear').fit(X_train, y_train)
```

In [22]:
```
# L2 regularized logistic regression
lr_l2 = LogisticRegressionCV(Cs=10, cv=4, penalty='l2', solver='liblinear').fit(X_train, y_train)
### END SOLUTION
```

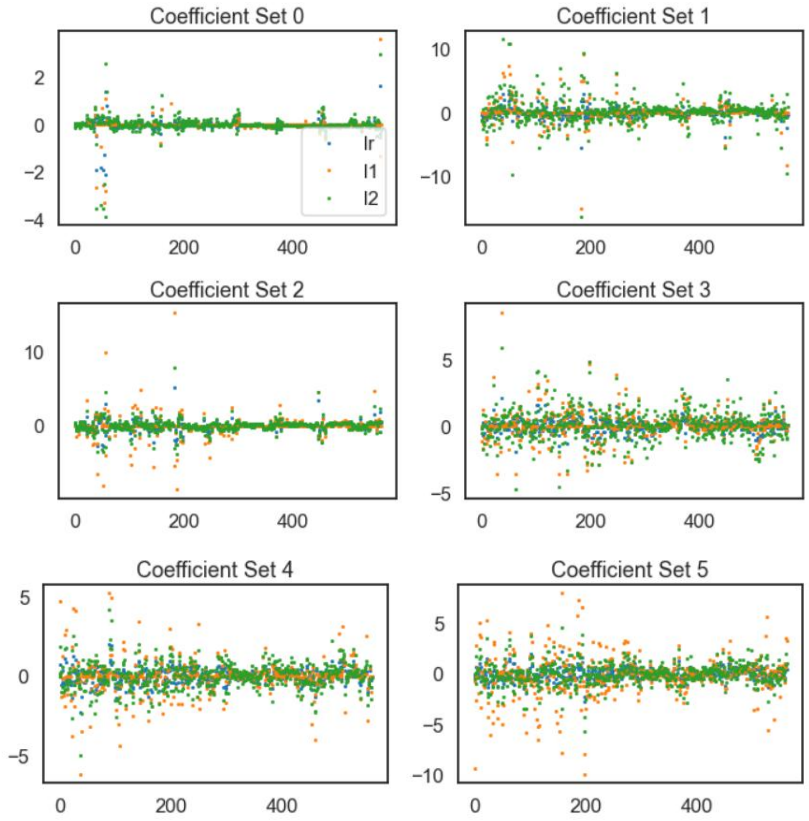
- Compare the magnitudes of the coefficients for each of the models. If one-vs-rest fitting was used, each set of coefficients can be plotted separately.

In [23]:
```
### BEGIN SOLUTION
# Combine all the coefficients into a dataframe
coefficients = list()

coeff_labels = ['lr', 'l1', 'l2']
coeff_models = [lr, lr_l1, lr_l2]

for lab,mod in zip(coeff_labels, coeff_models):
    coeffs = mod.coef_
    coeff_label = pd.MultiIndex(levels=[[lab], [0,1,2,3,4,5]],
                                codes=[[0,0,0,0,0,0], [0,1,2,3,4,5]])
    coefficients.append(pd.DataFrame(coeffs.T, columns=coeff_label))

coefficients = pd.concat(coefficients, axis=1)

coefficients.sample(10)
```

	lr						l1						l2		
	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2
504	0.039087	0.067478	-0.293047	-1.059035	0.955001	-0.424794	0.0	0.000000	-0.198053	-1.661399	1.703378	-1.591278	0.083564	0.981149	-0.525543
268	0.010562	-0.798042	-0.376238	-0.786235	0.288698	0.806855	0.0	-2.041827	-0.536102	-2.298778	0.020242	3.021548	0.030290	-2.838983	-0.469527
330	0.025223	-0.065299	0.180469	0.190374	-0.233112	0.119824	0.0	0.000000	0.000000	0.048453	0.000000	-0.035698	0.043901	0.058778	0.235508
2	0.030187	0.063196	0.245909	0.022339	0.058980	-0.402526	0.0	0.000000	0.000000	0.000000	0.000000	-2.615128	0.040581	-1.427489	0.464370
336	0.012760	0.211213	0.169117	-0.123426	0.175218	0.169581	0.0	0.000000	0.000000	-0.001496	0.111496	0.165215	0.002672	0.429630	0.179563
420	-0.005261	0.141068	0.178015	-0.158856	0.417940	-0.049662	0.0	0.000000	0.000000	-0.032385	0.465864	0.089012	-0.028151	0.296081	0.217800
231	-0.061735	-0.201630	-0.382630	0.267972	-0.175103	-0.176351	0.0	0.000000	0.000000	0.000000	0.000000	-0.858012	-0.103299	-0.510887	-0.536829
491	0.001267	0.239313	0.270985	0.037240	0.164332	0.572029	0.0	0.519335	0.582513	0.000431	0.299443	0.553175	-0.019911	0.572164	0.338196
22	0.087942	-0.747104	1.265813	1.232442	-1.393888	1.506603	0.0	-1.778920	3.417595	3.729152	-3.766175	5.249481	0.252127	-2.834183	2.023683
242	0.016123	-0.078789	0.061986	-0.731070	0.464251	-0.069518	0.0	0.000000	0.000000	-1.733254	1.457865	-0.628834	0.040247	-0.381804	0.075218

Prepare six separate plots for each of the multi-class coefficients.

In [24]:
```
fig, axList = plt.subplots(nrows=3, ncols=2)
axList = axList.flatten()
fig.set_size_inches(10,10)

for ax in enumerate(axList):
    loc = ax[0]
    ax = ax[1]

    data = coefficients.xs(loc, level=1, axis=1)
    data.plot(marker='o', ls='', ms=2.0, ax=ax, legend=False)

    if ax is axList[0]:
        ax.legend(loc=4)

    ax.set(title='Coefficient Set '+str(loc))

plt.tight_layout()
### END SOLUTION
```



- Predict and store the class for each model.
- Store the probability for the predicted class for each model.

In [25]:
```
### BEGIN SOLUTION
# Predict the class and the probability for each
y_pred = list()
y_prob = list()

coeff_labels = ['lr', 'l1', 'l2']
coeff_models = [lr, lr_l1, lr_l2]

for lab,mod in zip(coeff_labels, coeff_models):
    y_pred.append(pd.Series(mod.predict(X_test), name=lab))
    y_prob.append(pd.Series(mod.predict_proba(X_test).max(axis=1), name=lab))

y_pred = pd.concat(y_pred, axis=1)
y_prob = pd.concat(y_prob, axis=1)

y_pred.head()
```

Out[25]:

	lr	l1	l2
0	3	3	3
1	5	5	5
2	3	3	3
3	1	1	1
4	0	0	0

In [26]:
```
y_prob.head()
### END SOLUTION
```

Out[26]:

	lr	l1	l2
0	0.998939	0.998902	0.999757
1	0.988165	0.999431	0.999477
2	0.987592	0.995224	0.999670
3	0.981381	0.999162	0.994338
4	0.998277	0.999920	0.999997

For each model, calculate the following error metrics:

- Accuracy
- Precision
- Recall
- F-score
- Confusion Matrix

Decide how to combine the multi-class metrics into a single value for each model.

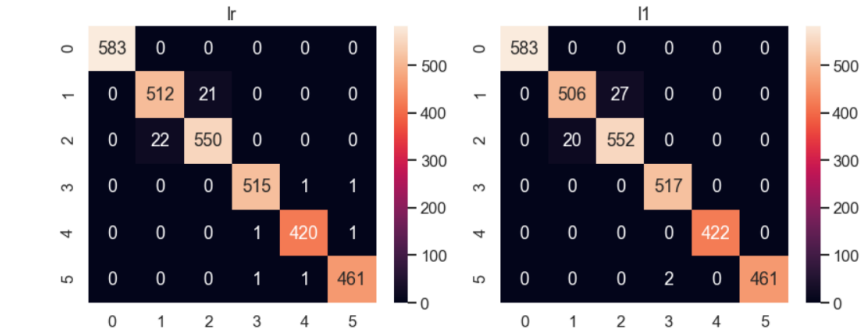
In [27]:
```
### BEGIN SOLUTION
from sklearn.metrics import precision_recall_fscore_support as score
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score
from sklearn.preprocessing import label_binarize

metrics = list()
cm = dict()

for lab in coeff_labels:

    # Preciision, recall, f-score from the multi-class support function
    precision, recall, fscore, _ = score(y_test, y_pred[lab], average='weighted')

    # The usual way to calculate accuracy
    accuracy = accuracy_score(y_test, y_pred[lab])

    # ROC-AUC scores can be calculated by binarizing the data
    auc = roc_auc_score(label_binarize(y_test, classes=[0,1,2,3,4,5]),
              label_binarize(y_pred[lab], classes=[0,1,2,3,4,5]),
              average='weighted')

    # Last, the confusion matrix
    cm[lab] = confusion_matrix(y_test, y_pred[lab])

    metrics.append(pd.Series({'precision':precision, 'recall':recall,
                              'fscore':fscore, 'accuracy':accuracy,
                              'auc':auc},
                             name=lab))

metrics = pd.concat(metrics, axis=1)
```

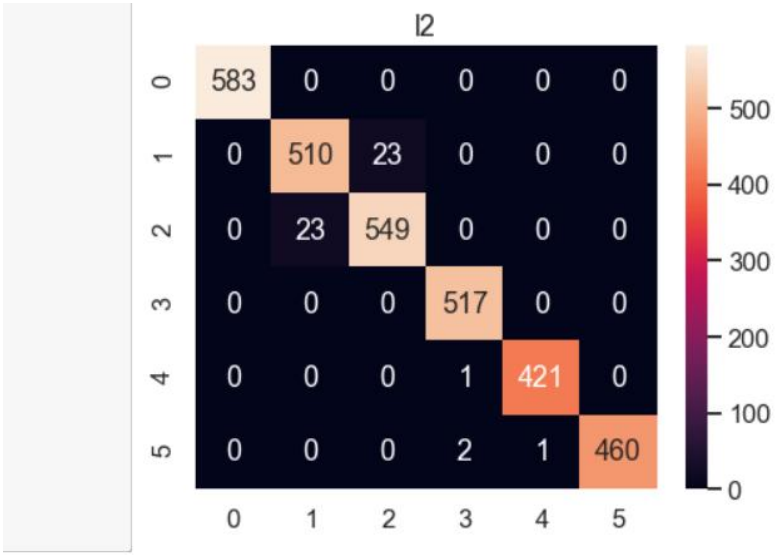
In [28]:
```
metrics
### END SOLUTION
```

	lr	l1	l2
precision	0.984144	0.984164	0.983824
recall	0.984142	0.984142	0.983819
fscore	0.984143	0.984138	0.983819
accuracy	0.984142	0.984142	0.983819
auc	0.990384	0.990339	0.990165

- Display or plot the confusion matrix for each model.

In [30]:
```
### BEGIN SOLUTION
fig, axList = plt.subplots(nrows=2, ncols=2)
axList = axList.flatten()
fig.set_size_inches(12, 10)

axList[-1].axis('off')

for ax,lab in zip(axList[:-1], coeff_labels):
    sns.heatmap(cm[lab], ax=ax, annot=True, fmt='d');
    ax.set(title=lab);

plt.tight_layout()
### END SOLUTION
```





b. Implement and demonstrate k-nearest Neighbour algorithm. Read the training data from a .CSV file and build the model to classify a test sample. Print both correct and wrong predictions.

Classification -KNN Algorithm

In this lab, you will learn about and practice the K Nearest Neighbor (KNN) model. KNN is a straightforward but very effective model that can be used for both classification and regression tasks. If the feature space is not very large, KNN can be a high-interpretable model because you can explain and understand how a prediction is made by looking at its nearest neighbors.

We will be using a tumor sample dataset containing lab test results about tumor samples. The objective is to classify whether a tumor is malicious (cancer) or benign. As such, it is a typical binary classification task.

After completing this lab, you will be able to:

Train KNN models with different neighbor hyper-parameters

Evaluate KNN models on classification tasks

Tune the number of neighbors and find the optimized one for a specific task

First, let's install seaborn for visualization tasks and import required libraries for this lab.

```
import pandas as pd

import numpy as np

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split

from sklearn import metrics

# Evaluation metrics related methods

from sklearn.metrics import classification_report, accuracy_score, f1_score, confusion_matrix, precision_recall_fscore_support, precision_score, recall_score

import matplotlib.pyplot as plt

import seaborn as sns

%matplotlib inline
```

Define a random seed to reproduce any random process

rs = 123

Load and explore the tumor sample dataset

We first load the dataset tumor.csv as a Pandas dataframe:

Read datast in csv format

dataset_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/tumor.csv"

tumor_df = pd.read_csv(dataset_url)

Then, let's quickly take a look at the head of the dataframe.

```
In [11]: tumor_df.head()
```

Out[11]:

	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
0	5	1	1	1	2	1	3	1	1	0
1	5	4	4	5	7	10	3	2	1	0
2	3	1	1	1	2	2	3	1	1	0
3	6	8	8	1	3	4	3	7	1	0
4	4	1	1	3	2	1	3	1	1	0

And, display its columns.

```
In [12]: tumor_df.columns
```

```
Out[12]: Index(['Clump', 'UnifSize', 'UnifShape', 'MargAdh', 'SingEpiSize', 'BareNuc',
       'BlandChrom', 'NormNucl', 'Mit', 'Class'],
      dtype='object')
```

Each observation in this dataset contains lab test results about a tumor sample, such as clump or shapes. Based on these lab test results or features, we want to build a classification model to predict if this tumor sample is malicious (cancer) or benign. The target variable `y` is specified in the `Class` column.

Then, let's split the dataset into input `X` and output `y`:

```
In [13]: X = tumor_df.iloc[:, :-1]
         y = tumor_df.iloc[:, -1:]
```

And, we first check the statistics summary of features in `X`:

```
In [14]: X.describe()
```

Out[14]:

	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit
count	683.000000	683.000000	683.000000	683.000000	683.000000	683.000000	683.000000	683.000000	683.000000
mean	4.442167	3.150805	3.215227	2.830161	3.234261	3.544656	3.445095	2.869693	1.603221
std	2.820761	3.065145	2.988581	2.864562	2.223085	3.643857	2.449697	3.052666	1.732674
min	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
25%	2.000000	1.000000	1.000000	1.000000	2.000000	1.000000	2.000000	1.000000	1.000000
50%	4.000000	1.000000	1.000000	1.000000	2.000000	1.000000	3.000000	1.000000	1.000000
75%	6.000000	5.000000	5.000000	4.000000	4.000000	6.000000	5.000000	4.000000	1.000000
max	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000

As we can see from the above cell output, all features are numeric and ranged between 1 to 10. This is very convenient as we do not need to scale the feature values as they are already in the same range.

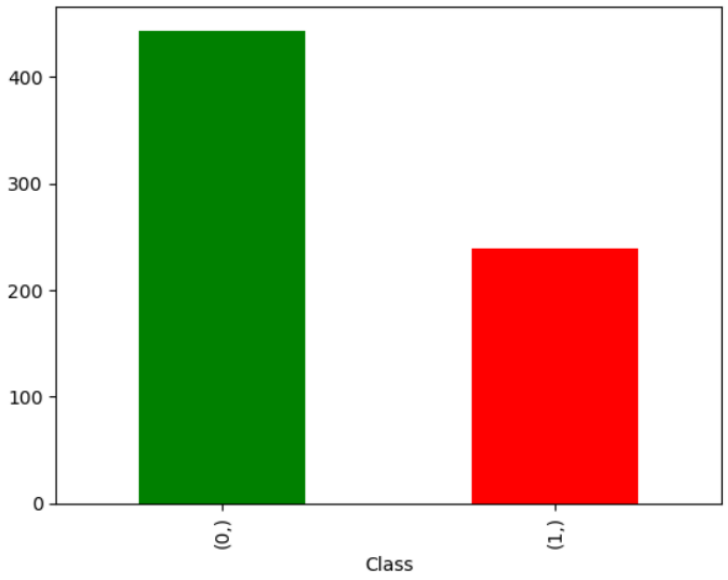
Next, let's check the class distribution of output `y`:

```
In [15]: y.value_counts(normalize=True)
```

```
Out[15]: Class
         0        0.650073
         1        0.349927
         dtype: float64
```

```
In [16]: y.value_counts().plot.bar(color=['green', 'red'])
```

```
Out[16]: <AxesSubplot:xlabel='Class'>
```



We have about 65% benign tumors (`Class = 0`) and 35% cancerous tumors (`Class = 1`), which is not a very imbalanced class distribution.

Split training and testing datasets

```
7]: # Split 80% as training dataset
    # and 20% as testing dataset
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state = rs)
```

Train and evaluate a KNN classifier with the number of neighbors set to 2

Training a KNN classifier is very similar to training other classifiers in `sklearn` , we first need to define a `KNeighborsClassifier` object. Here we use `n_neighbors=2` argument to specify how many neighbors will be used for prediction, and we keep other arguments to be their default values.

```
[18]: # Define a KNN classifier with `n_neighbors=2`
      knn_model = KNeighborsClassifier(n_neighbors=2)
```

Then we can train the model with `X_train` and `y_train` , and we use ravel() method to convert the data frame `y_train` to a vector.

```
[19]: knn_model.fit(X_train, y_train.values.ravel())
```

```
[19]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                          metric_params=None, n_jobs=None, n_neighbors=2, p=2,
                          weights='uniform')
```

And, we can make predictions on the `X_test` dataframe.

```
In [20]: preds = knn_model.predict(X_test)
```

To evaluate the KNN classifier, we provide a pre-defined method to return the commonly used evaluation metrics such as accuracy, recall, precision, f1score, and so on, based on the true classes in the 'y_test' and model predictions.

```
In [21]: def evaluate_metrics(yt, yp):
             results_pos = {}
             results_pos['accuracy'] = accuracy_score(yt, yp)
             precision, recall, f_beta, _ = precision_recall_fscore_support(yt, yp, average='binary')
             results_pos['recall'] = recall
             results_pos['precision'] = precision
             results_pos['f1score'] = f_beta
             return results_pos
```

```
In [22]: evaluate_metrics(y_test, preds)
```

```
Out[22]: {'accuracy': 0.9416058394160584,
          'recall': 0.875,
          'precision': 0.9545454545454546,
          'f1score': 0.9130434782608695}
```

We can see that there is a great classification performance on the tumor sample dataset. This means the KNN model can effectively recognize cancerous tumors. Next, it's your turn to try a different number of neighbors to see if we could get even better performance.

Tune the number of neighbors to find the optmized one

OK, you may wonder which `n_neighbors` argument may give you the best classification performance. We can try different `n_neighbors` (the K value) and check which `K` gives the best classification performance.

Here we could try K from 1 to 50, and store the aggregated `f1score` for each k into a list.

In [25]:
```
# Try K from 1 to 50
max_k = 50
# Create an empty list to store f1score for each k
f1_scores = []
```

Then we will train 50 KNN classifiers with K ranged from 1 to 50.

In [26]:
```
for k in range(1, max_k + 1):
    # Create a KNN classifier
    knn = KNeighborsClassifier(n_neighbors=k)
    # Train the classifier
    knn = knn.fit(X_train, y_train.values.ravel())
    preds = knn.predict(X_test)
    # Evaluate the classifier with f1score
    f1 = f1_score(preds, y_test)
    f1_scores.append((k, round(f1_score(y_test, preds), 4)))
# Convert the f1score list to a dataframe
f1_results = pd.DataFrame(f1_scores, columns=['K', 'F1 Score'])
f1_results.set_index('K')
```

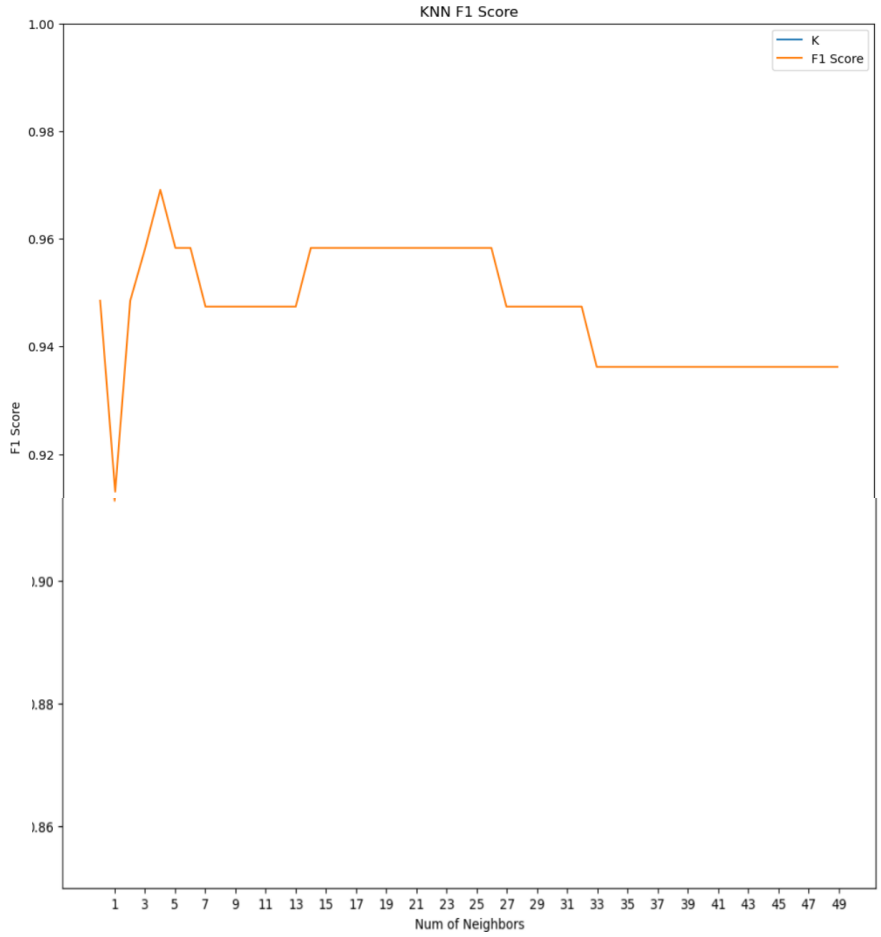
Out[26]:

K	F1 Score
1	0.9485
2	0.9130
3	0.9485
4	0.9583
5	0.9691
6	0.9583
7	0.9583
8	0.9474
9	0.9474
10	0.9474
11	0.9474

This is a long list and different to analysis, so let's visualize the list using a linechart.

In [27]:
```
# Plot F1 results
ax = f1_results.plot(figsize=(12, 12))
ax.set(xlabel='Num of Neighbors', ylabel='F1 Score')
ax.set_xticks(range(1, max_k, 2));
plt.ylim((0.85, 1))
plt.title('KNN F1 Score')
```

Out[27]:
```
Text(0.5, 1.0, 'KNN F1 Score')
```



c. Build a decision tree classifier or regressor. Control hyperparameters like tree depth to avoid overfitting. Visualize the tree.

Theory:

We will be using a tumor sample dataset, which contains lab test results about tumor samples. The objective is to classify whether a tumor is malicious (cancer) or benign. As such, it is a typical binary classification task.

After completing this lab, you will be able to:

- Train decision tree models with customized hyperparameters
- Evaluate decision tree models on classification tasks
- Visualize decision tree models by plotting the tree
- Tune the hyperparameters to find the optimized one for a specific task

```
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn import metrics
# Evaluation metrics related methods
from sklearn.metrics import classification_report, accuracy_score, f1_score, confusion_matrix, precision_recall_fscore_support, precision_score, recall_score

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

```
dataset_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/tumor.csv"
tumor_df = pd.read_csv(dataset_url)
```

And check its dataframe head

```
tumor_df.head()
```

	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
0	5	1	1	1	2	1	3	1	1	0
1	5	4	4	5	7	10	3	2	1	0
2	3	1	1	1	2	2	3	1	1	0
3	6	8	8	1	3	4	3	7	1	0
4	4	1	1	3	2	1	3	1	1	0

Each observation in this dataset contains lab tests results about a tumor sample, such as clump or shapes. Based on these lab test results or features, we want to build a classification model to predict if this tumor sample is malicious (cancer) and benign. The target variable `y` is specified in the `Class` column.

Then, let's split the dataframe into train and testing data

```
# Get the input features
X = tumor_df.iloc[:, :-1]
# Get the target variable
y = tumor_df.iloc[:, -1:]
```

```
# Split the training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state = rs)
```

Train a default decision tree

Training a decision classifier is very straightforward with `sklearn`, we first need to define a `DecisionTreeClassifier` will use all the default arguments.

```
[ ]  # Train a decision tree with all default arguments
     model = DecisionTreeClassifier(random_state=rs)
```

Then we can train the decision tree model with training and testing data

```
[ ]  model.fit(X_train, y_train.values.ravel())
```

```
DecisionTreeClassifier(random state=123)
```

And make predictions on the test data

```
[ ]  preds = model.predict(X_test)
```

Here we also provided a utility method to evaluate the trained decision tree model and output some standard evaluation metrics.

```
def evaluate_metrics(yt, yp):
    results_pos = {}
    results_pos['accuracy'] = accuracy_score(yt, yp)
    precision, recall, f_beta, _ = precision_recall_fscore_support(yt, yp, average='binary')
    results_pos['recall'] = recall
    results_pos['precision'] = precision
    results_pos['f1score'] = f_beta
    return results_pos
```

```
[ ]  evaluate_metrics(y_test, preds)
```

```
{'accuracy': 0.9562043795620438,
 'recall': 0.9583333333333334,
 'precision': 0.92,
 'f1score': 0.9387755102040817}
```

Now we can see that the trained decision model has very good classification results on the testing data, with a very high F1 score around 0.94. Next, let's try to visualize and interpret the trained decision tree model.

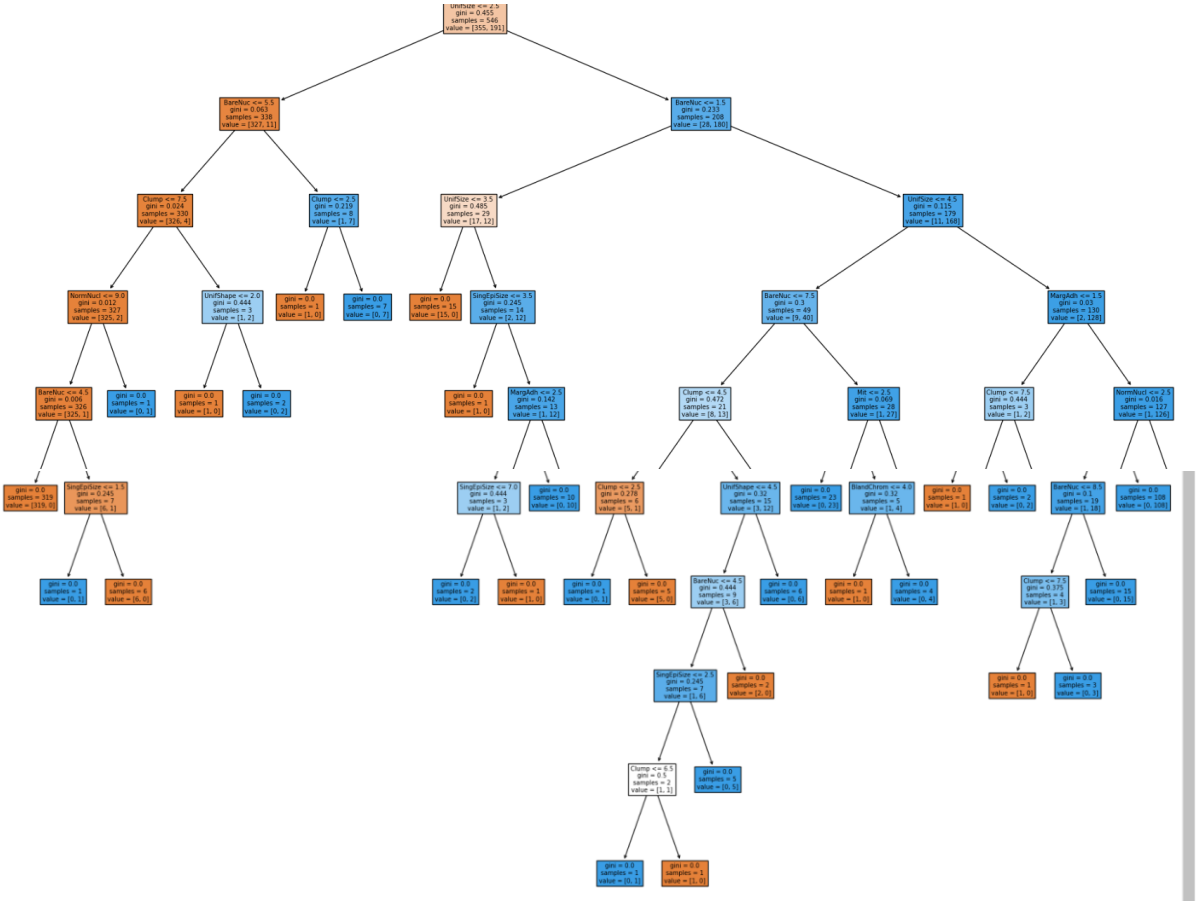
Visualize the trained decision tree

We will be using the `tree.plot_tree()` method provided by `sklearn` to quickly plot any decision tree model.

```
def plot_decision_tree(model, feature_names):
    plt.subplots(figsize=(25, 20))
    tree.plot_tree(model,
                   feature_names=feature_names,
                   filled=True)
    plt.show()
```

```
[ ]  feature_names = X.columns.values
```

```
[ ]  plot_decision_tree(model, feature_names)
```



And you should see a relatively complex decision tree model being plotted. First, you may notice the decision tree is color-labeled, orange node means a majority of samples in the node belong to `Class 0` and blue node means a majority of samples in the node belong to `Class 1`, and white node means it has an equal amount of `Class 0` and `Class 1` samples.

Because the tree is very big, so the rules and split threshold on each node are very difficult to see. In addition, big decision trees may easily bring large variance and cause overfitting. Next, let's try to build simplified decision trees, and hopefully the simplified decision trees may generate even better results.

Customize the decision tree model

The `DecisionTreeClassifier` has many arguments (model hyperparameters) that can be customized and eventually tune the generated decision tree classifiers. Among these arguments, there are three commonly tuned arguments as follows:

- criterion: `gini` or `entropy`, which specifies which criteria to be used when splitting a tree node
- max_depth: a numeric value to specify the max depth of the tree. Larger tree depth normally means larger model complexity
- min_samples_leaf: The minimal number of samples in leaf nodes. Larger samples in leaf nodes will tend to generate simpler trees

Let's first try the following hyperparameter values:

- criterion = 'entropy'
- max_depth = 10
- min_samples_leaf=3

```
[ ] # criterion = 'entropy'
    # max_depth = 10
    # min_samples_leaf=3
    custom_model = DecisionTreeClassifier(criterion='entropy', max_depth=10, min_samples_leaf=3, random_state=rs)
```

And let's train and evaluate the customized model

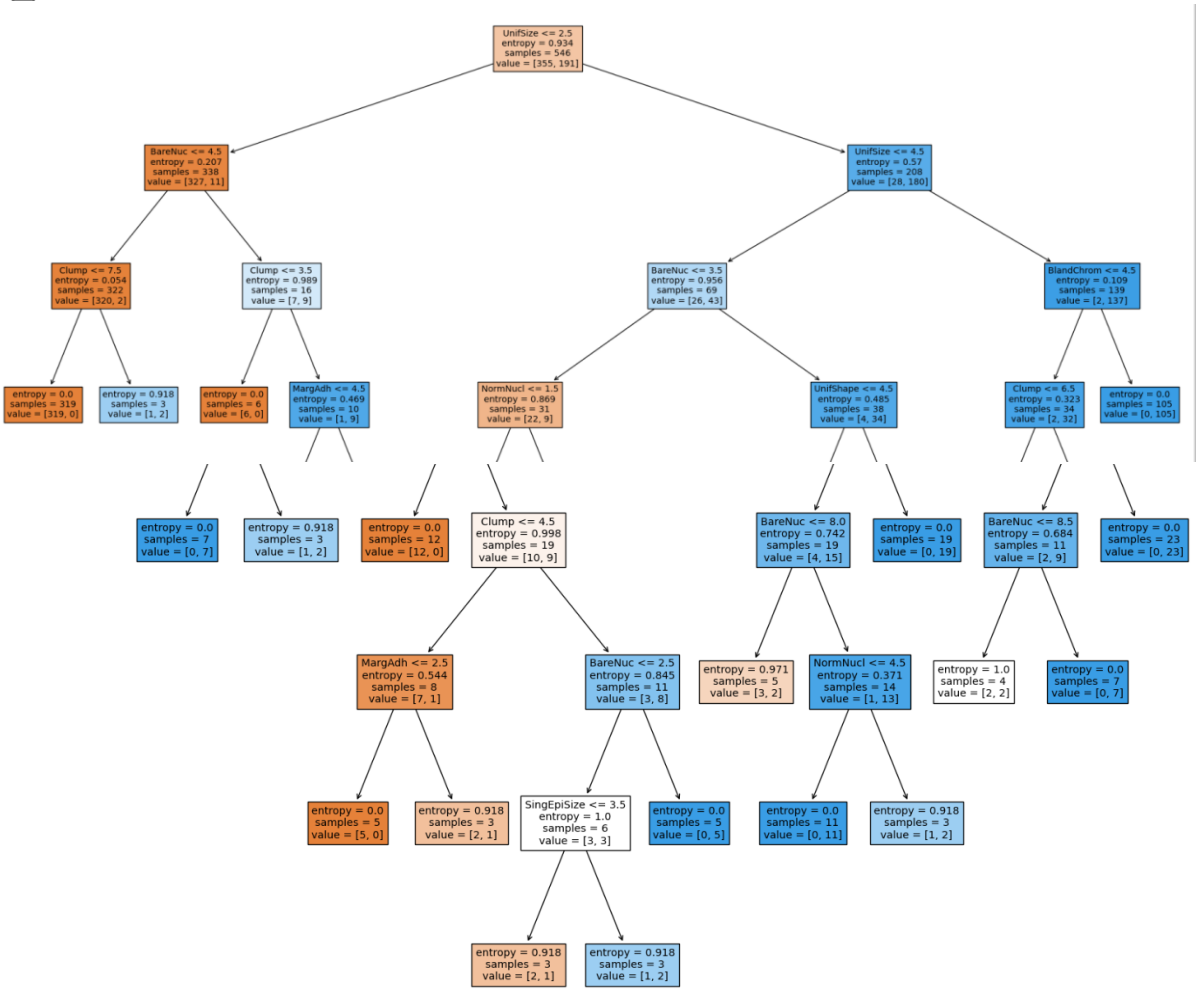
```
custom_model.fit(X_train, y_train.values.ravel())
preds = custom_model.predict(X_test)
evaluate_metrics(y_test, preds)
```

```
{'accuracy': 0.9635036496350365,
 'recall': 0.9166666666666666,
 'precision': 0.9777777777777777,
 'f1score': 0.946236559139785}
```

Its F1 score has increased to 0.946 now, which seems better than the previous default decision tree model.

Then, let's visualize the custom model using plot_decision_tree() utility method we created in the previous step:

```
[ ] # Plot the decision tree
    plot_decision_tree(custom_model, feature_names)
```



d. Implement a Support Vector Machine for any relevant dataset.

Theory:

We will be using the wine quality data set for these exercises. This data set contains various chemical properties of wine, such as acidity, sugar, pH, and alcohol. It also contains a quality metric (3-9, with highest being better) and a color (red or white). The name of the file is Wine_Quality_Data.csv.

In [1]:
```
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn

import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sns
```

Part 1: Setup

- Import the data.
- Create the target variable `y` as a 1/0 column where 1 means red.
- Create a `pairplot` for the dataset.
- Create a bar plot showing the correlations between each column and `y`
- Pick the most 2 correlated fields (using the absolute value of correlations) and create `X`
- Use MinMaxScaler to scale `X`. Note that this will output a np.array. Make it a DataFrame again and rename the columns appropriately.

In [2]:
```
data = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/Wi
```

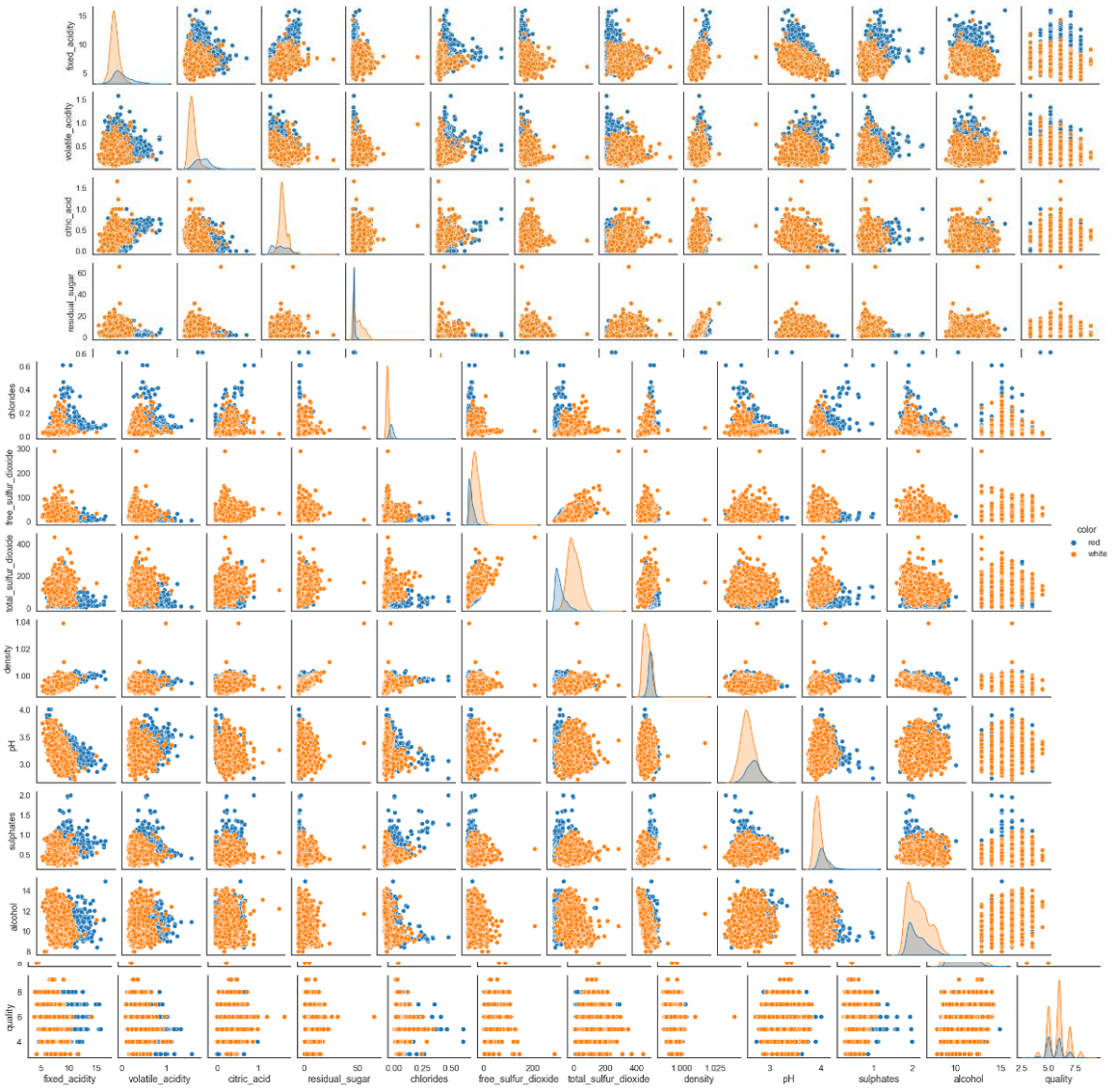
In [3]:
```
y = (data['color'] == 'red').astype(int)
fields = list(data.columns[:-1])  # everything except "color"
correlations = data[fields].corrwith(y)
correlations.sort_values(inplace=True)
correlations
```

Out[3]:
```
total_sulfur_dioxide   -0.700357
free_sulfur_dioxide    -0.471644
residual_sugar         -0.348821
citric_acid            -0.187397
quality                -0.119323
alcohol                -0.032970
pH                      0.329129
density                 0.390645
fixed_acidity           0.486740
sulphates               0.487218
chlorides               0.512678
volatile_acidity        0.653036
dtype: float64
```

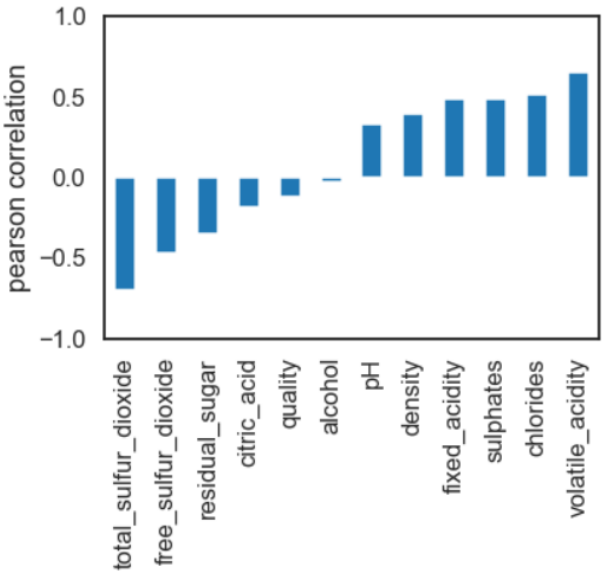
In [4]:
```
sns.set_context('talk')
#sns.set_palette(palette)
sns.set_style('white')
```

In [5]:
```
sns.pairplot(data, hue='color')
```

Out[5]: <seaborn.axisgrid.PairGrid at 0x19d9f4f5250>



In [6]:
```
ax = correlations.plot(kind='bar')
ax.set(ylim=[-1, 1], ylabel='pearson correlation');
```



In [7]:
```
from sklearn.preprocessing import MinMaxScaler

fields = correlations.map(abs).sort_values().iloc[-2:].index
print(fields)
X = data[fields]
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
X = pd.DataFrame(X, columns=['%s_scaled' % fld for fld in fields])
print(X.columns)
```

```
Index(['volatile_acidity', 'total_sulfur_dioxide'], dtype='object')
Index(['volatile_acidity_scaled', 'total_sulfur_dioxide_scaled'], dtype='object')
```

Part 2: Linear Decision Boundary

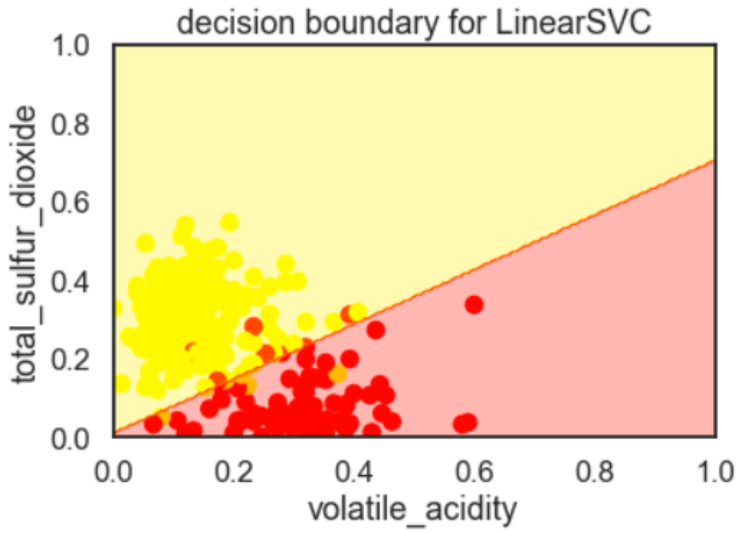
Our goal is to look at the decision boundary of a LinearSVC classifier on this dataset. Check out this example in sklearn's documentation.

- Fit a Linear Support Vector Machine Classifier to `X`, `y`.
- Pick 300 samples from `X`. Get the corresponding `y` value. Store them in variables `X_color` and `y_color`. This is because original dataset is too large and it produces a crowded plot.
- Modify `y_color` so that it has the value "red" instead of 1 and 'yellow' instead of 0.
- Scatter plot X_color's columns. Use the keyword argument "color=y_color" to color code samples.
- Use the code snippet below to plot the decision surface in a color coded way.

```
x_axis, y_axis = np.arange(0, 1, .005), np.arange(0, 1, .005)
xx, yy = np.meshgrid(x_axis, y_axis)
xx_ravel = xx.ravel()
yy_ravel = yy.ravel()
X_grid = pd.DataFrame([xx_ravel, yy_ravel]).T
y_grid_predictions = *[YOUR MODEL]*.predict(X_grid)
y_grid_predictions = y_grid_predictions.reshape(xx.shape)
ax.contourf(xx, yy, y_grid_predictions, cmap=colors, alpha=.3)
```

With LinearSVC, it is easy to experiment with different parameter choices and see the decision boundary.

In [8]:
```
from sklearn.svm import LinearSVC

LSVC = LinearSVC()
LSVC.fit(X, y)

X_color = X.sample(300, random_state=45)
y_color = y.loc[X_color.index]
y_color = y_color.map(lambda r: 'red' if r == 1 else 'yellow')
ax = plt.axes()
ax.scatter(
    X_color.iloc[:, 0], X_color.iloc[:, 1],
    color=y_color, alpha=1)
# -----------
x_axis, y_axis = np.arange(0, 1.005, .005), np.arange(0, 1.005, .005)
xx, yy = np.meshgrid(x_axis, y_axis)
xx_ravel = xx.ravel()
yy_ravel = yy.ravel()
X_grid = pd.DataFrame([xx_ravel, yy_ravel]).T
y_grid_predictions = LSVC.predict(X_grid)
y_grid_predictions = y_grid_predictions.reshape(xx.shape)
ax.contourf(xx, yy, y_grid_predictions, cmap=plt.cm.autumn_r, alpha=.3)
# -----------
ax.set(
    xlabel=fields[0],
    ylabel=fields[1],
    xlim=[0, 1],
    ylim=[0, 1],
    title='decision boundary for LinearSVC');
```



e. Train a random forest ensemble. Experiment with the number of trees and feature sampling. Compare performance to a single decision tree.

Theory:

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

```
[ ] path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/DA0101EN/automobileEDA.csv'
    df = pd.read_csv(path)
    df.head()
```

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km	horsepower-binned
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	13495.0	11.190476	Medium
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	16500.0	11.190476	Medium
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	9.0	154.0	5000.0	19	26	16500.0	12.368421	Medium
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	10.0	102.0	5500.0	24	30	13950.0	9.791667	Medium
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	8.0	115.0	5500.0	18	22	17450.0	13.055556	Medium

5 rows × 29 columns

```
df.dtypes
```

```
symboling              int64
normalized-losses      int64
make                  object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower           float64
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
city-L/100km         float64
horsepower-binned     object
diesel                 int64
```

```
df.describe()
```

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	bore	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg
count	201.000000	201.00000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	201.000000	197.000000	201.000000	201.000000	201.000000	201.000000	201.000000
mean	0.840796	122.00000	98.797015	0.837102	0.915126	53.766667	2555.666667	126.875622	3.330692	3.256904	10.164279	103.405534	5117.665368	25.179104	30.686567
std	1.254802	31.99625	6.066366	0.059213	0.029187	2.447822	517.296727	41.546834	0.268072	0.319256	4.004965	37.365700	478.113805	6.423220	6.815150
min	-2.000000	65.00000	86.600000	0.678039	0.837500	47.800000	1488.000000	61.000000	2.540000	2.070000	7.000000	48.000000	4150.000000	13.000000	16.000000
25%	0.000000	101.00000	94.500000	0.801538	0.890278	52.000000	2169.000000	98.000000	3.150000	3.110000	8.600000	70.000000	4800.000000	19.000000	25.000000
50%	1.000000	122.00000	97.000000	0.832292	0.909722	54.100000	2414.000000	120.000000	3.310000	3.290000	9.000000	95.000000	5125.369458	24.000000	30.000000
75%	2.000000	137.00000	102.400000	0.881788	0.925000	55.500000	2926.000000	141.000000	3.580000	3.410000	9.400000	116.000000	5500.000000	30.000000	34.000000
max	3.000000	256.00000	120.900000	1.000000	1.000000	59.800000	4066.000000	326.000000	3.940000	4.170000	23.000000	262.000000	6600.000000	49.000000	54.000000

```
[ ] df.dtypes
    for x in df:
        if df[x].dtypes == "int64":
            df[x] = df[x].astype(float)
            print (df[x].dtypes)
```

```
float64
float64
float64
float64
float64
float64
float64
float64
```

```
[ ] df = df.select_dtypes(exclude=['object'])
    df=df.fillna(df.mean())
    X = df.drop('price',axis=1)
    y = df['price']
```

```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

```
[ ] from sklearn.ensemble import RandomForestRegressor
    regressor = RandomForestRegressor(n_estimators = 1000, random_state = 42)
    regressor.fit(X_train, y_train)
```

```
RandomForestRegressor(n_estimators=1000, random_state=42)
```

```
y_pred = regressor.predict(X_test)
df=pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})
df
```

	Actual	Predicted
18	6295.0	5828.668000
170	10698.0	10492.447583
107	13860.0	21763.165000
98	13499.0	15317.683000
177	15750.0	16334.467000
...	...	...
30	6855.0	6115.846000
160	8238.0	8129.287000
40	12945.0	11438.811000
56	8845.0	10125.190667
131	15510.0	13600.018833

61 rows × 2 columns

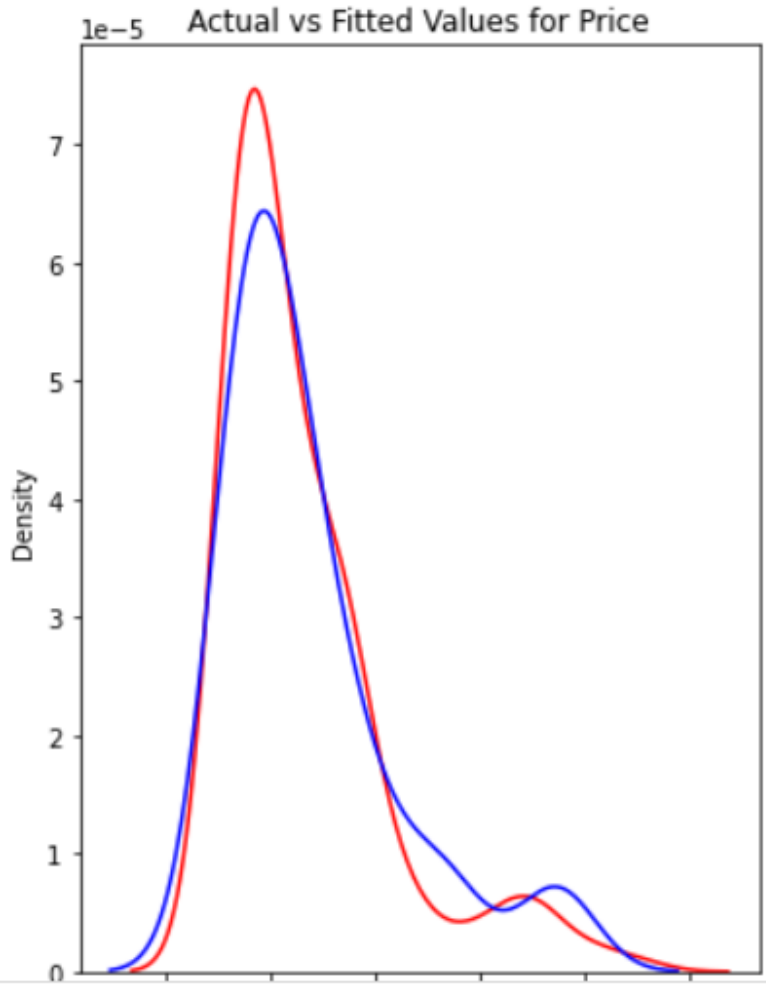
```
[ ] from sklearn import metrics
    print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
    print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
    print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

```
Mean Absolute Error: 1993.2901175839186
Mean Squared Error: 9668487.223350348
Root Mean Squared Error: 3109.4191134921566
```

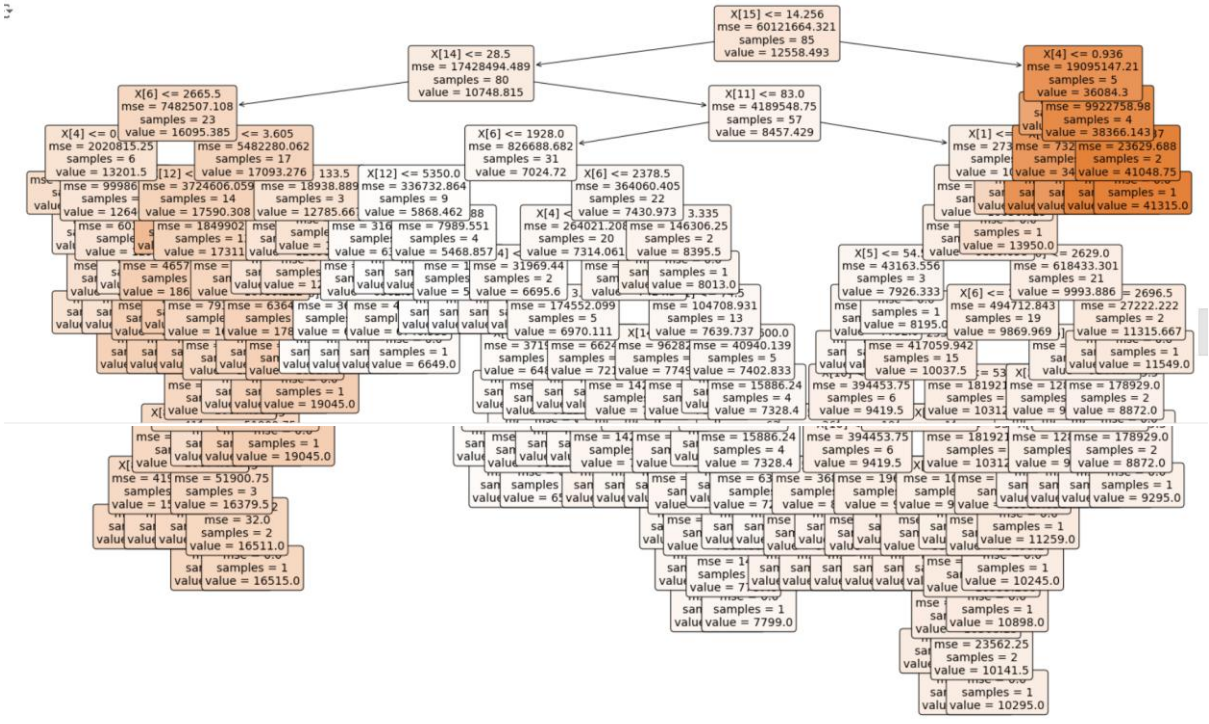
```
# Calculate the absolute errors
errors = abs(y_pred - y_test)
# Print out the mean absolute error (mae)
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')

# Calculate mean absolute percentage error (MAPE)
mape = 100 * (errors / y_test)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')
```

```
Mean Absolute Error: 1993.29 degrees.
Accuracy: 87.87 %.
```

```
import seaborn as sns
plt.figure(figsize=(5, 7))


ax = sns.distplot(y, hist=False, color="r", label="Actual Value")
sns.distplot(y_pred, hist=False, color="b", label="Fitted Values" , ax=ax)


plt.title('Actual vs Fitted Values for Price')


plt.show()
plt.close()
```



```
# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
# Pull out one tree from the forest
Tree = regressor.estimators_[5]
# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
# Pull out one tree from the forest
Tree = regressor.estimators_[5]
# Export the image to a dot file
from sklearn import tree
plt.figure(figsize=(25,15))
tree.plot_tree(Tree,filled=True,
               rounded=True,
               fontsize=14);
```



f. Implement a gradient boosting machine (e.g., XGBoost). Tune hyperparameters and explore feature importance.

Theory:

Adaptive Boosting (AdaBoost) for classification with Python

After completing this lab you will be able to:

Understand that AdaBoost is a linear combination of T weak classifiers

Apply AdaBoost.

Understand Hyperparameters selection in AdaBoost.

In this notebook, you will learn AdaBoost, short for Adaptive Boosting, is a classification algorithm; AdaBoost is actually part of a family of Boosting algorithms. Like Bagging and Random Forest (RF), AdaBoost combines the outputs of many classifiers into an ensemble, but there are some differences. In both Bagging and RF, each classifier in the ensemble is powerful but prone to overfitting. As Bagging or RF aggregate more and more classifiers, they reduce overfitting.

With AdaBoost, each Classifier usually has performance slightly better than random. This is referred to as a weak learner or weak classifier. AdaBoost combines these classifiers to get a strong classifier. Unlike Bagging and Random Forest, in AdaBoost, adding more learners can cause overfitting. As a result, AdaBoost requires Hyperparameter tuning, taking more time to train. One advantage of AdaBoost is that each classifier is smaller, so predictions are faster.

In AdaBoost, the strong classifier H(x) is a linear combination of T weak classifiers ht(x) and αt as shown in (1). Although each classifier ht(x) appears independent, the αt contains information about the error of classifiers from h1(x),..,ht−1(x). As we add more classifiers, the training accuracy gets larger. What's not so apparent in (1) is that during the training process, the values of that training sample are modified for ht(x). For a more in depth look at the theory behind Adaboost, check out The Elements of Statistical Learning Data Mining, Inference, and Prediction.

H(x)=extsign(∑Tt=1αtht(x)) [1]

Code:

```
import pandas as pd

import pylab as plt

import numpy as np

import scipy.optimize as opt

from sklearn import preprocessing
```

```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn import metrics
from tqdm import tqdm
def get_accuracy(X_train, X_test, y_train, y_test, model):
   return  {"test Accuracy":metrics.accuracy_score(y_test,
model.predict(X_test)),"train Accuracy": metrics.accuracy_score(y_train,
model.predict(X_train))}
def get_accuracy_bag(X,y,title,times=20,xlabel='Number
Estimators',Learning_rate_=[0.2,0.4,0.6,1]):

   lines_array=['solid','--', '-.', ':']

   N_estimators=[n for n in range(1,100)]

   times=20
   train_acc=np.zeros((times,len(Learning_rate_),len(N_estimators)))
   test_acc=np.zeros((times,len(Learning_rate_),len(N_estimators)))

   #Iterate through different number of Learning rate  and average out the results
   for n in tqdm(range(times)):
      X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.3)
      for n_estimators in N_estimators:
         for j,lr in enumerate(Learning_rate_):

            model =
AdaBoostClassifier(n_estimators=n_estimators+1,random_state=0,learning_rate=lr)

            model.fit(X_train,y_train)

Accuracy=get_accuracy(X_train, X_test, y_train, y_test,  model)
```

```
            train_acc[n,j,n_estimators-1]=Accuracy['train Accuracy']
            test_acc[n,j,n_estimators-1]=Accuracy['test Accuracy']



    fig, ax1 = plt.subplots()
    mean_test=test_acc.mean(axis=0)
    mean_train=train_acc.mean(axis=0)
    ax2 = ax1.twinx()

    for j,(lr,line) in enumerate(zip(Learning_rate_,lines_array)):

       ax1.plot(mean_train[j,:],linestyle = line,color='b',label="Learning rate "+str(lr))
       ax2.plot(mean_test[j,:],linestyle = line, color='r',label=str(lr))

    ax1.set_ylabel('Training accuracy',color='b')
    ax1.set_xlabel('No of estimators')
    ax1.legend()
    ax2.set_ylabel('Testing accuracy', color='r')
    ax2.legend()
    plt.show()
```

About the dataset

We will use a telecommunications dataset for predicting customer churn. This is a historical customer dataset where each row represents one customer. The data is relatively easy to understand, and you may uncover insights you can use immediately. Typically, it is less expensive to keep customers than to acquire new ones, so the focus of this analysis is to predict the customers who will stay with the company.

This data set provides information to help you predict what behavior will help you to retain customers. You can analyze all relevant customer data and develop focused customer retention programs.

The dataset includes information about:

- Customers who left within the last month – the column is called Churn

- Services that each customer has signed up for – phone, multiple lines, internet, online security, online backup, device protection, tech support, and streaming TV and movies
- Customer account information – how long they have been a customer, contract, payment method, paperless billing, monthly charges, and total charges
- Demographic info about customers – gender, age range, and if they have partners and dependents

Load Data From CSV File

```
[ ] churn_df = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML0101EN-SkillsNetwork/labs/Module%203/data/ChurnDa

    churn_df.head()
```

	tenure	age	address	income	ed	employ	equip	callcard	wireless	longmon	...	pager	internet	callwait	confer	ebill	loglong	logtoll	lninc	custcat	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	4.40	...	1.0	0.0	1.0	1.0	0.0	1.482	3.033	4.913	4.0	1.0
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	9.45	...	0.0	0.0	0.0	0.0	0.0	2.246	3.240	3.497	1.0	1.0
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	6.30	...	0.0	0.0	0.0	1.0	0.0	1.841	3.240	3.401	3.0	0.0
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	6.05	...	1.0	1.0	1.0	1.0	1.0	1.800	3.807	4.331	4.0	0.0
4	7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	7.10	...	0.0	0.0	1.0	1.0	0.0	1.960	3.091	4.382	3.0	0.0

5 rows × 28 columns

Data pre-processing and selection

Let's select some features for the modeling. Also, we change the target data type to be an integer, as it is a requirement by the skitlearn algorithm:

```
churn_df = churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip', 'callcard', 'wireless','churn']]
churn_df['churn'] = churn_df['churn'].astype('int')
churn_df.head()
```

	tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	1
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	1
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	0
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	0
4	7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	0

Select Variables at Random

Like Bagging, RF uses an independent bootstrap sample from the training data. In addition, we select m variables at random out of all M possible variables. Let's do an example.

```
[ ] X=churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip']]
```

there are 7 features

Train/Test dataset

Let's define X, and y for our dataset:

```
y = churn_df['churn']
y.head()
```

	churn
0	1
1	1
2	0
3	0
4	0

dtype: int64

Train/Test dataset

We split our dataset into train and test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.3, random_state=1)
print ('Train set', X_train.shape,  y_train.shape)
print ('Test set', X_test.shape,  y_test.shape)
```

```
Train set (140, 7) (140,)
Test set (60, 7) (60,)
```

AdaBoost

We can import the AdaBoost Classifier in Sklearn

```
[ ] from sklearn.ensemble import AdaBoostClassifier
```

The parameter `n_estimatorsint` is the maximum number of classifiers (default=50) at which boosting is stopped. If the results are perfect, the training procedure is stopped early.

```
[ ] n_estimators=5
    random_state=0
```

We can create a `AdaBoostClassifier` object.

+ Code + Text

```
[ ] model = AdaBoostClassifier(n_estimators=n_estimators,random_state=random_state)
```

If the outputs were y-1 and 1, the form of the classifier would be:

$$H(x) = extsign(\alpha_1 h_1(x) + \alpha_2 h_2(x) + \alpha_3 h_3(x) + \alpha_4 h_4(x) + \alpha_5 h_5(x))$$

We can fit the object finding all the $\alpha_t\ h_t(x)$ and then make a prediction:

```
[ ] model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_pred
```

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

We can find the training and testing accuracy:

```
print(get_accuracy(X_train, X_test, y_train, y_test,  model))
```

```
{'test Accuracy': 0.7666666666666667, 'train Accuracy': 0.7642857142857142}
```

We see the base model is a Decision Tree. Since it only has one layer, it's called a stump:

```
[ ] model.estimators_
```

```
[DecisionTreeClassifier(max_depth=1, random_state=209652396),
 DecisionTreeClassifier(max_depth=1, random_state=398764591),
 DecisionTreeClassifier(max_depth=1, random_state=924231285),
 DecisionTreeClassifier(max_depth=1, random_state=1478610112),
 DecisionTreeClassifier(max_depth=1, random_state=441365315)]
```

We see the weak classifiers do not perform as well:

```
[ ("for weak classifiers {} the we get ".format(i+1),get_accuracy(X_train, X_test, y_train, y_test,  weak_classifiers)) for i,weak_classifiers in enumerate(model.estimators_)]
```

```
[('for weak classifiers 1 the we get ',
  {'test Accuracy': 0.7, 'train Accuracy': 0.7428571428571429}),
 ('for weak classifiers 2 the we get ',
  {'test Accuracy': 0.6, 'train Accuracy': 0.6214285714285714}),
 ('for weak classifiers 3 the we get ',
  {'test Accuracy': 0.6333333333333333, 'train Accuracy': 0.6642857142857143}),
 ('for weak classifiers 4 the we get ',
  {'test Accuracy': 0.35, 'train Accuracy': 0.4642857142857143}),
 ('for weak classifiers 5 the we get ',
  {'test Accuracy': 0.4333333333333335, 'train Accuracy': 0.5})]
```

We can increase the number of weak classifiers:

```
[ ] n_estimators=100
    random_state=0
```

and then fit the model

```
[ ] model = AdaBoostClassifier(n_estimators=n_estimators,random_state=random_state)
    model.fit(X_train, y_train)

    #Predict the response for test dataset
    y_pred = model.predict(X_test)
```

We obtain the training and testing accuracy:

+ Code + Text

```
[ ] print(get_accuracy(X_train, X_test, y_train, y_test, model))
```

```
{'test Accuracy': 0.7333333333333333, 'train Accuracy': 1.0}
```

We see that adding more weak classifiers causes overfitting. We can verify by plotting the training and test accuracy over the number of classifiers:

We obtain the training and testing accuracy:

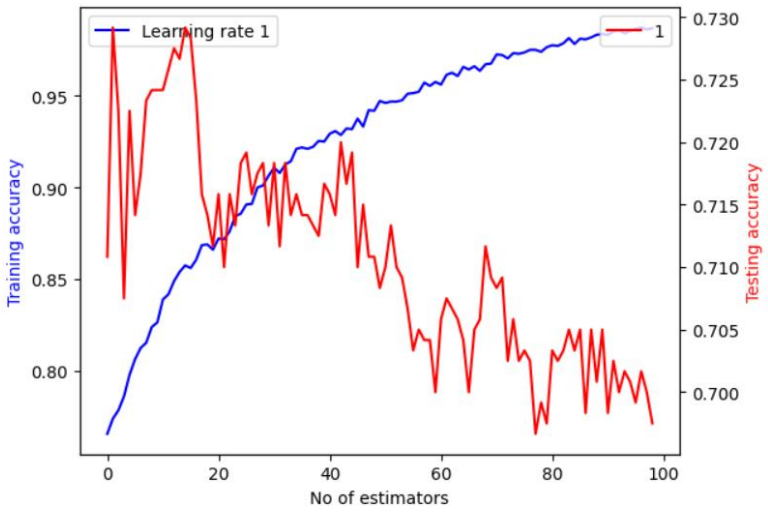
```
[ ] print(get_accuracy(X_train, X_test, y_train, y_test, model))
```

```
{'test Accuracy': 0.7333333333333333, 'train Accuracy': 1.0}
```

We see that adding more weak classifiers causes overfitting. We can verify by plotting the training and test accuracy over the number of classifiers:

```
[ ] get_accuracy_bag(X,y,title="Training and Test Accuracy vs Weak Classifiers",Learning_rate_=[1],times=20,xlabel='Number Estimators')
```

```
100%|██████████| 20/20 [04:13<00:00, 12.70s/it]
```



As you can see, as the number of classifiers increases so does the overfitting; the training accuracy increases and conversely, the testing accuracy decreases. One way to decrease overfitting is using the learning rate `learning_rate` with a default value of 1. This is a type of Regularization. For more detail on Regularization, check out here.

We can now train the model, make a prediction, and calculate the accuracy. We see that by increasing the learning rate the test accuracy has improved.

```
model = AdaBoostClassifier(n_estimators=n_estimators,random_state=random_state,learning_rate=learning_rate)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print(get_accuracy(X_train, X_test, y_train, y_test, model))
```

```
{'test Accuracy': 0.75, 'train Accuracy': 0.95}
```

+ Code + Text

Compared to the previous results we see the model does better on the test data. We can try different learning rates using the method `get_accuracy_bag`. In this case, the learning rates are 0.2,0.4,0.6, and 1. As the learning rate goes down we see that the testing accuracy increases while conversely, the training accuracy decreases .

```
[ ] get_accuracy_bag(X,y,title="Training and Test Accuracy vs Weak Classifiers",Learning_rate_=[0.2,0.4,0.6,1],times=20,xlabel='Number Estimators')
```

Another important parameter is `algorithm` with takes on the values `SAMME`, `SAMME.R`. The default is `'SAMME.R'`. The `SAMME.R` algorithm typically converges faster than `SAMME`, achieving a lower test error with fewer boosting iterations. For more details, check the paper. One issue is that `SAMME.R` can't be used all the time as we will need the Base classifier to generate the probability of belonging to each class.

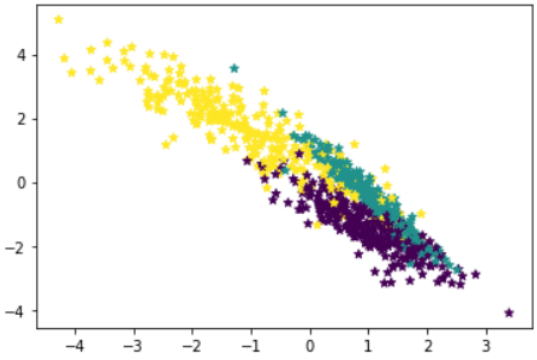
Practical 5

Aim: Generative Models.

a. Implement and demonstrate the working of a Naive Bayesian classifier using a sample data set. Build the model to classify a test sample.

In [1]:
```
from sklearn.datasets import make_classification

X, y = make_classification(
    n_features=6,
    n_classes=3,
    n_samples=800,
    n_informative=2,
    random_state=1,
    n_clusters_per_class=1,
)
```

In [2]:
```
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1], c=y, marker="*");
```



In [3]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=125
)
```

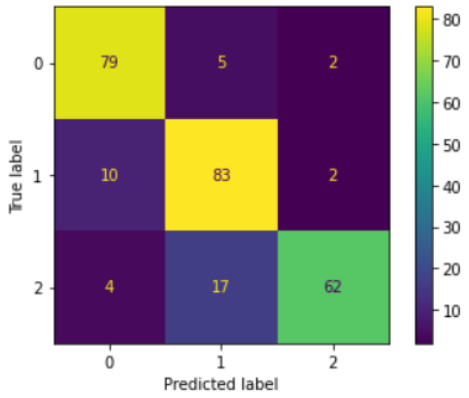
In [4]:
```
from sklearn.naive_bayes import GaussianNB

# Build a Gaussian Classifier
model = GaussianNB()

# Model training
model.fit(X_train, y_train)

# Predict Output
predicted = model.predict([X_test[6]])

print("Actual Value:", y_test[6])
print("Predicted Value:", predicted[0])
```

```
Actual Value: 0
Predicted Value: 0
```

In [5]:
```
from sklearn.metrics import (
    accuracy_score,
    confusion_matrix,
    ConfusionMatrixDisplay,
    f1_score,
)

y_pred = model.predict(X_test)
accuray = accuracy_score(y_pred, y_test)
f1 = f1_score(y_pred, y_test, average="weighted")

print("Accuracy:", accuray)
print("F1 Score:", f1)
```

```
Accuracy: 0.8484848484848485
F1 Score: 0.8491119695890328
```

In [6]:
```
labels = [0,1,2]
cm = confusion_matrix(y_test, y_pred, labels=labels)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
disp.plot();
```



In [7]:
```
#Naive Bayes Classifier with Loan Dataset
```

In [8]:
```
import pandas as pd


df = pd.read_csv('loan_data.csv')
df.head()
```

Out[8]:

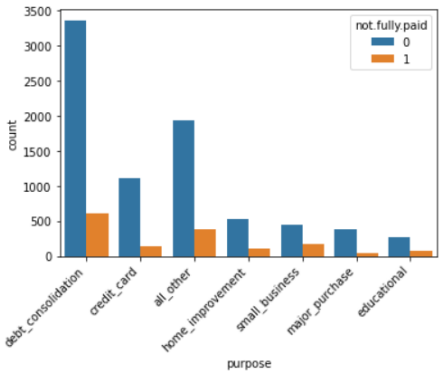
	credit.policy	purpose	int.rate	installment	log.annual.inc	dti	fico	days.with.cr.line	revol.bal	revol.util	inq.last.6mths	delinq.2yrs	pub.rec	not.f
0	1	debt_consolidation	0.1189	829.10	11.350407	19.48	737	5639.958333	28854	52.1	0	0	0	
1	1	credit_card	0.1071	228.22	11.082143	14.29	707	2760.000000	33623	76.7	0	0	0	
2	1	debt_consolidation	0.1357	366.86	10.373491	11.63	682	4710.000000	3511	25.6	1	0	0	
3	1	debt_consolidation	0.1008	162.34	11.350407	8.10	712	2699.958333	33667	73.2	1	0	0	
4	1	credit_card	0.1426	102.92	11.299732	14.97	667	4066.000000	4740	39.5	0	1	0	

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9578 entries, 0 to 9577
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   credit.policy      9578 non-null   int64
 1   purpose            9578 non-null   object
 2   int.rate           9578 non-null   float64
 3   installment        9578 non-null   float64
 4   log.annual.inc     9578 non-null   float64
 5   dti                9578 non-null   float64
 6   fico               9578 non-null   int64
 7   days.with.cr.line  9578 non-null   float64
 8   revol.bal          9578 non-null   int64
 9   revol.util         9578 non-null   float64
 10  inq.last.6mths     9578 non-null   int64
 11  delinq.2yrs        9578 non-null   int64
 12  pub.rec            9578 non-null   int64
 13  not.fully.paid     9578 non-null   int64
dtypes: float64(6), int64(7), object(1)
memory usage: 1.0+ MB
```

In [10]:
```
#n this example, we will be developing a model to predict the customers who have not fully paid the loan. Let's explore the pu
```

In [11]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.countplot(data=df,x='purpose',hue='not.fully.paid')
plt.xticks(rotation=45, ha='right');
```



In [12]:
```
#We will now convert the 'purpose' column from categorical to integer using pandas `get_dummies` function.
```

In [14]:
```
pre_df = pd.get_dummies(df,columns=['purpose'],drop_first=True)
pre_df.head()
```

Out[14]:

	credit.policy	int.rate	installment	log.annual.inc	dti	fico	days.with.cr.line	revol.bal	revol.util	inq.last.6mths	delinq.2yrs	pub.rec	not.fully.paid	purpos
0	1	0.1189	829.10	11.350407	19.48	737	5639.958333	28854	52.1	0	0	0	0	
1	1	0.1071	228.22	11.082143	14.29	707	2760.000000	33623	76.7	0	0	0	0	
2	1	0.1357	366.86	10.373491	11.63	682	4710.000000	3511	25.6	1	0	0	0	
3	1	0.1008	162.34	11.350407	8.10	712	2699.958333	33667	73.2	1	0	0	0	
4	1	0.1426	102.92	11.299732	14.97	667	4066.000000	4740	39.5	0	1	0	0	

In [15]:
```
#After that, we will define feature (X) and target (y) variables, and split the dataset into training and testing sets.
```

In [16]:
```
from sklearn.model_selection import train_test_split

X = pre_df.drop('not.fully.paid', axis=1)
y = pre_df['not.fully.paid']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=125
)
```

In [17]:
```
#Model building and training is quite simple. We will be training a model on a training dataset using default hyperparameters.
```

In [18]:
```
from sklearn.naive_bayes import GaussianNB

model = GaussianNB()

model.fit(X_train, y_train);
```

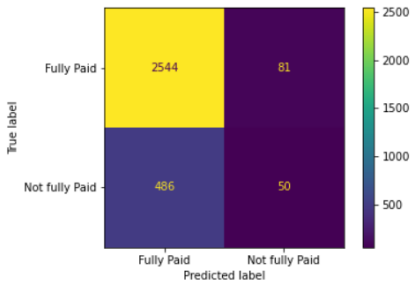
In [19]:
```
#We will use accuracy and f1 score to determine model performance, and it looks like the Gaussian Naive Bayes algorithm has perfo


from sklearn.metrics import (accuracy_score,
    confusion_matrix,
    ConfusionMatrixDisplay,
    f1_score,
    classification_report,
)
```

b.

In [20]:
```
y_pred = model.predict(X_test)

accuray = accuracy_score(y_pred, y_test)
f1 = f1_score(y_pred, y_test, average="weighted")

print("Accuracy:", accuray)
print("F1 Score:", f1)
```

```
Accuracy: 0.8206263840556786
F1 Score: 0.8686606980013266
```

In [21]:
```
labels = ["Fully Paid", "Not fully Paid"]
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)
disp.plot();
```



b. Implement Hidden Markov Models using hmmlearn

In [1]:
```
!pip install hmmlearn==0.2.6
```

```
Requirement already satisfied: hmmlearn==0.2.6 in c:\users\hina_\anaconda3\lib\site-packages (0.2.6)
Requirement already satisfied: numpy>=1.10 in c:\users\hina_\anaconda3\lib\site-packages (from hmmlearn==0.2.6) (1.19.2)
Requirement already satisfied: scikit-learn>=0.16 in c:\users\hina_\anaconda3\lib\site-packages (from hmmlearn==0.2.6) (1.3.2)
Requirement already satisfied: scipy>=0.19 in c:\users\hina_\anaconda3\lib\site-packages (from hmmlearn==0.2.6) (1.5.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\hina_\anaconda3\lib\site-packages (from scikit-learn>=0.16->hmm
learn==0.2.6) (2.1.0)
Requirement already satisfied: joblib>=1.1.1 in c:\users\hina_\anaconda3\lib\site-packages (from scikit-learn>=0.16->hmmlearn==
0.2.6) (1.4.2)
```

In [2]:
```
import hmmlearn
from hmmlearn import hmm
```

In [3]:
```
!pip show hmmlearn  # check that the version installed is 0.2.6
```

```
Name: hmmlearn
```

```
WARNING: Package(s) not found: #, 0.2.6, check, installed, is, that, the, version
```

```
Version: 0.2.6
Summary: Hidden Markov Models in Python with scikit-learn like API
Home-page: https://github.com/hmmlearn/hmmlearn
Author: None
Author-email: None
License: new BSD
Location: c:\users\hina_\anaconda3\lib\site-packages
Requires: scipy, numpy, scikit-learn
Required-by:
```

```
import numpy as np  # linear algebra
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns
from tqdm import tqdm
from matplotlib import pyplot as plt  # show graph
import random
```

```
import re
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\hina_\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[5]: True

In [6]:
```
from typing import List

from sklearn.model_selection import GroupShuffleSplit
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, precision_score, recall_score, \
    f1_score, roc_auc_score
```

In [7]:
```
#In this notebook we will look at the NER dataset and use it to understand HMM and also construct a POS tagger at the same time.¶
#Data Description:¶
```

In [8]:
```
#sentence: this column donates to which sentence the word belongs¶
#Word: the word in the sentence
#POS: Associated POS tag for the word
```

In [9]:
```
data = pd.read_csv("NER dataset.csv", encoding='latin1')
```

In [10]:
```
data = data.fillna(method="ffill")
data = data.rename(columns={'Sentence #': 'sentence'})
data.head(5)
```

Out[10]:

	sentence	Word	POS	Tag
0	Sentence: 1	Thousands	NNS	O
1	Sentence: 1	of	IN	O
2	Sentence: 1	demonstrators	NNS	O
3	Sentence: 1	have	VBP	O
4	Sentence: 1	marched	VBN	O

In [11]:
```
#Data pre-processing¶
#If you want to do some pre-processing (lowercase any words, remove stop words, replace numbers/names by a unique NUM/NAME tok

#Note : you could create a new dataset data_pre_precessed = pre_process(data) to keep both version and compare the effect of y
```

In [12]:
```
def pre_processing(text_column):
    # lowercase all text in the column
    text_column = text_column.str.lower()

    # replacing numbers with NUM token
    text_column = text_column.str.replace(r'\d+', 'NUM')

    # removing stopwords
    stop_words = set(stopwords.words('english'))
    text_column = text_column.apply(lambda x: ' '.join([word for word in x.split() if word not in stop_words]))

    return text_column
```

In [13]:
```
#The pre-processing step is performed to clean and transform the data in a way that is more suitable for further analysis. The

#First let's collect the unique words and the unique POS tags in the dataset, we will use this to construct the HMM later
```

In [14]:
```
data_pre_precessed = pre_processing(data.Word)
```

In [15]:
```
data_pre_precessed.head(20)
```

Out[15]:
```
0         thousands
1
2     demonstrators
3
4           marched
5
6            london
7
8           protest
9
10              war
11
12             iraq
13
14           demand
15
16       withdrawal
17
18          british
19           troops
Name: Word, dtype: object
```

In [16]:
```
#creating new dataframe with preprocessed word as a column
data_processed = data
data_processed['Word'] = data_pre_precessed
```

In [17]:
```
#removing the rows where word is empty
data_processed = data_processed[(data_processed['Word'] != '') | (data_processed['Word'].isna())]
```

In [18]:
```
data_processed.head(20)
```

Out[18]:

Out[18]:

	sentence	Word	POS	Tag
0	Sentence: 1	thousands	NNS	O
2	Sentence: 1	demonstrators	NNS	O
4	Sentence: 1	marched	VBN	O
6	Sentence: 1	london	NNP	B-geo
8	Sentence: 1	protest	VB	O
10	Sentence: 1	war	NN	O
12	Sentence: 1	iraq	NNP	B-geo
14	Sentence: 1	demand	VB	O
16	Sentence: 1	withdrawal	NN	O
18	Sentence: 1	british	JJ	B-gpe
19	Sentence: 1	troops	NNS	O
22	Sentence: 1	country	NN	O
23	Sentence: 1	.	.	O
24	Sentence: 2	families	NNS	O
26	Sentence: 2	soldiers	NNS	O
27	Sentence: 2	killed	VBN	O
30	Sentence: 2	conflict	NN	O
31	Sentence: 2	joined	VBD	O
33	Sentence: 2	protesters	NNS	O
35	Sentence: 2	carried	VBD	O

In [19]:
```
tags = list(set(data.POS.values))  # Unique POS tags in the dataset
words = list(set(data.Word.values))  # Unique words in the dataset
len(tags), len(words)
```

Out[19]: (42, 29764)

In [20]:
```
words1 = list(set(data_processed.Word.values))  # Unique words in the dataset
len(words1)
```

Out[20]: 29763

In [21]:
```
#We have 42 different tags and 29,764 different words, so the HMM that we construct will have the following properties¶
#The hidden states of the this HMM will correspond to the POS tags, so we will have 42 hidden states.
#The Observations for this HMM will correspond to the sentences and their words.
```

In [22]:
```
#Before constructing the HMM, we will split the data into train and test.
```

In [23]:
```
y = data.POS
X = data.drop('POS', axis=1)
```

In [24]:
```
gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix, test_ix = next(gs.split(X, y, groups=data['sentence']))
```

In [25]:
```
data_train = data.loc[train_ix]
data_test = data.loc[test_ix]
```

In [26]:
```
data_train.head(5)
```

Out[26]:

	sentence	Word	POS	Tag
24	Sentence: 2	families	NNS	O
25	Sentence: 2		IN	O
26	Sentence: 2	soldiers	NNS	O
27	Sentence: 2	killed	VBN	O
28	Sentence: 2		IN	O

In [27]:
```
data_test.head(5)
```

Out[27]:

	sentence	Word	POS	Tag
0	Sentence: 1	thousands	NNS	O
1	Sentence: 1		IN	O
2	Sentence: 1	demonstrators	NNS	O
3	Sentence: 1		VBP	O
4	Sentence: 1	marched	VBN	O

In [28]:
```
y1 = data_processed.POS
X1 = data_processed.drop('POS', axis=1)
data_processed.reset_index(drop=True, inplace=True)
gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix1, test_ix1 = next(gs.split(X1, y1, groups=data_processed['sentence']))
```

In [29]:
```
data_train1 = data_processed.loc[train_ix1]
data_test1 = data_processed.loc[test_ix1]
```

In [30]:
```
data_train1.head()
```

Out[30]:

	sentence	Word	POS	Tag
13	Sentence: 2	families	NNS	O
14	Sentence: 2	soldiers	NNS	O
15	Sentence: 2	killed	VBN	O
16	Sentence: 2	conflict	NN	O
17	Sentence: 2	joined	VBD	O

In [31]:
```
data_test1.head()
```

Out[31]:

	sentence	Word	POS	Tag
0	Sentence: 1	thousands	NNS	O
1	Sentence: 1	demonstrators	NNS	O
2	Sentence: 1	marched	VBN	O
3	Sentence: 1	london	NNP	B-geo
4	Sentence: 1	protest	VB	O

In [32]:
```
#Now lets encode the POS and Words to be used to generate the HMM.
```

In [33]:
```
dfupdate = data_train.sample(frac=.15, replace=False, random_state=42)
dfupdate.Word = 'UNKNOWN'
data_train.update(dfupdate)
words = list(set(data_train.Word.values))
# Convert words and tags into numbers
word2id = {w: i for i, w in enumerate(words)}
tag2id = {t: i for i, t in enumerate(tags)}
id2tag = {i: t for i, t in enumerate(tags)}
len(tags), len(words)
```

Out[33]: (42, 23607)

In [34]:
```
#In your theory classes you might have seen that the Hidden Markov Models can be learned by using the Baum-Welch
#algorithm by just using the observations. Although we can learn the Hidden States (POS tags) using Baum-Welch
#algorithm,We cannot map them back the states (words) to the POS tag. So for this exercise we will skip using the
#BW algorithm and directly create the HMM.
```

In [35]:
```
#For creating the HMM we should build the following three parameters.

#startprob_
#transmat_
#emissionprob_
```

In [36]:
```
count_tags = dict(data_train.POS.value_counts())  # Total number of POS tags in the dataset
# Now let's create the tags to words count
count_tags_to_words = data_train.groupby(['POS']).apply(
    lambda grp: grp.groupby('Word')['POS'].count().to_dict()).to_dict()
# We shall also collect the counts for the first tags in the sentence
count_init_tags = dict(data_train.groupby('sentence').first().POS.value_counts())
```

```
# Create a mapping that stores the frequency of transitions in tags to it's next tags
count_tags_to_next_tags = np.zeros((len(tags), len(tags)), dtype=int)
sentences = list(data_train.sentence)
pos = list(data_train.POS)
for i in tqdm(range(len(sentences)), position=0, leave=True):
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        prevtagid = tag2id[pos[i - 1]]
        nexttagid = tag2id[pos[i]]
        count_tags_to_next_tags[prevtagid][nexttagid] += 1
```

```
100%|██████████| 702936/702936 [00:01<00:00, 680765.97it/s]
```

In [37]:
```
#Now Let's build the parameter matrices
```

In [38]:
```
startprob = np.zeros((len(tags),))
transmat = np.zeros((len(tags), len(tags)))
emissionprob = np.zeros((len(tags), len(words)))
num_sentences = sum(count_init_tags.values())
sum_tags_to_next_tags = np.sum(count_tags_to_next_tags, axis=1)
for tag, tagid in tqdm(tag2id.items(), position=0, leave=True):
    floatCountTag = float(count_tags.get(tag, 0))
    startprob[tagid] = count_init_tags.get(tag, 0) / num_sentences
    for word, wordid in word2id.items():
        emissionprob[tagid][wordid] = count_tags_to_words.get(tag, {}).get(word, 0) / floatCountTag
    for tag2, tagid2 in tag2id.items():
        transmat[tagid][tagid2] = count_tags_to_next_tags[tagid][tagid2] / sum_tags_to_next_tags[tagid]
```

```
100%|██████████| 42/42 [00:00<00:00, 64.53it/s]
```

In [39]:
```
#ask 1: Similar to how we built the hidden state transition probability matrix as shown above, you will built the transition prob
```

In [40]:
```
#to create word transition matrix

#first step is to count the number of times each word appears in the dataset
count_words = {}
for word in data_train.Word.values:
    count_words[word] = count_words.get(word, 0) + 1

# then count the number of times a word appears after another word
count_word_transitions = {}
for sentence in data_train.groupby('sentence'):
    words = sentence[1]['Word'].values
    for i in range(len(words) - 1):
        w1, w2 = words[i], words[i+1]
        if w1 not in count_word_transitions:
            count_word_transitions[w1] = {}
        count_word_transitions[w1][w2] = count_word_transitions[w1].get(w2, 0) + 1

# convert the counts to probabilities
word_transition_matrix = np.zeros((len(word2id)+1, len(word2id)+1))
sum_words_to_next_words = np.sum([count_word_transitions[w1][w2] for w1 in count_word_transitions for w2 in count_word_transitior
for w1, w1id in word2id.items():
    for w2, w2id in word2id.items():
        word_transition_matrix[w1id][w2id] = count_word_transitions.get(w1, {}).get(w2, 0) / sum_words_to_next_words
print(word_transition_matrix.shape)
```

```
(23608, 23608)
```

[41]:
```
def calculate_log_likelihood(sentence: List[str], word_transition_matrix) -> float:
    """
    Given a sentence and word_transition_matrix, returns the log-likelihood of the sentence.
    """
    # converting the sentence to a list of word IDs
    sentence_ids = [word2id.get(w, word2id['UNKNOWN']) for w in sentence]

    # calculating the log-likelihood using the word transition matrix
    log_likelihood = np.log(word_transition_matrix[sentence_ids[0]][sentence_ids[1]])
    for i in range(1, len(sentence_ids) - 1):
        log_likelihood += np.log(word_transition_matrix[sentence_ids[i]][sentence_ids[i+1]] + 1e-10)
    return log_likelihood
```

In [42]:
```
calculate_log_likelihood(["This", "is", "a", "test", "sentence"], word_transition_matrix)
```

Out[42]:
```
-41.259970813020175
```

In [43]:
```
#In this task, a word transition probability matrix is built and a function that can calculate the log likelihood given a sentenc

#Then wrote a function to calculate the log-likelihood of a given sentence using the word transition probabilities. The sentence
```

In [44]:
```
#Now we will continue to constructing the HMM.¶
#We will use the hmmlearn implementation to initialize the HMM Model
```

In [50]:
```
model = hmm.MultinomialHMM(n_components=len(tags), algorithm='viterbi', random_state=42)
model.startprob_ = startprob
model.transmat_ = transmat
model.emissionprob_ = emissionprob
```

```
In [46]: #Before using the HMM to predict the POS tags, we have to fix the training set as some of the words and tags in the test data mig
```

```
In [51]: data_test.loc[~data_test['Word'].isin(words), 'Word'] = 'UNKNOWN'
word_test = list(data_test.Word)
samples = []
for i, val in enumerate(word_test):
    samples.append([word2id[val]])

# TODO use panda solution
lengths = []
count = 0
sentences = list(data_test.sentence)
for i in tqdm(range(len(sentences)), position=0, leave=True):
    if (i > 0) and (sentences[i] == sentences[i - 1]):
        count += 1
    elif i > 0:
        lengths.append(count)
        count = 1
    else:
        count = 1
```

```
100%|██████████| 345639/345639 [00:00<00:00, 1064414.93it/s]
```

```
In [49]: #Now that we have the HMM ready lets predict the best path from them.
pos_predict = model.predict(samples, lengths)
pos_predict
```

```
Out[49]: array([40, 37, 35, ..., 34, 30, 13])
```

Practical 6

Aim: Probabilistic Models

a. Implement Bayesian Linear Regression to explore prior and posterior distribution.

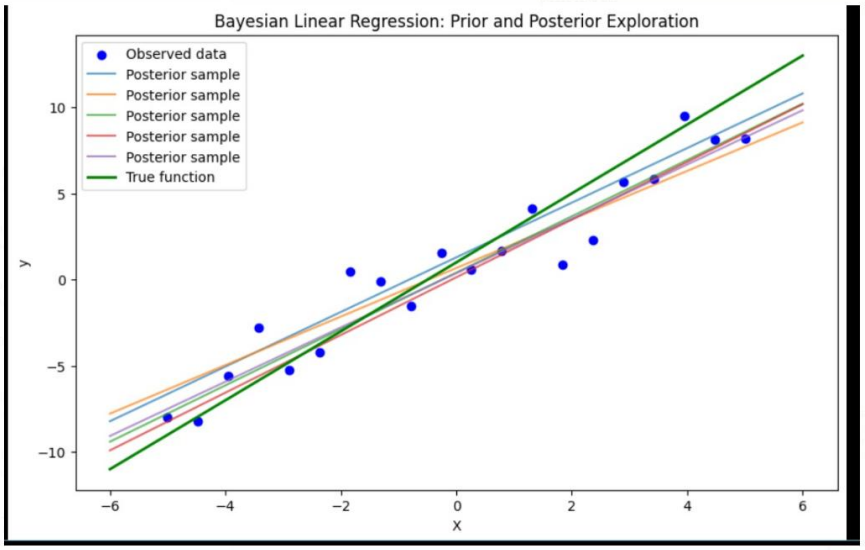
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal


# Generate synthetic data
np.random.seed(42)
X = np.linspace(-5, 5, 20)  # Input features
y = 2 * X + 1 + np.random.normal(0, 2, X.shape)  # True relationship with noise


# Add a bias term
X_design = np.vstack((np.ones_like(X), X)).T
```

```
# Define prior
prior_mean = np.array([0, 0])  # Prior mean
prior_covariance = np.eye(2) * 10  # Prior covariance

# Likelihood and posterior calculation
noise_variance = 4  # Variance of the noise
posterior_covariance = np.linalg.inv(
    np.linalg.inv(prior_covariance) + (X_design.T @ X_design) / noise_variance
)
posterior_mean = posterior_covariance @ (
    np.linalg.inv(prior_covariance) @ prior_mean + (X_design.T @ y) /
noise_variance
)

# Sample from the posterior
n_samples = 5
posterior_samples = np.random.multivariate_normal(posterior_mean,
posterior_covariance, n_samples)

# Plot data and posterior predictions
plt.figure(figsize=(10, 6))
plt.scatter(X, y, label="Observed data", color="blue")
X_test = np.linspace(-6, 6, 100)
X_test_design = np.vstack((np.ones_like(X_test), X_test)).T

for sample in posterior_samples:
    y_test = X_test_design @ sample
    plt.plot(X_test, y_test, label="Posterior sample", alpha=0.7)

# True function
plt.plot(X_test, 2 * X_test + 1, label="True function", color="green", linewidth=2)
```

```
plt.title("Bayesian Linear Regression: Prior and Posterior Exploration")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```

Output:



b. Implement Gaussian Mixture Models for density estimation and unsupervised clustering.

Code:

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "unsupervised_learning"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="jpg", resolution=1200):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
from sklearn.datasets import load_iris
```

```
[35]  1 data = load_iris()
      2 X = data.data
      3 y = data.target
      4 data.target_names
```

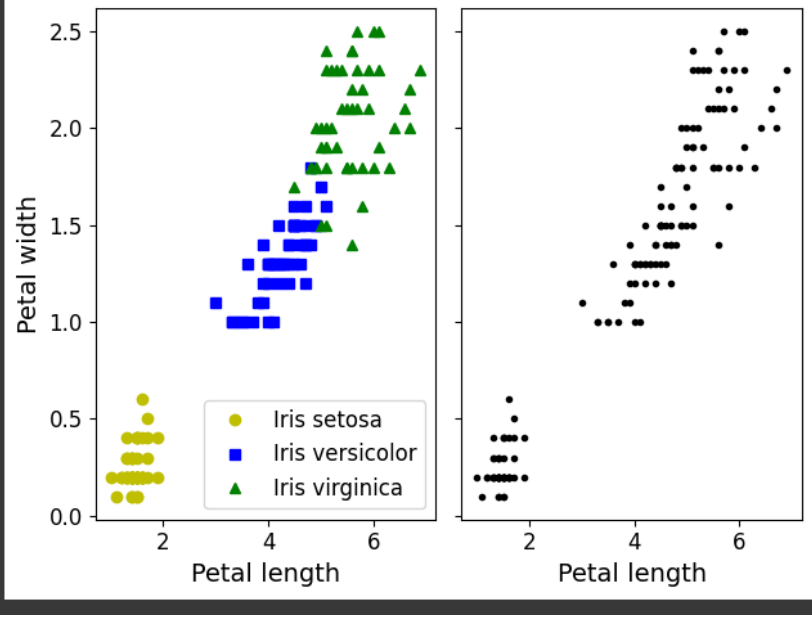
```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

```
1 plt.figure(figsize=(9, 3.5))
```

```
<Figure size 900x350 with 0 Axes>
<Figure size 900x350 with 0 Axes>
```

```
[68]  1 plt.subplot(121)
      2 plt.plot(X[y==0, 2], X[y==0, 3], "yo", label="Iris setosa")
      3 plt.plot(X[y==1, 2], X[y==1, 3], "bs", label="Iris versicolor")
      4 plt.plot(X[y==2, 2], X[y==2, 3], "g^", label="Iris virginica")
      5 plt.xlabel("Petal length", fontsize=14)
      6 plt.ylabel("Petal width", fontsize=14)
      7 plt.legend(fontsize=12)
      8 plt.subplot(122)
      9 plt.scatter(X[:, 2], X[:, 3], c="k", marker=".")
     10 plt.xlabel("Petal length", fontsize=14)
     11 plt.tick_params(labelleft=False)
     12 save_fig("classification_vs_clustering_plot")
     13 plt.show()
```

```
Saving figure classification_vs_clustering_plot
```




```
1 #A Gaussian mixture model (explained below) can actually separate these clusters pretty well (using all 4 fe
2 from sklearn.mixture import GaussianMixture
3 y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
```

```
[41]  1 #Let's map each cluster to a class. Instead of hard coding the mapping (as is done in the book, for simplici
```

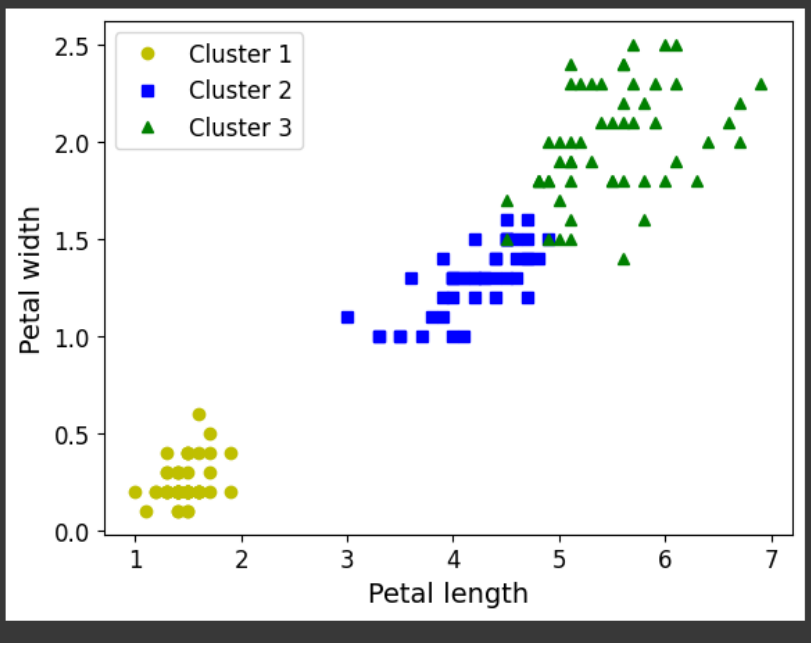
```
[42]  1 import numpy as np
      2 from scipy import stats
      3
      4 mapping = {}
      5 for class_id in np.unique(y):
      6     mode_result = stats.mode(y_pred[y == class_id], keepdims=True)  # `keepdims` is needed for compatibility
      7     mode = mode_result.mode[0]  # Access the mode from the `ModeResult` object
      8     mapping[mode] = class_id
      9
```

```
[43]  1 mapping
```

```
{1: 0, 2: 1, 0: 2}
```

```
[44]  1 y_pred = np.array([mapping[cluster_id] for cluster_id in y_pred])
```

```
plt.plot(X[y_pred==0, 2], X[y_pred==0, 3], "yo", label="Cluster 1")
plt.plot(X[y_pred==1, 2], X[y_pred==1, 3], "bs", label="Cluster 2")
plt.plot(X[y_pred==2, 2], X[y_pred==2, 3], "g^", label="Cluster 3")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=12)
plt.show()
```




```
[46] 1 np.sum(y_pred==y)
```

```
145
```

```
[47] 1 np.sum(y_pred==y) / len(y_pred)
```

```
0.9666666666666667
```

```
[48] 1 #Gaussian Mixtures
     2
```

```
1 gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
2 gm.fit(X)
```

```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

```
[50] 1 gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
     2 gm.fit(X)
```

```
GaussianMixture(n_components=3, n_init=10, random_state=42)
```

```
[51] 1 gm.weights_
```

```
array([0.30118609, 0.33333333, 0.36548058])
```

```
[53]  1 gm.means_

array([[5.91697517, 2.77803998, 4.20523542, 1.29841561],
       [5.006     , 3.428     , 1.462     , 0.246     ],
       [6.54632887, 2.94943079, 5.4834877 , 1.98716063]])

[54]  1 gm.covariances_

array([[[0.27550587, 0.09663458, 0.18542939, 0.05476915],
        [0.09663458, 0.09255531, 0.09103836, 0.04299877],
        [0.18542939, 0.09103836, 0.20227635, 0.0616792 ],
        [0.05476915, 0.04299877, 0.0616792 , 0.03232217]],

       [[0.121765  , 0.097232  , 0.016028  , 0.010124  ],
        [0.097232  , 0.140817  , 0.011464  , 0.009112  ],
        [0.016028  , 0.011464  , 0.029557  , 0.005948  ],
        [0.010124  , 0.009112  , 0.005948  , 0.010885  ]],

       [[0.38741443, 0.09223101, 0.30244612, 0.06089936],
        [0.09223101, 0.11040631, 0.08386768, 0.0557538 ],
        [0.30244612, 0.08386768, 0.32595958, 0.07283247],
        [0.06089936, 0.0557538 , 0.07283247, 0.08488025]]])

[55]  1 gm.converged_

True

[56]  1 gm.n_iter_

17
```

```
1 gm.predict(X)

array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])

[58]  1 gm.predict_proba(X)

       [9.99568375e-001, 1.94269937e-060, 4.31625248e-004],
       [9.99900419e-001, 5.87756309e-035, 9.95809797e-005],
       [9.94256042e-001, 1.44936206e-068, 5.74395807e-003],
       [9.97665901e-001, 4.03844987e-063, 2.33409881e-003],
       [9.98147774e-001, 3.85525817e-067, 1.85222572e-003],
       [9.99554475e-001, 1.68462043e-072, 4.45524964e-004],
       [9.99953439e-001, 3.30681507e-028, 4.65611857e-005],
       [9.98738347e-001, 1.90989312e-064, 1.26165336e-003],
       [1.29254164e-016, 3.13908163e-203, 1.00000000e+000],
       [1.57862631e-007, 1.76725715e-128, 9.99999842e-001],
       [4.50135537e-009, 3.22273399e-181, 9.99999995e-001],
       [5.63712653e-005, 1.36169343e-148, 9.99943629e-001],
       [1.46887543e-011, 9.27772115e-178, 1.00000000e+000],
       [5.21152316e-011, 1.39218538e-228, 1.00000000e+000],
       [1.47184041e-005, 3.28840631e-095, 9.99985282e-001],
       [2.84181556e-007, 1.49457125e-196, 9.99999716e-001],
       [9.32966009e-009, 1.10306495e-166, 9.99999991e-001],
       [9.57107490e-013, 1.86732538e-207, 1.00000000e+000],
       [7.66918118e-005, 1.44222575e-129, 9.99923308e-001],
       [2.50269349e-007, 8.01374099e-140, 9.99999750e-001],
       [9.53720203e-009, 9.26043250e-158, 9.99999990e-001],
       [4.02560940e-012, 1.96692631e-130, 1.00000000e+000],
       [8.48201094e-022, 2.04993235e-156, 1.00000000e+000],
       [1.74092396e-012, 9.49052261e-156, 1.00000000e+000],
       [9.69556924e-004, 6.70599987e-143, 9.99030443e-001],
       [7.17123049e-006, 2.77043245e-228, 9.99992829e-001],
       [1.29767785e-021, 7.72084918e-265, 1.00000000e+000],
       [7.06674419e-005, 1.39033929e-113, 9.99929333e-001],
```

```
1 X_new, y_new = gm.sample(6)
2 X_new
```

```
array([[5.83622262, 2.38632796, 3.83246674, 1.05938726],
       [4.54932764, 2.79918888, 1.5112294 , 0.2811497 ],
       [5.12603898, 3.63951039, 1.45067152, 0.0663176 ],
       [5.42433832, 4.14847238, 1.61928971, 0.34547167],
       [5.24310611, 3.85082984, 1.17992935, 0.17167228],
       [6.30073347, 3.21564689, 5.48932725, 2.328207  ]])
```

```
[60] 1 y_new
```

```
array([0, 1, 1, 1, 1, 2])
```

```
[61] 1 #Notice that they are sampled sequentially from each cluster.
     2
     3 #You can also estimate the log of the probability density function (PDF) at any location using the score_samples() method:
```

```
[62] 1 gm.score_samples(X)
```

```
array([ 1.57050082,  0.73787138,  1.14436656,  0.92913238,  1.411028  ,
       -0.09451903,  0.05266884,  1.62442195,  0.27082378,  0.16706624,
        0.83489877,  0.77168582,  0.29597841, -1.79224582, -3.41557928,
       -2.10529279, -1.12995447,  1.47503579, -0.84612536,  0.97699215,
       -0.92934784,  0.41079066, -3.83509616, -1.88906058, -3.17355662,
       -0.12403068,  0.51111724,  1.37663152,  1.12464925,  0.69029112,
        0.78206572, -0.69467132, -2.12834347, -0.8778815 ,  1.153231  ,
        0.11508687, -1.11928741,  0.22543724,  0.13115634,  1.49896403,
        0.94007659, -4.48978774, -0.34371496, -4.48037212, -2.58855877,
        0.67996207,  0.3937127 ,  1.04001332,  1.16051416,  1.54721281,
       -2.04219856, -0.26690497, -0.85097585, -2.32641778, -1.1625818 ,
       -0.79356973, -0.81650435, -1.40144428, -0.44903274, -1.64498537,
       -2.59281522, -0.60402676, -2.52104033, -0.11016408, -1.92916117,
       -1.15964891, -1.27337947, -2.94118443, -5.17236275,  0.26249387,
       -2.48742036, -0.02454445, -2.18957362, -2.5780906 ,  0.09261092,
       -0.47884423, -1.32879795, -2.26652606, -0.06028083, -1.86484538,
        0.12565062, -0.48783426,  0.53511418, -1.42580847, -2.52337005,
       -2.22487564, -0.31666947, -3.75463238, -0.44087896, -0.35405085,
```

```
[63] 1 #Let's check that the PDF integrates to 1 over the whole space.
     2 #We just take a large square around the clusters, and chop it into a grid of tiny squares,
     3 #then we compute the approximate probability that the instances will be generated
     4 #in each tiny square (by multiplying the PDF at one corner of the tiny square by the area
     5 #and finally summing all these probabilities). The result is very close to 1:
```

```
[64] 1 resolution = 100
     2 grid = np.arange(-10, 10, 1 / resolution)
     3 xx, yy = np.meshgrid(grid, grid)
     4 X_full = np.vstack([xx.ravel(), yy.ravel()]).T
     5
```

```
[65] 1 from sklearn.mixture import GaussianMixture
     2
     3 # Retrain with the correct number of features
     4 gm = GaussianMixture(n_components=4, random_state=0)  # Adjust n_components as needed
     5 gm.fit(X_full)
     6 pdf = np.exp(gm.score_samples(X_full))
     7 pdf_probas = pdf * (1 / resolution) ** 2
     8 print(pdf_probas.sum())
     9
```

```
0.8998770611010112
```

```
[ ] 1 Start coding or generate with AI.
```

Practical 7

Aim: Model Evaluation and Hyperparameter Tuning.

a. Implement cross-validation techniques (k-fold, stratified, etc.) for robust model evaluation.
b. Systematically explore combinations of hyperparameters to optimize model performance. (use grid and randomized search) (Refer Logistic Regression).

```
# STRATIFIES K-FOLD CROSS VALIDATION { 10-fold }
from statistics import mean, stdev
from sklearn import preprocessing
from sklearn.model_selection import StratifiedKFold
from sklearn import linear_model
from sklearn import datasets

# FEATCHING FEATURES AND TARGET VARIABLES IN ARRAY FORMAT.
cancer = datasets.load_breast_cancer()
# Input_x_Features.
x = cancer.data

# Input_ y_Target_Variable.
y = cancer.target

# Feature Scaling for input features.
scaler = preprocessing.MinMaxScaler()
x_scaled = scaler.fit_transform(x)

# Create classifier object.
lr = linear_model.LogisticRegression()
```

```
# Create StratifiedKFold object.
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=1)
lst_accu_stratified = []

for train_index, test_index in skf.split(x, y):
  x_train_fold, x_test_fold = x_scaled[train_index], x_scaled[test_index]
  y_train_fold, y_test_fold = y[train_index], y[test_index]
  lr.fit(x_train_fold, y_train_fold)
  lst_accu_stratified.append(lr.score(x_test_fold, y_test_fold))

# Print the output.
print('List of possible accuracy:', lst_accu_stratified)
print('\nMaximum Accuracy That can be obtained from this model is:',
  max(lst_accu_stratified)*100, '%')
print('\nMinimum Accuracy:',
  min(lst_accu_stratified)*100, '%')
print('\nOverall Accuracy:',
  mean(lst_accu_stratified)*100, '%')
print('\nStandard Deviation is:', stdev(lst_accu_stratified))
```

```
List of possible accuracy: [0.9298245614035088, 0.9649122807017544, 0.9824561403508771, 1.0, 0.9649122807017544, 0.9649122807017544,

Maximum Accuracy That can be obtained from this model is: 100.0 %

Minimum Accuracy: 92.98245614035088 %

Overall Accuracy: 96.66353383458647 %

Standard Deviation is: 0.02097789213195869
```

Practical 8

Aim: Bayesian Learning.

Bayesian Inference Learning

Code:

```
#Import the necessary libraries
import torch
import pyro
import pyro.distributions as dist
from pyro.infer import SVI, Trace_ELBO, Predictive
from pyro.optim import Adam
import matplotlib.pyplot as plt
import seaborn as sns

# Generate some sample data
torch.manual_seed(0)
X = torch.linspace(0, 10, 100)
true_slope = 2
true_intercept = 1
Y = true_intercept + true_slope * X + torch.randn(100)

# Define the Bayesian regression model
def model(X, Y):
    # Priors for the parameters
    slope = pyro.sample("slope", dist.Normal(0, 10))
    intercept = pyro.sample("intercept", dist.Normal(0, 10))
    sigma = pyro.sample("sigma", dist.HalfNormal(1))

    # Expected value of the outcome
```

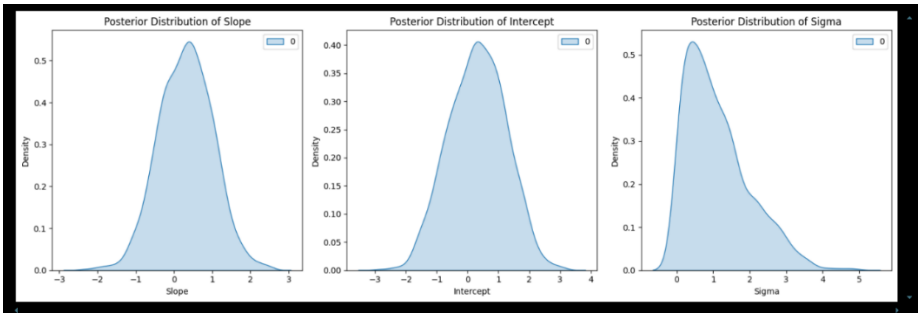
```
    mu = intercept + slope * X

    # Likelihood (sampling distribution) of the observations
    with pyro.plate("data", len(X)):
        pyro.sample("obs", dist.Normal(mu, sigma), obs=Y)

# Run Bayesian inference using SVI (Stochastic Variational Inference)
def guide(X, Y):
    # Approximate posterior distributions for the parameters
    slope_loc = pyro.param("slope_loc", torch.tensor(0.0))
    slope_scale = pyro.param("slope_scale",
torch.tensor(1.0),constraint=dist.constraints.positive)
    intercept_loc = pyro.param("intercept_loc", torch.tensor(0.0))
    intercept_scale = pyro.param("intercept_scale",
torch.tensor(1.0),constraint=dist.constraints.positive)
    sigma_loc = pyro.param("sigma_loc", torch.tensor(1.0),
constraint=dist.constraints.positive)

    # Sample from the approximate posterior distributions
    slope = pyro.sample("slope", dist.Normal(slope_loc, slope_scale))
    intercept = pyro.sample("intercept", dist.Normal(intercept_loc, intercept_scale))
    sigma = pyro.sample("sigma", dist.HalfNormal(sigma_loc))

# Initialize the SVI and optimizer
optim = Adam({"lr": 0.01})
svi = SVI(model, guide, optim, loss=Trace_ELBO())

# Run the inference loop
num_iterations = 1000
for i in range(num_iterations):
    loss = svi.step(X, Y)
```

```
    if (i + 1) % 100 == 0:
        print(f"Iteration {i + 1}/{num_iterations} - Loss: {loss}")

# Obtain posterior samples using Predictive
predictive = Predictive(model, guide=guide, num_samples=1000)
posterior = predictive(X, Y)

# Extract the parameter samples
slope_samples = posterior["slope"]
intercept_samples = posterior["intercept"]
sigma_samples = posterior["sigma"]

# Compute the posterior means
slope_mean = slope_samples.mean()
intercept_mean = intercept_samples.mean()
sigma_mean = sigma_samples.mean()

# Print the estimated parameters
print("Estimated Slope:", slope_mean.item())
print("Estimated Intercept:", intercept_mean.item())
print("Estimated Sigma:", sigma_mean.item())

# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot the posterior distribution of the slope
sns.kdeplot(slope_samples, shade=True, ax=axs[0])
axs[0].set_title("Posterior Distribution of Slope")
axs[0].set_xlabel("Slope")
axs[0].set_ylabel("Density")
```

```
# Plot the posterior distribution of the intercept
sns.kdeplot(intercept_samples, shade=True, ax=axs[1])
axs[1].set_title("Posterior Distribution of Intercept")
axs[1].set_xlabel("Intercept")
axs[1].set_ylabel("Density")

# Plot the posterior distribution of sigma
sns.kdeplot(sigma_samples, shade=True, ax=axs[2])
axs[2].set_title("Posterior Distribution of Sigma")
axs[2].set_xlabel("Sigma")
axs[2].set_ylabel("Density")

# Adjust the layout
plt.tight_layout()

# Show the plot
plt.show()
```

Output:

```
c:\Users\mohan\AppData\Local\Programs\Python\Python311\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. 
  from .autonotebook import tqdm as notebook_tqdm
Iteration 100/1000 - Loss: 68686.4638903141
Iteration 200/1000 - Loss: 1957.55493080616
Iteration 300/1000 - Loss: 647.4665781259537
Iteration 400/1000 - Loss: 788.4606136083603
Iteration 500/1000 - Loss: 3308.1984667778015
Iteration 600/1000 - Loss: 20155.736622929573
Iteration 700/1000 - Loss: 2545.918571889405
Iteration 800/1000 - Loss: 515579.75857794285
Iteration 900/1000 - Loss: 1881.5400274429321
Iteration 1000/1000 - Loss: 50892.460729599
Estimated Slope: 0.30964675545692444
Estimated Intercept: 0.31471437215805054
Estimated Sigma: 1.1101857423782349
C:\Users\mohan\AppData\Local\Temp\ipykernel_12676\1259798654.py:85: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(slope_samples, shade=True, ax=axs[0])
C:\Users\mohan\AppData\Local\Temp\ipykernel_12676\1259798654.py:91: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.
```



Practical 9

Aim: Deep Generative Models.

a. Set up a generator network to produce samples and a discriminator network to distinguish between real and generated data. (Use a simple small dataset)

Code:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np

# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

✓ 9.8s Python

```python
# Define a basic transform
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

✓ 0.0s Python

```python
train_dataset = datasets.CIFAR10(root='./data',\
    train=True, download=True, transform=transform)
dataloader = torch.utils.data.DataLoader(train_dataset, \
    batch_size=32, shuffle=True)
```

✓ 2.3s Python

```
Files already downloaded and verified
```

```python
# Hyperparameters
latent_dim = 100
lr = 0.0002
beta1 = 0.5
beta2 = 0.999
num_epochs = 10
```

✓ 0.0s Python

```python
# Define the generator
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128 * 8 * 8),
            nn.ReLU(),
            nn.Unflatten(1, (128, 8, 8)),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128, momentum=0.78),
            nn.ReLU(),
            nn.Upsample(scale_factor=2),
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64, momentum=0.78),
            nn.ReLU(),
            nn.Conv2d(64, 3, kernel_size=3, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img
```

✓ 0.0s

```python
# Define the discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
        nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.25),
        nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
        nn.ZeroPad2d((0, 1, 0, 1)),
        nn.BatchNorm2d(64, momentum=0.82),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
        nn.BatchNorm2d(128, momentum=0.82),
        nn.LeakyReLU(0.2),
        nn.Dropout(0.25),
        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256, momentum=0.8),
        nn.LeakyReLU(0.25),
        nn.Dropout(0.25),
        nn.Flatten(),
        nn.Linear(256 * 5 * 5, 1),
        nn.Sigmoid()
        )

    def forward(self, img):
        validity = self.model(img)
        return validity
```

✓ 0.0s

```python
# Define the generator and discriminator
# Initialize generator and discriminator
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)
# Loss function
adversarial_loss = nn.BCELoss()
# Optimizers
optimizer_G = optim.Adam(generator.parameters()\
                         , lr=lr, betas=(beta1, beta2))
optimizer_D = optim.Adam(discriminator.parameters()\
                         , lr=lr, betas=(beta1, beta2))
```

✓ 0.0s

```python
# Training loop
for epoch in range(num_epochs):
    for i, batch in enumerate(dataloader):
        # Convert list to tensor
        real_images = batch[0].to(device)
        # Adversarial ground truths
        valid = torch.ones(real_images.size(0), 1, device=device)
        fake = torch.zeros(real_images.size(0), 1, device=device)
        # Configure input
        real_images = real_images.to(device)

        # ---------------------
        #  Train Discriminator
        # ---------------------
        optimizer_D.zero_grad()
        # Sample noise as generator input
        z = torch.randn(real_images.size(0), latent_dim, device=device)
        # Generate a batch of images
        fake_images = generator(z)

        # Measure discriminator's ability
        # to classify real and fake images
        real_loss = adversarial_loss(discriminator\
                                     (real_images), valid)
        fake_loss = adversarial_loss(discriminator\
                                     (fake_images.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        # Backward pass and optimize
        d_loss.backward()
        optimizer_D.step()
```

```
            # -----------------
            #  Train Generator
            # -----------------

            optimizer_G.zero_grad()
            # Generate a batch of images
            gen_images = generator(z)
            # Adversarial loss
            g_loss = adversarial_loss(discriminator(gen_images), valid)
            # Backward pass and optimize
            g_loss.backward()
            optimizer_G.step()
            # ---------------------
            #  Progress Monitoring
            # ---------------------
            if (i + 1) % 100 == 0:
                print(
                    f"Epoch [{epoch+1}/{num_epochs}]\
                      Batch {i+1}/{len(dataloader)} "
                    f"Discriminator Loss: {d_loss.item():.4f} "
                    f"Generator Loss: {g_loss.item():.4f}"
                )
        # Save generated images for every epoch
        if (epoch + 1) % 10 == 0:
            with torch.no_grad():
                z = torch.randn(16, latent_dim, device=device)
                generated = generator(z).detach().cpu()
                grid = torchvision.utils.make_grid(generated,\
                                                   nrow=4, normalize=True)
                plt.imshow(np.transpose(grid, (1, 2, 0)))
                plt.axis("off")
                plt.show()
[8]  ✓ 94m 52.5s
```

Output:

```
Epoch [1/10]                    Batch 100/1563 Discriminator Loss: 0.6009 Generator Loss: 1.2484
Epoch [1/10]                    Batch 200/1563 Discriminator Loss: 0.5531 Generator Loss: 1.0415
Epoch [1/10]                    Batch 300/1563 Discriminator Loss: 0.5796 Generator Loss: 0.9874
Epoch [1/10]                    Batch 400/1563 Discriminator Loss: 0.5940 Generator Loss: 1.1672
Epoch [1/10]                    Batch 500/1563 Discriminator Loss: 0.5818 Generator Loss: 1.1546
Epoch [1/10]                    Batch 600/1563 Discriminator Loss: 0.5397 Generator Loss: 1.2826
Epoch [1/10]                    Batch 700/1563 Discriminator Loss: 0.5291 Generator Loss: 1.3528
Epoch [1/10]                    Batch 800/1563 Discriminator Loss: 0.6477 Generator Loss: 0.7852
Epoch [1/10]                    Batch 900/1563 Discriminator Loss: 0.7827 Generator Loss: 0.9457
Epoch [1/10]                    Batch 1000/1563 Discriminator Loss: 0.6312 Generator Loss: 0.7516
Epoch [1/10]                    Batch 1100/1563 Discriminator Loss: 0.5767 Generator Loss: 1.1332
Epoch [1/10]                    Batch 1200/1563 Discriminator Loss: 0.6649 Generator Loss: 0.7990
Epoch [1/10]                    Batch 1300/1563 Discriminator Loss: 0.9004 Generator Loss: 0.5943
Epoch [1/10]                    Batch 1400/1563 Discriminator Loss: 0.5979 Generator Loss: 0.9015
Epoch [1/10]                    Batch 1500/1563 Discriminator Loss: 0.6479 Generator Loss: 0.9679
Epoch [2/10]                    Batch 100/1563 Discriminator Loss: 0.7191 Generator Loss: 0.8595
Epoch [2/10]                    Batch 200/1563 Discriminator Loss: 0.6600 Generator Loss: 1.1399
Epoch [2/10]                    Batch 300/1563 Discriminator Loss: 0.5487 Generator Loss: 1.0457
Epoch [2/10]                    Batch 400/1563 Discriminator Loss: 0.6590 Generator Loss: 0.8623
Epoch [2/10]                    Batch 500/1563 Discriminator Loss: 0.7693 Generator Loss: 0.9907
Epoch [2/10]                    Batch 600/1563 Discriminator Loss: 0.5266 Generator Loss: 1.1867
Epoch [2/10]                    Batch 700/1563 Discriminator Loss: 0.7736 Generator Loss: 0.9273
Epoch [2/10]                    Batch 800/1563 Discriminator Loss: 0.7513 Generator Loss: 0.8284
Epoch [2/10]                    Batch 900/1563 Discriminator Loss: 0.7425 Generator Loss: 1.0712
Epoch [2/10]                    Batch 1000/1563 Discriminator Loss: 0.5084 Generator Loss: 1.1212
...
Epoch [10/10]                    Batch 1200/1563 Discriminator Loss: 0.4470 Generator Loss: 1.5577
Epoch [10/10]                    Batch 1300/1563 Discriminator Loss: 0.7464 Generator Loss: 0.6852
Epoch [10/10]                    Batch 1400/1563 Discriminator Loss: 0.6033 Generator Loss: 0.5511
Epoch [10/10]                    Batch 1500/1563 Discriminator Loss: 0.6797 Generator Loss: 1.1381
```