Machine Learning Python Lab

Stefano Di Lena

2024/2025

Contents

Chapter 1

Regression Lab

1.1 Importing Packages

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.pyplot import subplots
import seaborn as sns
```

These are some standard libraries used in machine learning coding.

- `numpy`: is used to create N-dimensional arrays, but it also offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.
- `pandas`: is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool.
- `matplotlib`: is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- `seaborn`: is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

1.2 Boston housing dataset

The `Boston` dataset records `medv` (median house value) for 506 neighborhoods around Boston. We will build a regression model to predict `medv` using 13 predictors such as `rmvar` (average number of rooms per house), `age` (proportion of owner-occupied units built prior to 1940), and `lstat` (percent of households with low socioeconomic status).

```python
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]

# Define the feature names, as they are not included in the dataset
feature_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
    'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT']

# Create the DataFrame using the defined feature names
df = pd.DataFrame(data, columns=feature_names)
df['MEDV'] = target
```

In these lines of code we:

1. upload the code from the URL (`http://lib.stat.cmu.edu/datasets/boston`)
2. extract the data with `pandas` (`pd.read_csv`):
   - `data_url`: specifies the URL of the dataset
   - $sep = "\backslash s+"$: make a separator (one or more white space are considered to divide the data, because them are not divided by comma or other delimiters)
   - `skiprows=22`: skips the first 22 rows of the file (because them contain metadata or dataset descriptions, that are not useful for data analysis)
   - `header=None`: say that there are no header rows (if we don't write this `pandas` interprets the first rows as header)
3. concatenate the data with `numpy.hstack`:
   - `raw_df.values[::2, :]`: combine the first 12 value from the even rows
   - `raw_df.values[1::2, :2]`: adds the last 2 values from the odd rows
4. set that our target is in the third position of the odd rows ($raw\_df.values[1::2,2]$)
5. define the feature names, as they are not included in the dataset
6. create the DataFrame utilizing `pd.DataFrame`, from the array data it adds the names of the columns (specified in `feature_names`) to identifier every feature.
7. add `MEDV` in the DataFrame, the values of this column we'll be the target value (who identifies the **Median Value of Owner-Occupied Homes**)

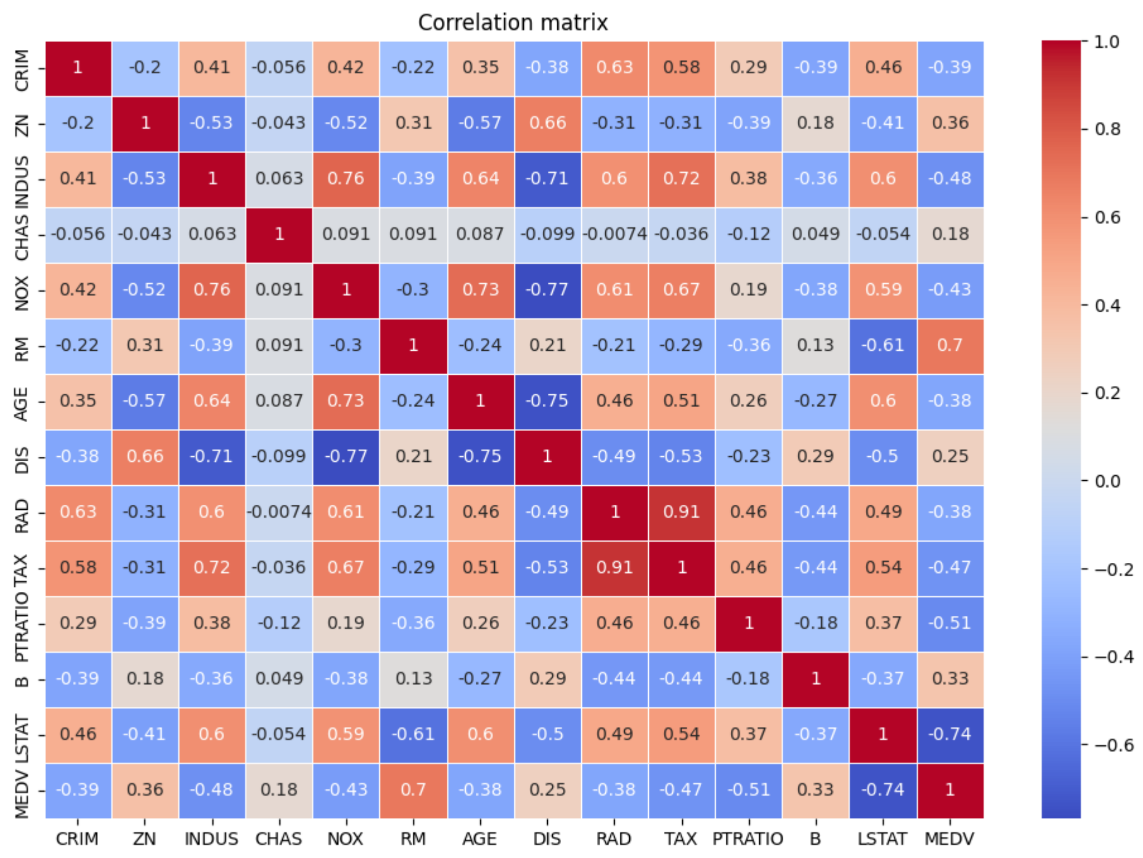
```python
# Descriptive statistics
print(df.describe())

# Check missing data
print(df.isnull().sum())
```

With `print(df.describe())` we produce a typical output that has:

- *count*: numbers of non-null values
- *mean*: mean of the values
- *std*: standard deviation
- *min*: minimum value
- *25%, 50%, 75%*: quartile
- *max*: maximum value

With `print(df.isnull().sum())` we verify the presence of NaN (null values) in each columns of the DataFrame.

```python
# Explore correlation between each couple of features
plt.figure(figsize=(12, 8))
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths
    =0.5)
plt.title('Correlation matrix')
plt.show()
```



In this code we:

1. set the dimensions of the figure in order to be more readable:

   `plt.figure(figsize=(12, 8))`

2. compute the Pearson correlation coefficient for each couple of numeric columns in the DataFrame with *df.corr*(). The result is a quadratic matrix in witch every element represent the level of correlation in two columns. (these correlation values vary from:

   (a) **1.0**: perfect positive correlation

   (b) **0.0**: no correlation

   (c) **-1.0**: perfect negative correlation

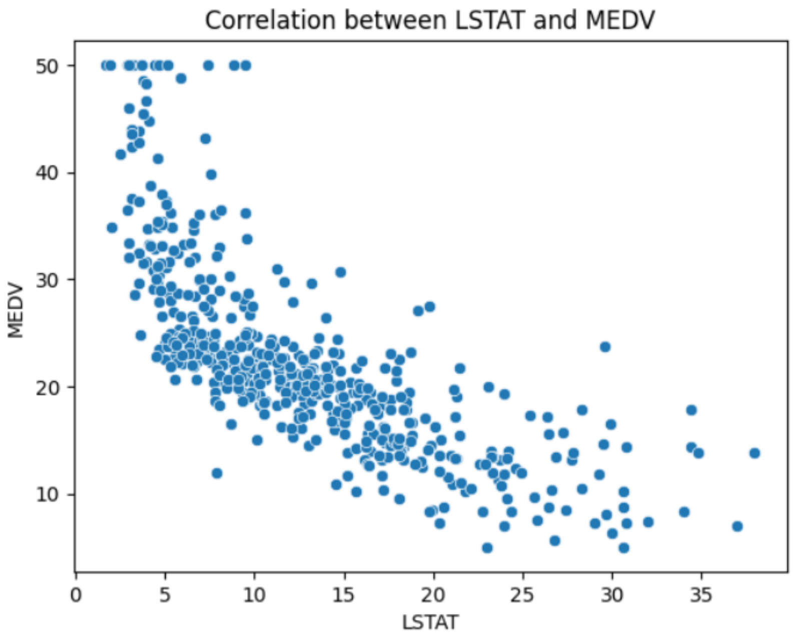
3. visualize the matrix with an heatmap, who associates the values of matrix at some colors (`sns.heatmap`):

   - `annot=True`: write the numeric value directly in the heatmap cells
   - `cmap='coolwarm'`: utilize the color palette 'coolwarm' to distinguish the correlation easily (the red ones are positive correlation and blue ones are negative)

4. added a title to the heatmap

5. plotted the heatmap

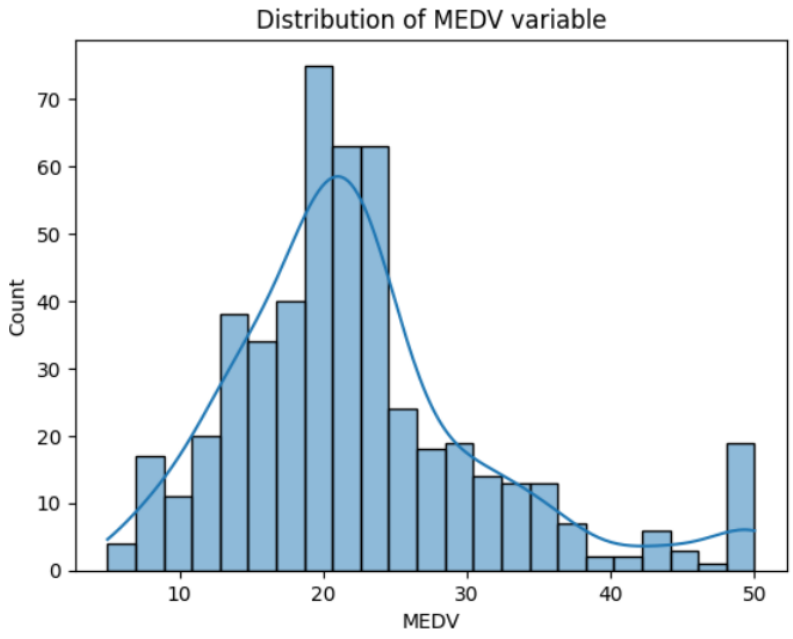
```
# Scatter plot between LSTAT and MEDV
sns.scatterplot(x='LSTAT', y='MEDV', data=df)
plt.title('Correlation between LSTAT and MEDV')
plt.show()
```



In this section:

1. we use scatterplot from the seaborn library to create the graph
   - `x='LSTAT'`: represent the variable LSTAT on the x-axis
   - `y='MEDV'`: represent the variable MEDV on the y-axis
   - `data=df`: specify that the data are from the DataFrame
2. we add a title for the graph and visualize it.

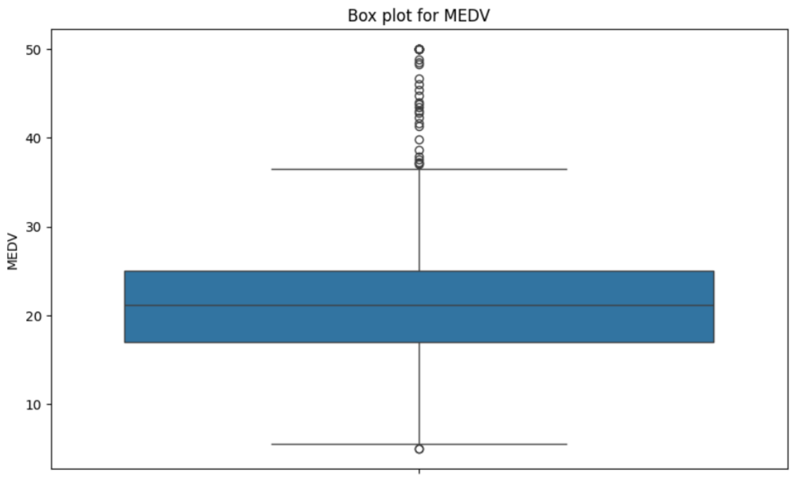
```
# Distribution of the target variable MEDV
sns.histplot(df['MEDV'], kde=True)
plt.title('Distribution of MEDV variable')
plt.show()
```



in this code:

1. we use histplot from the seaborn library to create an histogram
    - `df['MEDV']`: specify the variable to represent (the target, MEDV, in this case)
    - `kde=True`: overlap a KDE (Kernel Density Estimation) curve in the histogram. The KDE approx the real distribution of data with a continues function
2. we create a title and plot the graph

```
# Box plot for MEDV
plt.figure(figsize=(10, 6))
sns.boxplot(df['MEDV'])
plt.title('Box plot for MEDV')
plt.show()
```



In this section:

1. we make the graph more readable using `plt.figure(figsize=(10, 6))`
2. we use the boxplot function from the seaborn library to create the box plot. We specify the variable to represent with `df['MEDV']`
3. we add the title of the plot and we show it

1.3 Regression Models

```
import statsmodels.api as sm

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
```

1.3.1 Simple Linear Regression

```
X = df[['LSTAT']]
y = df['MEDV']

model_sk = LinearRegression()
model_sk.fit(X, y)
```

In this code we:

1. define the variable X with the features (in this case only LSTAT but we put [[]] because scikit-learn waits a bi-dimensional matrix
2. define de variable y containing our target, MEDV
3. create the linear regression model [this model compute a line that minimize the mean squared errors of the observed values y and the predicted ones. OLS (Ordinary Least Squares)]
4. fit the model using our feature and target (X,y)

```python
# Extract some basic information
print(f'Intercept: {model_sk.intercept_}')
print(f'Coefficient: {model_sk.coef_}')
```

We print in output the intercept of the model (β_0) and the coefficients (β_1).

```python
# Remember that you have to add a dummy variable before fitting OLS
    function
X = sm.add_constant(X)
model = sm.OLS(y, X).fit()
print(model.summary())
```

This code uses`statsmodels` library to do a linear regression analysis with the OLS method.

1. *sm.add_constant*(X): adds a column of ones to X (that represent the itercepts of the model)
2. *sm.OLS*$(y, X).fit()$: fit the OLS model, computing the optimal values for the parameters β_0, β_1
3. `print(model.summary())`: generate a detailed recap of the model statistics, that includes:
    - **coefficients**
    - **standard error**
    - **p-value**
    - **R-squared**
    - **F-statistic**

```python
# Verify the new dataset with the dummy variable
print(X)
```

This section print the new dataset that has an added column of ones at the start of the DaaFrame. If we don't do this, the model will think that the intercept is 0 and can lead to wrong interpretation or results inaccuracy.

```python
# Test new data
new_df = pd.DataFrame({'LSTAT':[5, 10, 15]})

print(new_df)
```

In this code, we create a new DataFrame whit a dictionary that has only a columns (**LSTAT**) whose values are (5, 10, 15). In ountput are printed the contents of this new DataFrame.

```python
# Get predictions
predictions_sk = model_sk.predict(new_df)
new_df1=new_df
# Add the prediction to the new dataframe new_df1
new_df1['Predictions'] = predictions_sk

print(new_df1)
```

This code use the trained model `model_sk` to make prediction on the **MEDV** values using the **LSTAT** values of the new DataFrame. We create a copy of the DataFrame at which we add a new column, called *Predictions*, that has the predicted values (*predictions_sk*). Finally, we print in output this new DataFrame.

```python
# Create a new DataFrame with only the necessary features for prediction
new_X = sm.add_constant(new_df[['LSTAT']])  # Select only the 'LSTAT'
    column

# Get the predictions and the confidence intervals
predictions = model.get_prediction(new_X)
conf_int = predictions.conf_int(alpha=0.05)  # alpha=0.05 for 95%
    confidence intervals
print(conf_int)
```

In this section we:

1. dd a column of constants (intercept) at the selected column (LSTAT) of the new_df
2. make predictions for MEDV using the new values of LSTAT, usinc the linear regression model
3. compute the confidence interval (95% for each prediction, because $alpha = 0.05 \rightarrow 1 - 0.05 = 95\%$)
4. we print in output the confidence intervals (a bi-dimensional array whose row are the inferior and superior limits for one observation)

```python
# Get the indivdual confidence intervals
predictions.conf_int(obs=True, alpha=0.05)
```

If we don't write nothing [as the example before] or write `obs=False`, we have the confidence intervals (whose output are the upper and lower limits for the conditioned mean of the target values for a specific value of an independent variable). When we write `obs=True` we consider the prevision interval (that include also the residual variability σ^2 associated at the individual observation)

1.3.2 Defining functions for Plotting data

```python
def abline(ax, b, m):
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m * xlim[0] + b, m * xlim[1] + b]
    ax.plot(xlim, ylim)
```

With the `abline` function we can add a rect to a graph.

- **ax**: is the object of matplotlib in which we add the line
- **b**:is the intercept of the line

- **m**: is the slope of the line

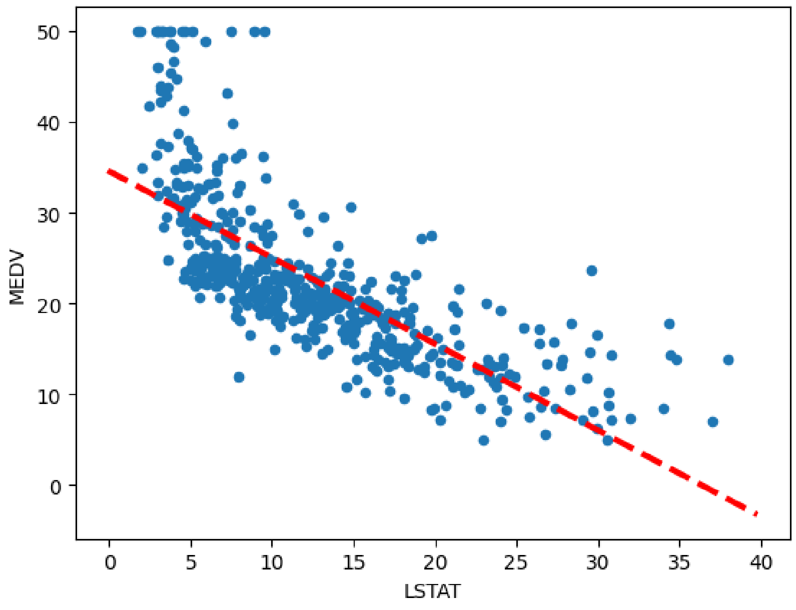
After that we obtain the x-axis limits and compute the y-axis ones using the rect equation $y = mx + b$. We can draw the line on the graph with `ax.plot(xlim, ylim)`.

```python
def abline(ax, b, m, *args, **kwargs):
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m * xlim[0] + b, m * xlim[1] + b]
    ax.plot(xlim, ylim, *args, **kwargs)
```

We used the same function as before but we added new parameters:

- with***args** we can pass extra arguments to the function (like an array or a list of values, to personalize the graph)
- with ****kwargs** we can pass the *color*, *linestyle* and *linewidth*

```python
ax = df.plot.scatter('LSTAT', 'MEDV')
abline(ax,
       model.params[0],
       model.params[1],
       'r--',
       linewidth=3)
```

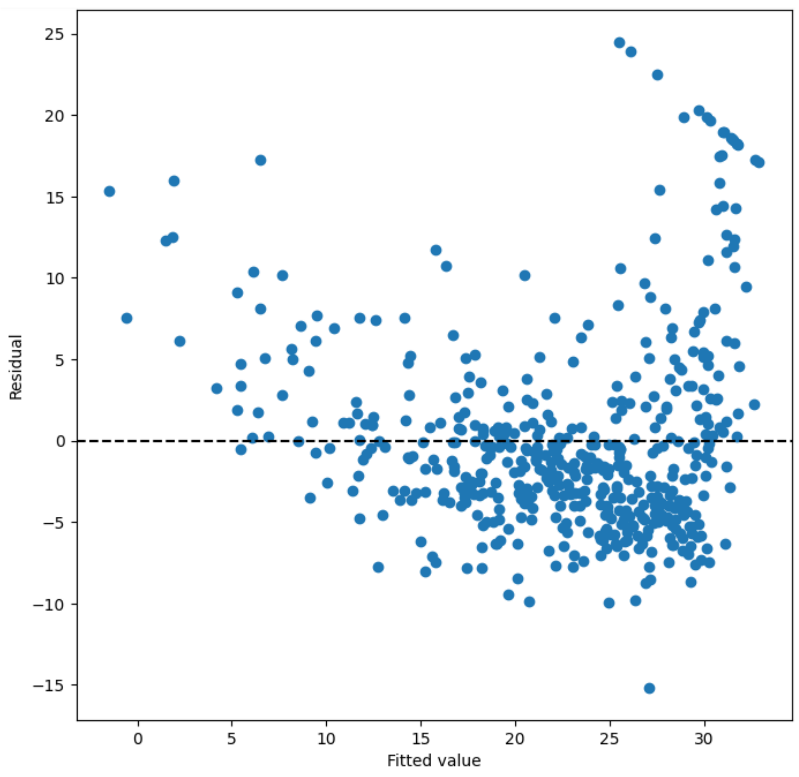


Using the DataFrame to create a scatter plot between LSTAT (x-axis) and MEDV (y-axis). We overlaps the regression line to the graph (*ax*) with abline, who contains the parameters of the predicted model.

- *model.params[0]*: the intercept (b)
- *model.params[1]*: the slope (m)

With **'r–'** we define the color (red) and the stile of the line (dashed). With `linewidth=3` we define the line thickness.

```python
ax = subplots(figsize=(8,8))[1]
ax.scatter(model.fittedvalues, model.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');
```



With this code a graph of residual is build to value if the model is appropriate.

1. `subplots(figsize=(8,8))`: create a 8x8 pixels figure, the **[1]** select the second axis of the array in output
2. create a scatter plot with the predicted values (`model.fittedvalues`) on the x-axis and the residual (`model.resid`) on the y-axis
3. we added some label for the axes
4. and with `axhline(0)` add an horizontal line (y=0) to represent the null residual, the color is setted on black (**'k'**) abd the line is dashed ('–')

1.3.3 Multiple Linear Regression

Multiple linear regression is used to fit a model with more variables, the next fit is done using two variables: lstat and age.

```
X = df[['LSTAT','AGE']]  # Select your predictor
y = df['MEDV']      # Response variable

X = sm.add_constant(X)
model1 = sm.OLS(y, X).fit()
print(model1.summary())
```

If we want to use all the 12 variables available in the Boston data set, we can do:

```
X = df[feature_names]  # Select your predictor
y = df['MEDV']      # Response variable

X = sm.add_constant(X)
model2 = sm.OLS(y, X).fit()
print(model2.summary())
```

In the above regression age has an high p-value so we can fit a model using all predictors except this one by doing:

```
new_feature_names=feature_names
new_feature_names.remove('AGE')
X = df[feature_names]  # Select your predictor
y = df['MEDV']     # Response variable

X = sm.add_constant(X)
model3 = sm.OLS(y, X).fit()
print(model3.summary())
```

1.3.4 Multivariate Goodness of Fit

We can access the individual components of model by name [dir(model) shows us what is available]. Hence model.rsquared gives us the R^2 , and np.sqrt(model.scale) gives us the RSE.

1.3.5 Interaction Terms

```
# Assuming df is your DataFrame
X = df[['LSTAT', 'AGE']]
y = df['MEDV']

# Add constant
X = sm.add_constant(X)

# Create interaction term manually
X['LSTAT_AGE'] = X['LSTAT'] * X['AGE']  # Create interaction term

model4 = sm.OLS(y, X).fit()
print(model4.summary())
```

With this code we make a multiple linear regression that as LSTAT and AGE as iteration terms.

A lot of steps are the same of the simple linear regression.

*X['LSTAT_AGE'] = X['LSTAT'] * X['AGE']* compute the iteration term whose can modify the combined effect of these two variable on the dependent variable MEDV.

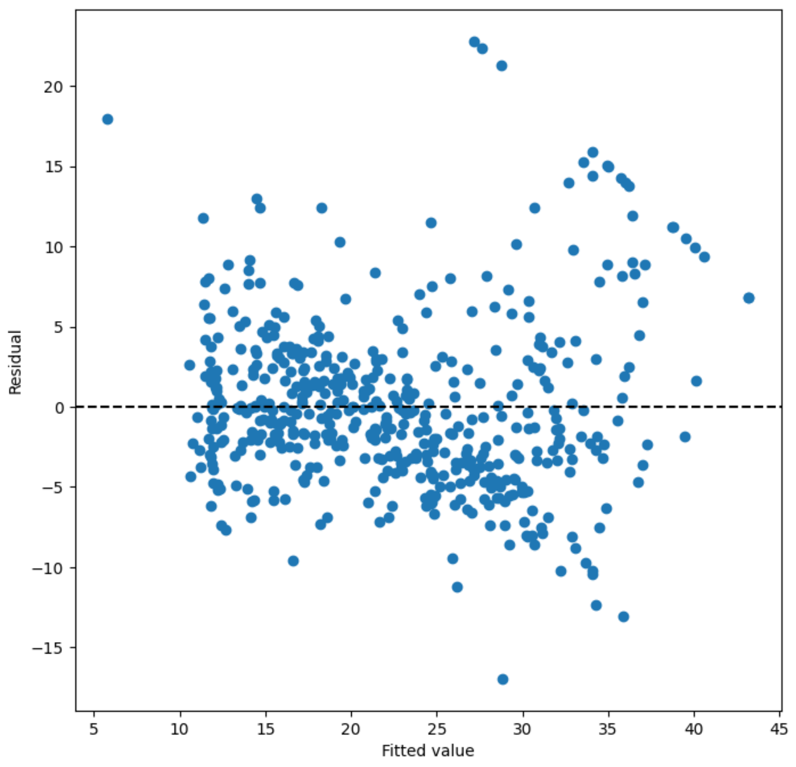
We make an Ordinary Least Square fit and print a detailed report of the result of the regression (coefficients, p-value, R-squared).

1.3.6 Non-linear Transformations of the Predictors

```
# Assuming df is your DataFrame
X = df[['LSTAT', 'AGE']]
y = df['MEDV']

# Add constant
X = sm.add_constant(X)

# Add LSTAT squared (LSTAT^2) using numpy
X['LSTAT_sq'] = np.power(X['LSTAT'], 2) # Create polynomial term

model5 = sm.OLS(y, X).fit()
print(model5.summary())
```

```
ax = subplots(figsize=(8,8))[1]
ax.scatter(model5.fittedvalues, model5.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');
```

We see that when the quadratic term is included in the model, there is little discernible pattern in the residuals.



1.3.7 Categorical Variables

For an example of linear regression with categorical variables, let's consider a hypothetical dataset related to houses, where we want to predict the sale price of a house based on various characteristics. The variables might include:

1. Size (in square meters) (numerical variable)

2. Number of rooms (numerical variable)

3. Neighborhood type (categorical variable: "Downtown", "Suburb", "Outskirts")

To include the categorical variable "Neighborhood type" in the model, we need to transform it into dummy variables. For example, if "Neighborhood type" has three categories (Downtown, Suburb, Outskirts), we can represent this variable using two dummy variables:

- Neighborhood_Suburb: 1 if the house is in the suburb, 0 otherwise.

- Neighborhood_Outskirts: 1 if the house is in the outskirts, 0 otherwise.

In the next code we:

1. create a dictionary (data)

2. create a DataFrame from that dictionary

3. create a dummy variable for *Neighborhood*, so the value (Downtown, Suburb, Outskirts) are converted in binary variables. Using `drop_first=True` one of the categories (Downtown) will be excluded as reference to avoid multicolinearity. The new columns are `Neighborhood_Suburb` (1 if the house is in the suburb area, 0 otherwise) and `Neighborhood_Outskirts` (1 if the house is in outskirts, 0 otherwise)

4. define the independent and dependent variables

5. ensure that all the columnn in X and y are numeric

6. add the intercept of the model

7. fit the model using the OLS methods

8. visualize a summary of the results of the model

```python
# Create the dataset
data = {
    'Size': [100, 80, 120, 75, 90, 85, 110, 130, 95, 105, 115, 70, 60, 150,
        140, 160],
    'Rooms': [3, 2, 4, 2, 3, 2, 4, 5, 3, 3, 4, 2, 1, 5, 4, 5],
    'Neighborhood': ['Downtown', 'Suburb', 'Outskirts', 'Downtown', 'Suburb
       ', 'Suburb', 'Outskirts', 'Downtown', 'Suburb', 'Outskirts', '
       Downtown', 'Suburb', 'Downtown', 'Outskirts', 'Downtown', 'Suburb'
       ],
    'Price': [300000, 200000, 400000, 250000, 220000, 210000, 380000,
       420000, 240000, 90000, 320000, 190000, 180000, 450000, 410000,
       460000]
}

df = pd.DataFrame(data)

# Create dummy variables for the categorical column 'Neighborhood'
df = pd.get_dummies(df, columns=['Neighborhood'], drop_first=True)

# Define independent (X) and dependent (y) variables
X = df[['Size', 'Rooms', 'Neighborhood_Suburb', 'Neighborhood_Outskirts']]
y = df['Price']

X = X.astype(float)  # Ensure all columns in X are numeric
y = y.astype(float)  # Ensure y is numeric

# Add a constant (intercept) to the model
X = sm.add_constant(X)

# Build the linear regression model
model = sm.OLS(y, X).fit()

# Display the summary of the model
print(model.summary())
```

Chapter 2

Classification Lab

2.1 The Stock Market Data

The `Smarket` data consists of percentae returns for the S&P 500 stock index over 1250 days, from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, `Lag1` through `Lag5`. We have also recorded `Volume` (the number of shares traded on the previous day, in billions), `Today` (the percentage return on the date in question) and `Direction` (whether the market was *Up* or *Down* on this date).

```python
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm

from sklearn.metrics import confusion_matrix

from sklearn.discriminant_analysis import \
    (LinearDiscriminantAnalysis as LDA,
     QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

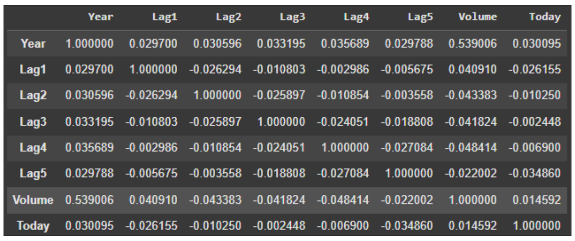
```python
from google.colab import drive

# Mount Google Drive
drive.mount('/content/drive')

# my path
csv_path = '/content/drive/MyDrive/Smarket.csv'

# Load the csv data
Smarket = pd.read_csv(csv_path)

# Check it!
print(Smarket.head())
```

```python
Smarket.columns
```

We compute the correlation matrix using the corr() method for data frames, which produces a matrix that contains all of the pairwise correlations among the variables.

By instructing pandas to use only numeric variables, the corr() method does not report a correlation for the Direction variable because it is qualitative.

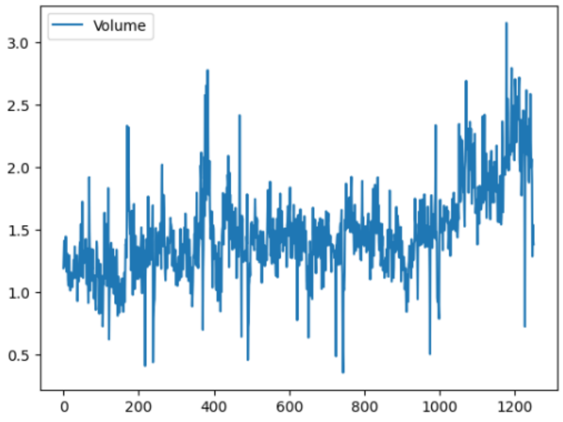
```python
Smarket.corr(numeric_only=True)
```



	Year	Lag1	Lag2	Lag3	Lag4	Lag5	Volume	Today
Year	1.000000	0.029700	0.030596	0.033195	0.035689	0.029788	0.539006	0.030095
Lag1	0.029700	1.000000	-0.026294	-0.010803	-0.002986	-0.005675	0.040910	-0.026155
Lag2	0.030596	-0.026294	1.000000	-0.025897	-0.010854	-0.003558	-0.043383	-0.010250
Lag3	0.033195	-0.010803	-0.025897	1.000000	-0.024051	-0.018808	-0.041824	-0.002448
Lag4	0.035689	-0.002986	-0.010854	-0.024051	1.000000	-0.027084	-0.048414	-0.006900
Lag5	0.029788	-0.005675	-0.003558	-0.018808	-0.027084	1.000000	-0.022002	-0.034860
Volume	0.539006	0.040910	-0.043383	-0.041824	-0.048414	-0.022002	1.000000	0.014592
Today	0.030095	-0.026155	-0.010250	-0.002448	-0.006900	-0.034860	0.014592	1.000000

As one would expect, the correlations between the lagged return variables and today's return are close to zero. The only substantial correlation is between Year and Volume. By plotting the data we see that Volume is increasing over time. In other words, the average number of shares traded daily increased from 2001 to 2005.

```python
Smarket.plot(y='Volume');
```



2.2 Logistic Regression

Next, we will fit a logistic regression model in order to predict Direction using Lag1 through Lag5 and Volume. The sm.GLM() function fits generalized linear models, a class of models that includes logistic regression. Alternatively, the function sm.Logit() fits a logistic regression model directly. The syntax of sm.GLM() is similar to that of sm.OLS(), except that we must pass in the argument family=sm.families.Binomial() in order to tell statsmodels to run a logistic regression rather than some other type of generalized linear model.

```python
allvars = Smarket.columns.drop(['Today', 'Direction', 'Year'])
X = Smarket[allvars].values
y = Smarket.Direction == 'Up'
glm = sm.GLM(y,
             X,
             family=sm.families.Binomial())
results = glm.fit()
print(results.summary())
```

The smallest p-value here is associated with Lag1. The negative coefficient for this predictor suggests that if the market had a positive return yesterday, then it is less likely to go up today. However, at a value of 0.15, the p-value is still relatively large, and so there is no clear evidence of a real association between Lag1 and Direction.

We use the params attribute of results in order to access just the coefficients for this fitted model.

```
results.params
```

Likewise we can use the pvalues attribute to access the p-values for the coefficients.

```
results.pvalues
```

The predict() method of results can be used to predict the probability that the market will go up, given values of the predictors. This method returns predictions on the probability scale. If no data set is supplied to the predict() function, then the probabilities are computed for the training data that was used to fit the logistic regression model. As with linear regression, one can pass an optional exog argument consistent with a design matrix if desired. Here we have printed only the first ten probabilities.

```
probs = results.predict()
probs[:10]
```

In order to make a prediction as to whether the market will go up or down on a particular day, we must convert these predicted probabilities into class labels, Up or Down. The following two commands create a vector of class predictions based on whether the predicted probability of a market increase is greater than or less than 0.5.

```
labels = np.array(['Down']*1250)
labels[probs>0.5] = "Up"
```

The confusion_matrix() function summarizes these predictions, showing how many observations were correctly or incorrectly classified.

```
conf_matrix = confusion_matrix(labels, Smarket.Direction)  # Generate
    confusion matrix
print(conf_matrix) # Display the matrix
```

The np.mean() function can be used to compute the fraction of days for which the prediction was correct. In this case, logistic regression correctly predicted the movement of the market 52% of the time.

```
np.mean(labels == Smarket.Direction)
```

At first glance, it appears that the logistic regression model is working a little better than random guessing. However, this result is misleading because we trained and tested the model on the same set of 1,250 observations. In other words, 100-52=48 is the training error rate. As we have seen previously, the training error rate is often overly optimistic — it tends to underestimate the test error rate. In order to better assess the accuracy of the logistic regression model in this setting, we can fit the model using part of the data, and then examine how well it predicts the held out data. This will yield a more realistic error rate, in the sense that in practice we will be interested in our model's performance not on the data that we used to fit the model, but rather on days in the future for which the market's movements are unknown.

To implement this strategy, we first create a Boolean vector corresponding to the observations from 2001 through 2004. We then use this vector to create a held out data set of observations from 2005.

```
train = (Smarket.Year < 2005)
Smarket_train = Smarket.loc[train]
Smarket_test = Smarket.loc[~train]
Smarket_test.shape
```

The object train is a vector of 1,250 elements, corresponding to the observations in our data set. The elements of the vector that correspond to observations that occurred before 2005 are set to True, whereas those that correspond to observations in 2005 are set to False. Hence train is a boolean array, since its elements are True and False. Boolean arrays can be used to obtain a subset of the rows or columns of a data frame using the loc method. For instance, the command Smarket.loc[train] would pick out a submatrix of the stock market data set, corresponding only to the dates before 2005, since those are the ones for which the elements of train are True. The $\sim$ symbol can be used to negate all of the elements of a Boolean vector. That is, $\sim train$ is a vector similar to train, except that the elements that are True in train get swapped to False in $\sim train$, and vice versa. Therefore, Smarket.loc[$\sim train$] yields a subset of the rows of the data frame of the stock market data containing only the observations for which train is False. The output above indicates that there are 252 such observations.

We now fit a logistic regression model using only the subset of the observations that correspond to dates before 2005. We then obtain predicted probabilities of the stock market going up for each of the days in our test set — that is, for the days in 2005.

```
# Convert X and y to pandas DataFrames or Series
X = pd.DataFrame(X)  # Convert X to DataFrame
y = pd.Series(y)     # Convert y to Series

X_train, X_test = X.loc[train], X.loc[~train]
y_train, y_test = y.loc[train], y.loc[~train]
glm_train = sm.GLM(y_train,
                   X_train,
                   family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
```

Notice that we have trained and tested our model on two completely separate data sets: training was performed using only the dates before 2005, and testing was performed using only the dates in 2005.

Finally, we compare the predictions for 2005 to the actual movements of the market over that time period. We will first store the test and training labels (recall y_test is binary).

```
D = Smarket.Direction
L_train, L_test = D.loc[train], D.loc[~train]
```

Now we threshold the fitted probability at 50% to form our predicted labels.

```
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_matrix(labels, L_test)
```

The test accuracy is about 58% while the error rate is about 42%

```
np.mean(labels == L_test), np.mean(labels != L_test)
```

The != notation means not equal to, and so the last command computes the test set error rate. The results are rather disappointing. Of course this result is not all that surprising, given that one would not generally expect to be able to use previous days' returns to predict future market performance.

We recall that the logistic regression model had very underwhelming p-values associated with all of the predictors, and that the smallest p-value, though not very small, corresponded to Lag1. Perhaps by removing the variables that appear not to be helpful in predicting Direction, we can obtain a more effective model. After all, using predictors that have no relationship with the response tends to cause a deterioration in the test error rate (since such predictors cause an increase in variance without a corresponding decrease in bias), and so removing such predictors may in turn yield an improvement. Below we refit the logistic regression using just Lag1 and Lag2, which seemed to have the highest predictive power in the original logistic regression model.

```
# Definizione delle feature e della variabile target
features = ['Lag1', 'Lag2']
X = Smarket[features]
X_train, X_test = X.loc[train], X.loc[~train]
glm_train = sm.GLM(y_train,
                   X_train,
                   family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_matrix(labels, L_test)
```

Let's evaluate the overall accuracy as well as the accuracy within the days when logistic regression predicts an increase.

```
(64+74)/252,74/(74+47)
```

It is worth noting that in this case, a much simpler strategy of predicting that the market will increase every day will also be correct 54% of the time! Hence, in terms of overall error rate, the logistic regression method is no better than the naive approach.

Suppose that we want to predict the returns associated with particular values of Lag1 and Lag2. In particular, we want to predict Direction on a day when Lag1 and Lag2 equal 1.2 and 1.1 , respectively, and on a day when they equal 1.5 and -0.8 . We do this using the predict() function.

```
# Nuovi dati per la predizione
newdata = pd.DataFrame({'Lag1': [1.2, 1.5],
                        'Lag2': [1.1, -0.8]})

# Probability predictions on the new data
new_probs = results.predict(newdata)
print(new_probs)
```

2.3 Linear Discriminant Analysis

We begin by performing LDA on the Smarket data, using the function LinearDiscriminantAnalysis(), which we have abbreviated LDA(). We fit the model using only the observations before 2005.

```
lda = LDA(store_covariance=True)
```

Since the LDA estimator automatically adds an intercept, we should remove the column corresponding to the intercept in both X_train and X_test. We can also directly use the labels rather than the Boolean vectors y_train.

```
# Definizione del modello LDA
lda = LDA()

# Fitting del modello LDA usando X_train e le etichette L_train
lda.fit(X_train, L_train)
```

Having fit the model, we can extract the means in the two classes with the means_ attribute. These are the average of each predictor within each class, and are used by LDA as estimates of μk . These suggest that there is a tendency for the previous 2 days' returns to be negative on days when the market increases, and a tendency for the previous days' returns to be positive on days when the market declines.

```
lda.means_
```

The estimated prior probabilities are stored in the priors_ attribute. The package sklearn typically uses this trailing _ to denote a quantity estimated when using the fit() method. We can be sure of which entry corresponds to which label by looking at the classes_ attribute.

```
lda.classes_
```

The LDA output indicates that $\hat{\pi}_{Down} = 0.492$ and $\hat{\pi}_{Up} = 0.508$.

```
lda.priors_
```

The linear discriminant vectors can be found in the scalings_ attribute:

```
lda.scalings_
```

These values provide the linear combination of Lag1 and Lag2 that are used to form the LDA decision rule. If $-0.64 \times Lag1 - 0.51 \times Lag2$ is lare, then the LDA classifier will predict a market increase, and if it is small, then the LDA classifier will predict a market decline.

```
lda_pred = lda.predict(X_test)
```

As we observed in our comparison of classification methods, the LDA and logistic regression predictions are almost identical.

```
confusion_matrix(lda_pred, L_test)
```

We can also estimate the probability of each class for each point in a training set. Applying a 50% threshold to the posterior probabilities of being in class one allows us to recreate the predictions contained in lda_pred.

```
lda_prob = lda.predict_proba(X_test)
np.all(
       np.where(lda_prob[:,1] >= 0.5, 'Up','Down') == lda_pred
       )
```

Above, we used the np.where() function that creates an array with value 'Up' for indices where the second column of lda_prob (the estimated posterior probability of 'Up') is greater than 0.5. For problems with more than two classes the labels are chosen as the class whose posterior probability is highest:

```
np.all(
       [lda.classes_[i] for i in np.argmax(lda_prob, 1)] == lda_pred
       )
```

If we wanted to use a posterior probability threshold other than 50% in order to make predictions, then we could easily do so. For instance, suppose that we wish to predict a market decrease only if we are very certain that the market will indeed decrease on that day — say, if the posterior probability is at least 90%. We know that the first column of lda_prob corresponds to the label Down after having checked the classes_ attribute, hence we use the column index 0 rather than 1 as we did above.

```
np.sum(lda_prob[:,0] > 0.9)
```

No days in 2005 meet that threshold! In fact, the greatest posterior probability of decrease in all of 2005 was 52.02

The LDA classifier above is the first classifier from the sklearn library. We will use several other objects from this library. Specifically, the methods first create a generic classifier without referring to any data. This classifier is then fit to data with the fit() method and predictions are always produced with the predict() method. This pattern of first instantiating the classifier, followed by fitting it, and then producing predictions is an explicit design choice of sklearn. This uniformity makes it possible to cleanly copy the classifier so that it can be fit on different data; e.g. different training sets arising in cross-validation. This standard pattern also allows for a predictable formation of workflows.

2.4 Quadratic Discriminant Analysis

```
qda = QDA(store_covariance=True)
qda.fit(X_train, L_train)
```

The QDA() function will again compute means and priors.

```
qda.means_, qda.priors_
```

The QDA() classifier will estimate one covariance per class. Here is the estimated covariance in the first class:

```
qda.covariance_[0]
```

The output contains the group means. But it does not contain the coefficients of the linear discriminants, because the QDA classifier involves a quadratic, rather than a linear, function of the predictors. The predict() function works in exactly the same fashion as for LDA.

```
qda_pred = qda.predict(X_test)
confusion_matrix(qda_pred, L_test)
```

Interestingly, the QDA predictions are accurate almost 60% of the time, even though the 2005 data was not used to fit the model.

```
np.mean(qda_pred == L_test)
```

This level of accuracy is quite impressive for stock market data, which is known to be quite hard to model accurately. This suggests that the quadratic form assumed by QDA may capture the true relationship more accurately than the linear forms assumed by LDA and logistic regression. However, we recommend evaluating this method's performance on a larger test set before betting that this approach will consistently beat the market!

2.5 Naive Bayes

Next, we fit a naive Bayes model to the Smarket data. The syntax is similar to that of LDA() and QDA(). By default, this implementation GaussianNB() of the naive Bayes classifier models each quantitative feature using a Gaussian distribution. However, a kernel density method can also be used to estimate the distributions.

```
NB = GaussianNB()
NB.fit(X_train, L_train)
```

The classes are stored as classes_.

```
NB.classes_
NB.class_prior_
```

```
NB.theta_

NB.var_
```

We can easily verify the mean computation:

```
X_train[L_train == 'Down'].mean()
```

Similarly for the variance:

```
X_train[L_train == 'Down'].var(ddof=0)
```

Since NB() is a classifier in the sklearn library, making predictions uses the same syntax as for LDA() and QDA() above.

```
nb_labels = NB.predict(X_test)
confusion_matrix(nb_labels, L_test)
```

As for LDA, the predict_proba() method estimates the probability that each observation belongs to a particular class.

```
NB.predict_proba(X_test)[:5]
```

2.6 K-Nearest Neighbors

```
knn1 = KNeighborsClassifier(n_neighbors=1)
X_train, X_test = [np.asarray(X) for X in [X_train, X_test]]
knn1.fit(X_train, L_train)
knn1_pred = knn1.predict(X_test)
confusion_matrix(knn1_pred, L_test)
```

The results using K=1 are not very good, since only 50 of the observations are correctly predicted. Of course, it may be that K=1 results in an overly-flexible fit to the data.

```
(83+43)/252, np.mean(knn1_pred == L_test)
```

We repeat the analysis below using K=3.

```
knn3 = KNeighborsClassifier(n_neighbors=3)
knn3_pred = knn3.fit(X_train, L_train).predict(X_test)
np.mean(knn3_pred == L_test)
```

The results have improved slightly. But increasing K further provides no further improvements. It appears that for these data, and this train/test split, QDA gives the best results of the methods that we have examined so far.

KNN does not perform well on the Smarket data, but it often does provide impressive results. As an example we will apply the KNN approach to the Caravan data set. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is Purchase, which indicates whether or not a given individual purchases a caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

```
# Monta Google Drive
drive.mount('/content/drive')

# Specifica il percorso del CSV che vuoi caricare
csv_path = '/content/drive/MyDrive/Caravan.csv'
```

```python

# Carica il CSV in un DataFrame
Caravan = pd.read_csv(csv_path)

# Visualizza i primi cinque record per verificare che il caricamento sia
    stato corretto
print(Caravan.head())
```

```python
Purchase = Caravan.Purchase
Purchase.value_counts()
```

The method value_counts() takes a pd.Series or pd.DataFrame and returns a pd.Series with the corresponding counts for each unique element. In this case Purchase has only Yes and No values and the method returns how many values of each there are.

```python
348 / 5822
```

Our features will include all columns except Purchase.

```python
feature_df = Caravan.drop(columns=['Purchase'])
```

Because the KNN classifier predicts the class of a given test observation by identifying the observations that are nearest to it, the scale of the variables matters. Any variables that are on a large scale will have a much larger effect on the distance between the observations, and hence on the KNN classifier, than variables that are on a small scale. For instance, imagine a data set that contains two variables, salary and age (measured in dollars and years, respectively). As far as KNN is concerned, a difference of 1,000 USD in salary is enormous compared to a difference of 50 years in age. Consequently, salary will drive the KNN classification results, and age will have almost no effect. This is contrary to our intuition that a salary difference of 1,000 USD is quite small compared to an age difference of 50 years. Furthermore, the importance of scale to the KNN classifier leads to another issue: if we measured salary in Japanese yen, or if we measured age in minutes, then we'd get quite different classification results from what we get if these two variables are measured in dollars and years.

A good way to handle this problem is to standardize the data so that all variables are given a mean of zero and a standard deviation of one. Then all variables will be on a comparable scale. This is accomplished using the StandardScaler() transformation.

```python
scaler = StandardScaler(with_mean=True,
                        with_std=True,
                        copy=True)
```

The argument with_mean indicates whether or not we should subtract the mean, while with_std indicates whether or not we should scale the columns to have standard deviation of 1 or not. Finally, the argument copy=True indicates that we will always copy data, rather than trying to do calculations in place where possible.

This transformation can be fit and then applied to arbitrary data. In the first line below, the parameters for the scaling are computed and stored in scaler, while the second line actually constructs the standardized set of features.

```python
scaler.fit(feature_df)
X_std = scaler.transform(feature_df)
```

Now every column of feature_std below has a standard deviation of one and a mean of zero.

```python
feature_std = pd.DataFrame(
                 X_std,
                 columns=feature_df.columns);
feature_std.std()
```

Notice that the standard deviations are not quite 1 here; this is again due to some procedures using the 1/n convention for variances (in this case scaler()), while others use 1/(n-1) (the std() method). In this case it does not matter, as long as the variables are all on the same scale.

Using the function train_test_split() we now split the observations into a test set, containing 1000 observations, and a training set containing the remaining observations. The argument random_state=0 ensures that we get the same split each time we rerun the code.

```
(X_train,
 X_test,
 y_train,
 y_test) = train_test_split(np.asarray(feature_std),
                            Purchase,
                            test_size=1000,
                            random_state=0)
```

?train_test_split reveals that the non-keyword arguments can be lists, arrays, pandas dataframes etc that all have the same length (shape[0]) and hence are indexable. In this case they are the dataframe feature_std and the response variable Purchase. {Note that we have converted feature_std to an ndarray to address a bug in sklearn.} We fit a KNN model on the training data using K=1 , and evaluate its performance on the test data.

```
knn1 = KNeighborsClassifier(n_neighbors=1)
knn1_pred = knn1.fit(X_train, y_train).predict(X_test)
np.mean(y_test != knn1_pred), np.mean(y_test != "No")
```

The KNN error rate on the 1,000 test observations is about 11 . At first glance, this may appear to be fairly good. However, since just over 6% of customers purchased insurance, we could get the error rate down to almost 6% by always predicting No regardless of the values of the predictors! This is known as the null rate.}

Suppose that there is some non-trivial cost to trying to sell insurance to a given individual. For instance, perhaps a salesperson must visit each potential customer. If the company tries to sell insurance to a random selection of customers, then the success rate will be only 6%, which may be far too low given the costs involved. Instead, the company would like to try to sell insurance only to customers who are likely to buy it. So the overall error rate is not of interest. Instead, the fraction of individuals that are correctly predicted to buy insurance is of interest.

```
confusion_matrix(knn1_pred, y_test)
```

It turns out that KNN with K=1 does far better than random guessing among the customers that are predicted to buy insurance. Among 62 such customers, 9, or 14.5%, actually do purchase insurance. This is double the rate that one would obtain from random guessing.

```
9/(53+9)
```

2.6.1 Tuning Parameters

The number of neighbors in KNN is referred to as a tuning parameter, also referred to as a hyperparameter. We do not know a priori what value to use. It is therefore of interest to see how the classifier performs on test data as we vary these parameters. This can be achieved with a for loop. Here we use a for loop to look at the accuracy of our classifier in the group predicted to purchase insurance as we vary the number of neighbors from 1 to 5:

```
for K in range(1,6):
    knn = KNeighborsClassifier(n_neighbors=K)
    knn_pred = knn.fit(X_train, y_train).predict(X_test)
    C = confusion_matrix(knn_pred, y_test)
    # Convert the confusion matrix to a Pandas DataFrame for using .loc
    C = pd.DataFrame(C, index=['No', 'Yes'], columns=['No', 'Yes'])
    templ = ('K={0:d}: # predicted to rent: {1:>2},' +
            '  # who did rent {2:d}, accuracy {3:.1%}')
    pred = C.loc['Yes'].sum()
    did_rent = C.loc['Yes','Yes']
    print(templ.format(
          K,
          pred,
          did_rent,
          did_rent / pred))
```

2.6.2 Comparison to Logistic Regression

As a comparison, we can also fit a logistic regression model to the data. This can also be done with sklearn, though by default it fits something like the ridge regression version of logistic regression. This can be modified by appropriately setting the argument C below. Its default value is 1 but by setting it to a very large number, the algorithm converges to the same solution as the usual (unregularized) logistic regression estimator discussed above.

Unlike the statsmodels package, sklearn focuses less on inference and more on classification. Hence, the summary methods seen in statsmodels and our simplified version seen with summarize are not generally available for the classifiers in sklearn.

```
logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict_proba(X_test)
logit_labels = np.where(logit_pred[:,1] > .5, 'Yes', 'No')
confusion_matrix(logit_labels, y_test)
```

We used the argument solver='liblinear' above to avoid a warning with the default solver which would indicate that the algorithm does not converge.

If we use 0.5 as the predicted probability cut-off for the classifier, then we have a problem: only two of the test observations are predicted to purchase insurance. However, we are not required to use a cut-off of 0.5 . If we instead predict a purchase any time the predicted probability of purchase exceeds 0.25 , we get much better results: we predict that 29 people will purchase insurance, and we are correct for about 31% of these people. This is almost five times better than random guessing!

```
logit_labels = np.where(logit_pred[:,1]>0.25, 'Yes', 'No')
confusion_matrix(logit_labels, y_test)

9/(20+9)
```

Chapter 3

Lasso and Ridge Regression

3.1 Data Importation and EDA

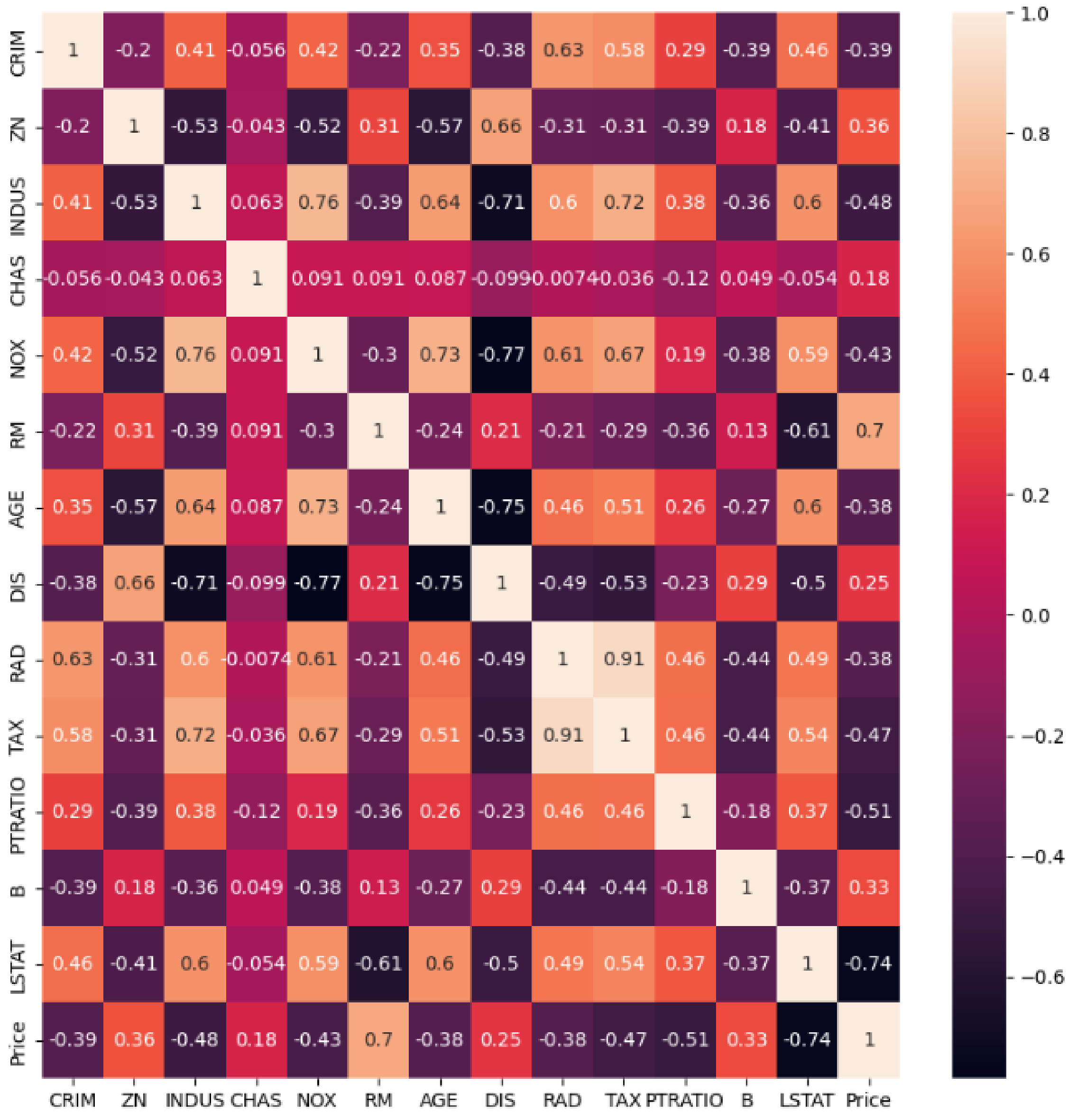
```python
# Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge, RidgeCV, Lasso
from sklearn.preprocessing import StandardScaler

# Load the Boston dataset directly from its source
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]

boston_df = pd.DataFrame(data, columns=['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX
    ', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'])
boston_df['Price'] = target
# Preview
boston_df.head()

# Exploration
plt.figure(figsize = (10, 10))
sns.heatmap(boston_df.corr(), annot = True)
```

After uploading the libraries and our dataset, we create a DataFrame with these data. Later, with *np.hstack* we concatenate horizontally the two numpy array created by the slicing:

- $raw\_df.values[::2, :]$ who takes all the column of the dataset rows that has even index
- $raw\_df.values[1::2, :2]$ who takes the first two column of the dataset rows that has an odd index

We select also the target from the third column (2) of the rows that has an odd index (1::2). Next we give names to the dataset columns and we add the Price (target) column to the DataFrame.

Then we plot an heatmap of the correlation matrix, showing also the numeric value of the correlation ($annot = True$)



```python
# There are cases of multicolinearity, we will drop a few columns
boston_df.drop(columns = ["INDUS", "NOX"], inplace = True)

# pairplot
sns.pairplot(boston_df)

# log the LSTAT Column
boston_df.LSTAT = np.log(boston_df.LSTAT)
```

Multicolinearity verifies when two or more independent variables are strongly correlated. This can lead to instability of the linear regression coefficients (making the model difficult to interpret). So we remove the columns (INDUS and NOX) from the DataFrame; with $inplace = True$ we make sure that the modifies are done on the original DataFrame.

Later, we do a pairplot, utilizing seaborn, that is a scatterplot grid who shows the relation between each numeric variables couple in the dataset.

It helps find linear relation between the variables, pattern or cluster of the data and outlier presence.

Finally, we do the logarithmic transformation, computing the natural logarithm of each value in the LSTAT column and changing the original value with this new ones.

We often do this to reduce the variance and make the distribution similar to a Gaussian, but also to make the data less skewed.

3.2 Data Splitting and Scaling

```python
# preview
features = boston_df.columns[0:11]
target = boston_df.columns[-1]

# X and y values
X = boston_df[features].values
y = boston_df[target].values

# split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=17)

print("The dimension of X_train is {}".format(X_train.shape))
print("The dimension of X_test is {}".format(X_test.shape))
# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In this code we:

1. select the first 11 columns of the DataFrame as independent variables and the last one as dependent variable
2. extract the values and assign it tu X and y
3. divide in train and test (70% trainig and 30% test), every execution will have the same division of the data ($random\_state = 17$)
4. print the dimension of the split
5. scale the features (centering the data with mean 0 and standard deviation 1). With *scaler.fit_transform(X_traini)* we compute mean and standard deviation only on the training set, transform the value of X_train based on this parameters. With *scaleer.transform(X_test)* we transform the value of X_test based on the mean and variance used in the training set

3.3 Linear and Ridge Regression Models

In this section we compare the linear regression model and the ridge regression one.

1. we fit the linear regression model with the training set
2. we value the of the $R^2 = 1 - \frac{RSS}{TSS}$ for the linear model, and print it out (if the score is too high means that the data are overfitted; with a low score the model doesn't capture the data well)
3. we fit the ridge regression with an $\alpha = 10$ (higher value of alpha correspond to an higher coefficients reduction)
4. after valuating the model we print the scores

```python
# Linear Regression Model
lr = LinearRegression()

# fitting
lr.fit(X_train, y_train)

# predict
#prediction = lr.predict(X_test)

# actual
actual = y_test

train_score_lr = lr.score(X_train, y_train)
test_score_lr = lr.score(X_test, y_test)

print("The train score for lr model is {}".format(train_score_lr))
print("The test score for lr model is {}".format(test_score_lr))


# Ridge Regression Model
ridgeReg = Ridge(alpha=10)

ridgeReg.fit(X_train,y_train)

train_score_ridge = ridgeReg.score(X_train, y_train)
test_score_ridge = ridgeReg.score(X_test, y_test)

print("\nRidge Model............................................\n")
print("The train score for ridge model is {}".format(train_score_ridge))
print("The test score for ridge model is {}".format(test_score_ridge))
```

Using an alpha value of 10, the evaluation of the model, the train, and test data indicate better performance on the ridge model than on the linear regression model.

We can also plot the coefficients for both the linear and ridge models.

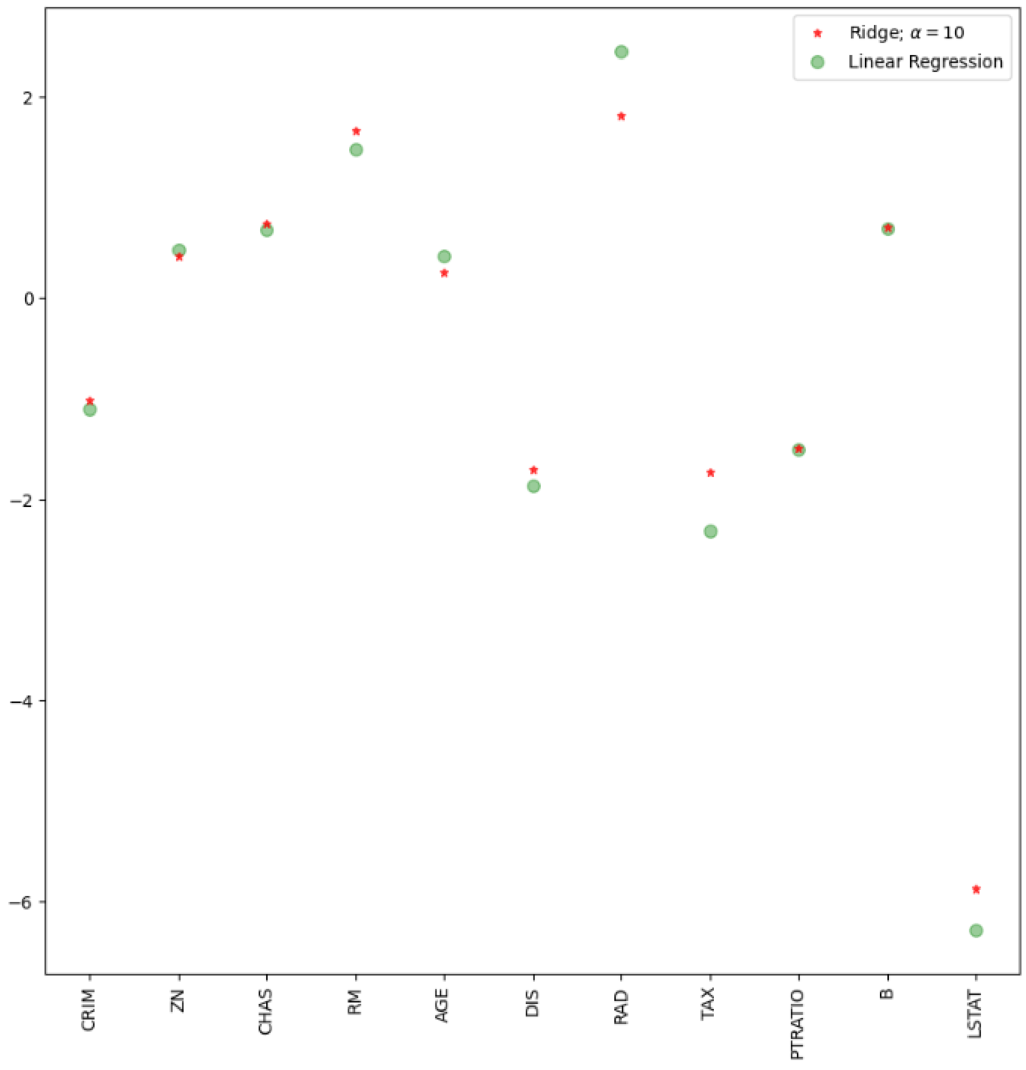
```python
plt.figure(figsize = (10, 10))
plt.plot(features,ridgeReg.coef_,alpha=0.7,linestyle='none',marker='*',
    markersize=5,color='red',label=r'Ridge; $\alpha = 10$',zorder=7)
plt.plot(features,lr.coef_,alpha=0.4,linestyle='none',marker='o',markersize
    =7,color='green',label='Linear Regression')
plt.xticks(rotation = 90)
plt.legend()
plt.show()
```

Here an explanation of the code:

```
plt.plot(
    features,              # Put the name of the feature on the x-axis
    ridgeReg.coef_,        # Coefficients of the ridge regression (y-axis)
    alpha=0.7,                  # Trasparency
    linestyle='none',           # No line
    marker='*',                 # Select the shape of the marker
    markersize=5,               # Dimension of the marker
    color='red',                # Color of the marker
    label=r'Ridge; $\alpha = 10$', # Label
    zorder=7               # Drawing order
                           # (higher number -> above the other elements)
)
```

```
plt.plot(
    features,         # Put the name of the feature on the x-axis
    lr.coef_,          # Coefficients of the linear regression (y-axis)
    alpha=0.4,         # Trasparency
    linestyle='none',  # No line
    marker='o',        # shape og the marker
    markersize=7,      # Dimension of the marker
    color='green',     # Color of the marker
    label='Linear Regression' # Label
)
```

with *plt.xticks*(*rotation* = 90) we rotate the names of the features in the x-axis to make them more readable. Finally we add the legend to distinguish the models and plot the final graph.



3.4 Lasso Regression

```
# Lasso regression model
print("\nLasso Model............................................\n")
lasso = Lasso(alpha = 10)
lasso.fit(X_train,y_train)
train_score_ls =lasso.score(X_train,y_train)
test_score_ls =lasso.score(X_test,y_test)

print("The train score for ls model is {}".format(train_score_ls))
print("The test score for ls model is {}".format(test_score_ls))
```

Now, we implement the lasso regression with an $\alpha = 10$, train it an value the score in the training and test set.

```
pd.Series(lasso.coef_, features).sort_values(ascending = True).plot(kind =
    "bar")
```

Whit this line we have in output a series containing the features name and predicted coefficients, sorted by ascending order (the negative feature will appear down, the positive ones up), We create a bar graph with this info (each bar represent a coefficient associated with a feature).

If some bars are absent the lasso eliminated that features (the coefficients are reduced to zero, there is no visible bar).

Features that has an higher bar (positive or negative) are more influent for the model than the bar near zero.

The bars over the x-axis represent positive effects on the target variable, bars under the x-axis represent negative effects.

3.5 Selecting Optimal Alpha values Using CV in sklearn

We may need to try out different alpha values to find the optimal constraint value. For this case, we can use the cross-validation model in the sklearn package.

```
# Using the linear CV model
from sklearn.linear_model import LassoCV

# Lasso Cross validation
lasso_cv = LassoCV(alphas = [0.0001, 0.001,0.01, 0.1, 1, 10], random_state
    =0).fit(X_train, y_train)


# score
print(lasso_cv.score(X_train, y_train))
print(lasso_cv.score(X_test, y_test))
```

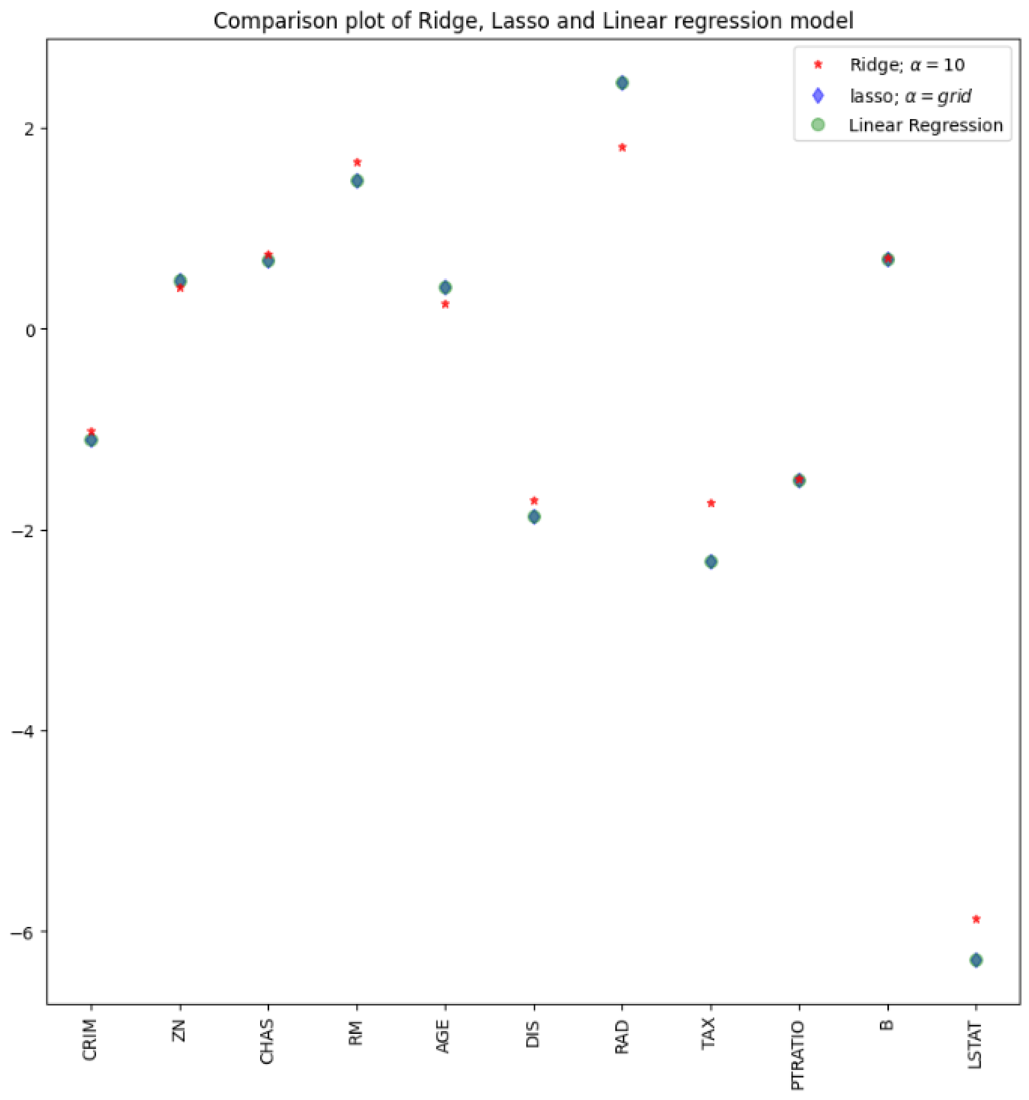
The model will be trained on different alpha values that I have specified in the LassoCV function (0.0001, 0.001,0.01, 0.1, 1, 10). We can observe a better performance of the model, removing the tedious effort of manually changing alpha values.

After testing all of the alpha values on different division of the data (by default is used the 5-fold cross-validation) it choose the value that best minimize the MSE.

We can compare the coefficients from the lasso model with the rest of the models (linear and ridge).

```
plt.figure(figsize = (10, 10))
# add plot for ridge regression
plt.plot(features,ridgeReg.coef_,alpha=0.7,linestyle='none',marker='*',
    markersize=5,color='red',label=r'Ridge; $\alpha = 10$',zorder=7)

# add plot for lasso regression
plt.plot(lasso_cv.coef_,alpha=0.5,linestyle='none',marker='d',markersize=6,
    color='blue',label=r'lasso; $\alpha = grid$')

# add plot for linear model
plt.plot(features,lr.coef_,alpha=0.4,linestyle='none',marker='o',markersize
    =7,color='green',label='Linear Regression')

# rotate axis
plt.xticks(rotation = 90)
plt.legend()
plt.title("Comparison plot of Ridge, Lasso and Linear regression model")
plt.show()
```

In output we have:



```python
# Using the linear CV model
from sklearn.linear_model import RidgeCV

# Lasso Cross validation
ridge_cv = RidgeCV(alphas = [0.0001, 0.001,0.01, 0.1, 1, 10]).fit(X_train,
    y_train)

# score
print("The train score for ridge model is {}".format(ridge_cv.score(X_train
    , y_train)))
print("The train score for ridge model is {}".format(ridge_cv.score(X_test,
     y_test)))
```

With this code we do the same as before but for the Ridge regression α instead of the lasso one.

```python
print("Best alpha:", ridge_cv.alpha_)
```

Using this line with can see the best value of alpha that's been choosen.

Chapter 4

Trees

In this session, we will build a spam classification by using an email dataset. Our goal is to develop optimal models to predict whether an email is spam or not spam based on word characteristics within each email. We have to perform the following steps:

4.1 Prepare the dataset

```python
import numpy as np
import pandas as pd
import math
import matplotlib
import matplotlib.pyplot as plt
import sklearn.metrics as metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score,roc_curve
%matplotlib inline
from tqdm import tqdm
from sklearn.model_selection import learning_curve

# mount google drive
from google.colab import drive
drive.mount('/content/drive')
```

We import useful packages, we have already seen most of them; tqdm is used for long loops.

After that we upload our dataset from google drive (mounted before).

```python
# import dataset
df =pd.read_csv("/content/drive/MyDrive/University/Data set/Lab3_dataset1.
    csv")
columns = ["Column_"+str(i+1) for i in range(df.shape[1]-1)] + ['Spam']
df.columns = columns
df.head()
```

We then renominate the column name with "Column_(number of the column)" for all the columns except the last one named "Spam".

The numbers of the columns are chosen from i+1 to N-1 where i goes from 0 to N; with N the total number of columns minus one.

With *df.shape*[1] we access at the second value of the tuple (total number of column in the DataFrame; the first value was the total number of rows).

We use the *range*(*df.shape*[1] − 1) because we don't want to consider the last column "Spam".

```
# Split data into train and test
np.random.seed(10)
indx = np.random.rand(len(df)) < 0.7
print(indx)
df_train = df[indx]
df_test = df[~indx]

# Split predictor and response columns
x_train, y_train = df_train.drop(['Spam'], axis=1), df_train['Spam']
x_test, y_test = df_test.drop(['Spam'], axis=1), df_test['Spam']

print(df_train.shape)
print(df_test.shape)

# Check Percentage of Spam in Train and Test Set
print("Percentage of Spam in Training Set :", str(100*y_train.sum()/len(
    y_train))+'%')
print("Percentage of Spam in Testing Set :",str(100*y_test.sum()/len(y_test
    ))+'%')
```

In this code:

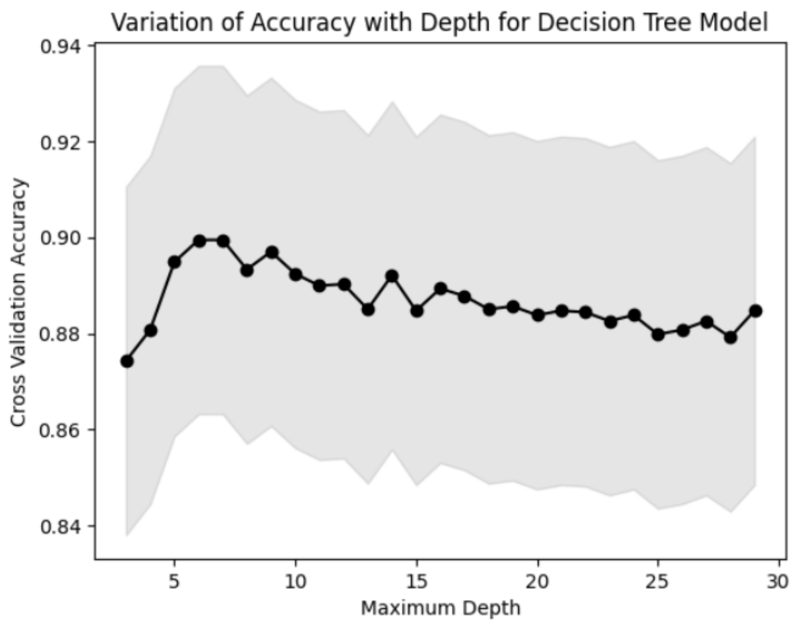
1. `np.random.seed(10)`: set a seed of the random generator to make sure the result are reproducible.

2. `np.random.rand(len(df)) < 0.7`: generate an array of random numbers evenly distributed between 0 and 1, with the same length as the number of rows in the DataFrame; then it compare each value of the array with 0.7. The result is a boolean array `indx` that is `True` if the row is included in the training set and `False` if it's included in the test set.

3.

4.2 Train the Decision Tree Model

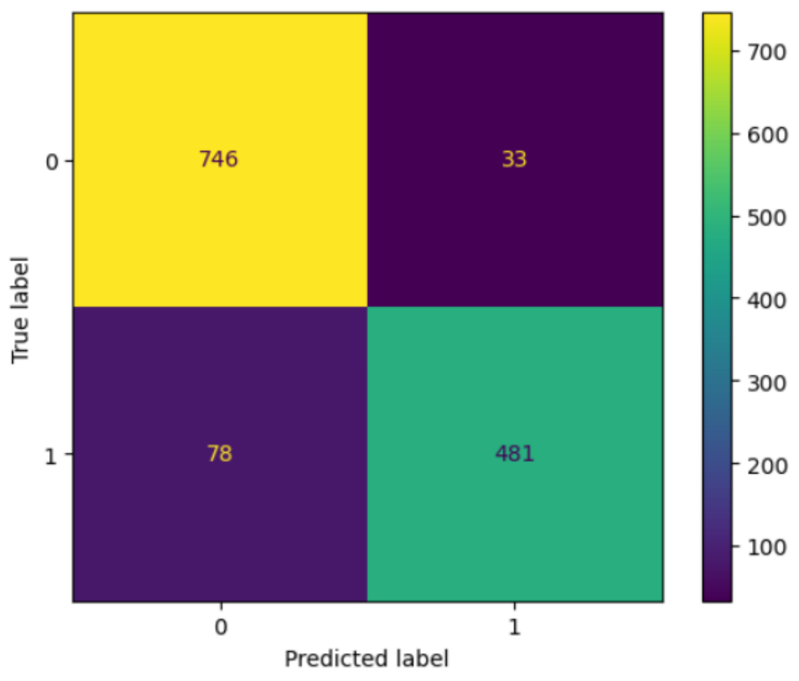
```
# Tuning of the parameter depth: find optimal depth of trees
depth= {}
tree_start, tree_end = 3, 30
for i in range(tree_start, tree_end):
    model = DecisionTreeClassifier(max_depth=i)
    scores = cross_val_score(estimator=model, X=x_train, y=y_train, cv=5,
        n_jobs=-1)
    depth[i] = scores.mean()

# Plot of results
print(depth)
lists = sorted(depth.items())
x, y = zip(*lists)
y_err = scores.std()
plt.ylabel("Cross Validation Accuracy")
plt.xlabel("Maximum Depth")
plt.title('Variation of Accuracy with Depth for Decision Tree Model')
plt.plot(x, y, 'k-', marker='o')
plt.fill_between(x, y - y_err, y + y_err, color='grey', alpha=0.2)
plt.show()
```



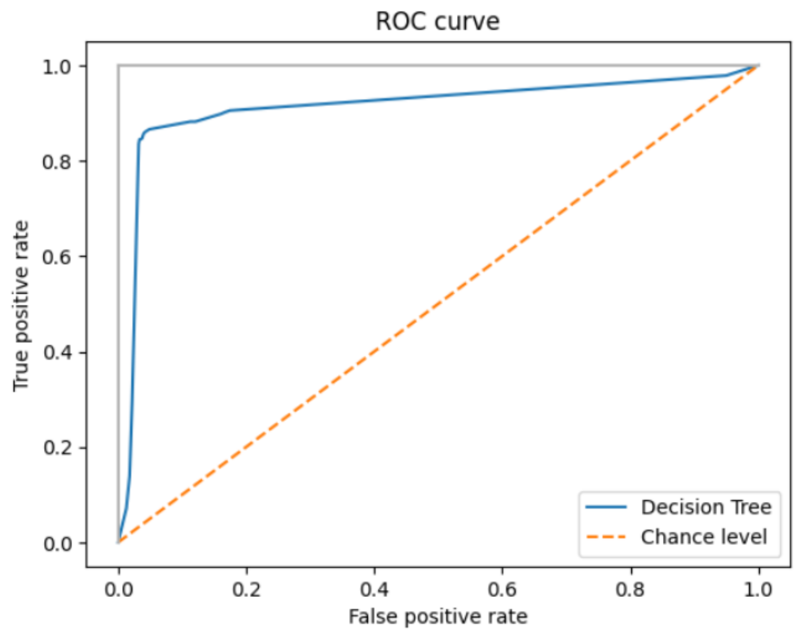
```python
# Make best depth a variable
best_depth = sorted(depth, key=depth.get, reverse=True)[0]
print("The best depth is:", best_depth)
```

```python
# Evalaute the performance choosing the best depth
model = DecisionTreeClassifier(max_depth=best_depth)
model.fit(x_train, y_train)

# Check Accuracy of Spam Detection in Train and Test Set
print("Accuracy, Training Set: {:.1%}".format(accuracy_score(y_train, model
    .predict(x_train))))
print("Accuracy, Testing Set: {:.1%}".format(accuracy_score(y_test, model.
    predict(x_test))))
```

```python
# Confusion Matrix
confusion_matrix = metrics.confusion_matrix(y_test, model.predict(x_test))

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
    confusion_matrix)

cm_display.plot()
plt.show()

pd.crosstab(y_test, model.predict(x_test), margins=True, rownames=['Actual'
    ], colnames=['Predicted'])
```



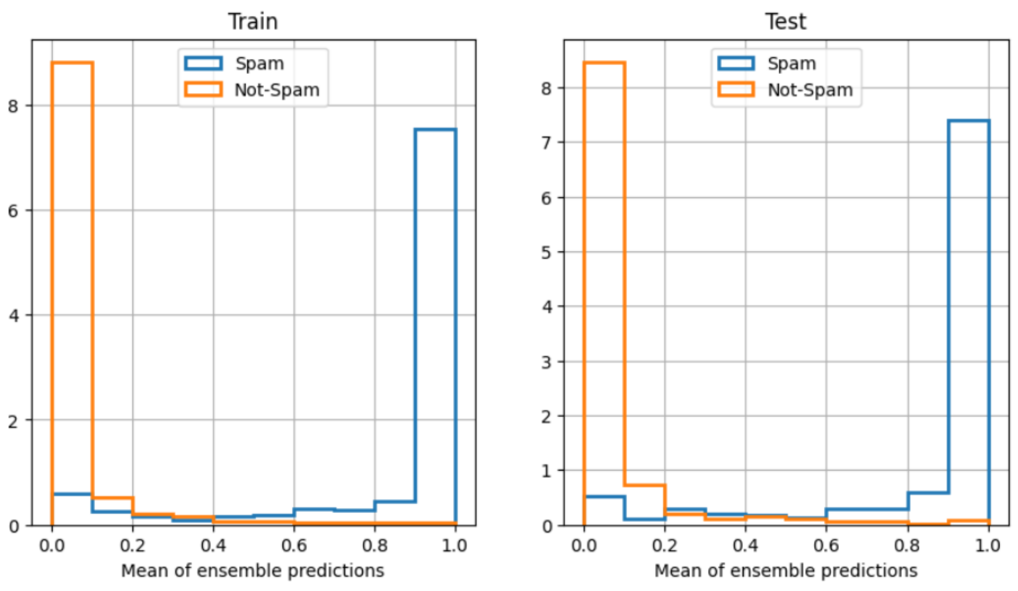
```
y_proba = model.predict_proba(x_test)[:,1]
print("Roc AUC:", roc_auc_score(y_test, model.predict_proba(x_test)[:,1],
    average='macro'))
fpr, tpr, thresholds = roc_curve(y_test, y_proba)
plt.plot(fpr, tpr, label='Decision Tree')
plt.plot([0, 1], ls="--",label='Chance level')
plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()
```



4.3 Fit the Bagging Model

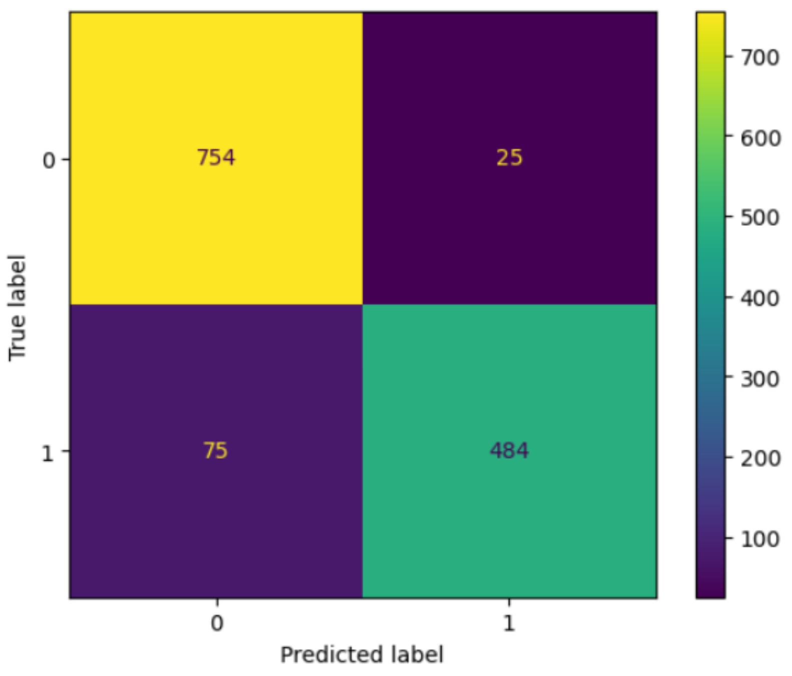
```python
# Creating model
np.random.seed(0)
model = DecisionTreeClassifier(max_depth=5)

# Initializing variables
n_trees = 100
predictions_train = np.zeros((df_train.shape[0], n_trees))
predictions_test = np.zeros((df_test.shape[0], n_trees))

# Bootstraping iterations
for i in range(n_trees):
    temp_sample = df_train.sample(frac=1, replace=True)
    response_variable = temp_sample['Spam']
    temp_sample = temp_sample.drop(['Spam'], axis=1)
    model.fit(temp_sample, response_variable)
    predictions_train[:,i] = model.predict(x_train)
    predictions_test[:,i] = model.predict(x_test)

# Make Predictions Dataframe
columns = ["Bootstrap-Model_"+str(i+1) for i in range(n_trees)]
predictions_train = pd.DataFrame(predictions_train, columns=columns)
predictions_test = pd.DataFrame(predictions_test, columns=columns)
print(predictions_train.shape)
print(predictions_test.shape)
```

```python
y_train = df_train['Spam'].values
y_test = df_test['Spam'].values

# n_trees
num_to_avg = 100

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
for (ax, label, predictions, y) in [
    (axs[0], 'Train', predictions_train, y_train),
    (axs[1], 'Test', predictions_test, y_test)
]:
    mean_predictions = predictions.iloc[:,:num_to_avg].mean(axis=1)
    mean_predictions[y == 1].hist(density=True, histtype='step', range
        =[0,1], label='Spam', lw=2, ax=ax)
    mean_predictions[y == 0].hist(density=True, histtype='step', range
        =[0,1], label='Not-Spam', lw=2, ax=ax)
    ax.legend(loc='upper center');
    ax.set_xlabel("Mean of ensemble predictions")
    ax.set_title(label)
```



```
# Function to ensemble the prediction of each bagged decision tree model
def get_prediction(df, count=-1):
    count = df.shape[1] if count==-1 else count
    temp = df.iloc[:,0:count]
    return np.mean(temp, axis=1)>0.5

# Check performance metrics of Spam Detection in Test Set
Accuracy = metrics.accuracy_score(y_test, get_prediction(predictions_test,
    count=-1))
Sensitivity = metrics.recall_score(y_test, get_prediction(predictions_test,
     count=-1))
Specificity = metrics.recall_score(y_test, get_prediction(predictions_test,
     count=-1),pos_label=0)
F1_score = metrics.f1_score(y_test, get_prediction(predictions_test, count
    =-1))

print({"Accuracy":Accuracy,"Sensitivity":Sensitivity,"Specificity":
    Specificity,"F1_score":F1_score})
```

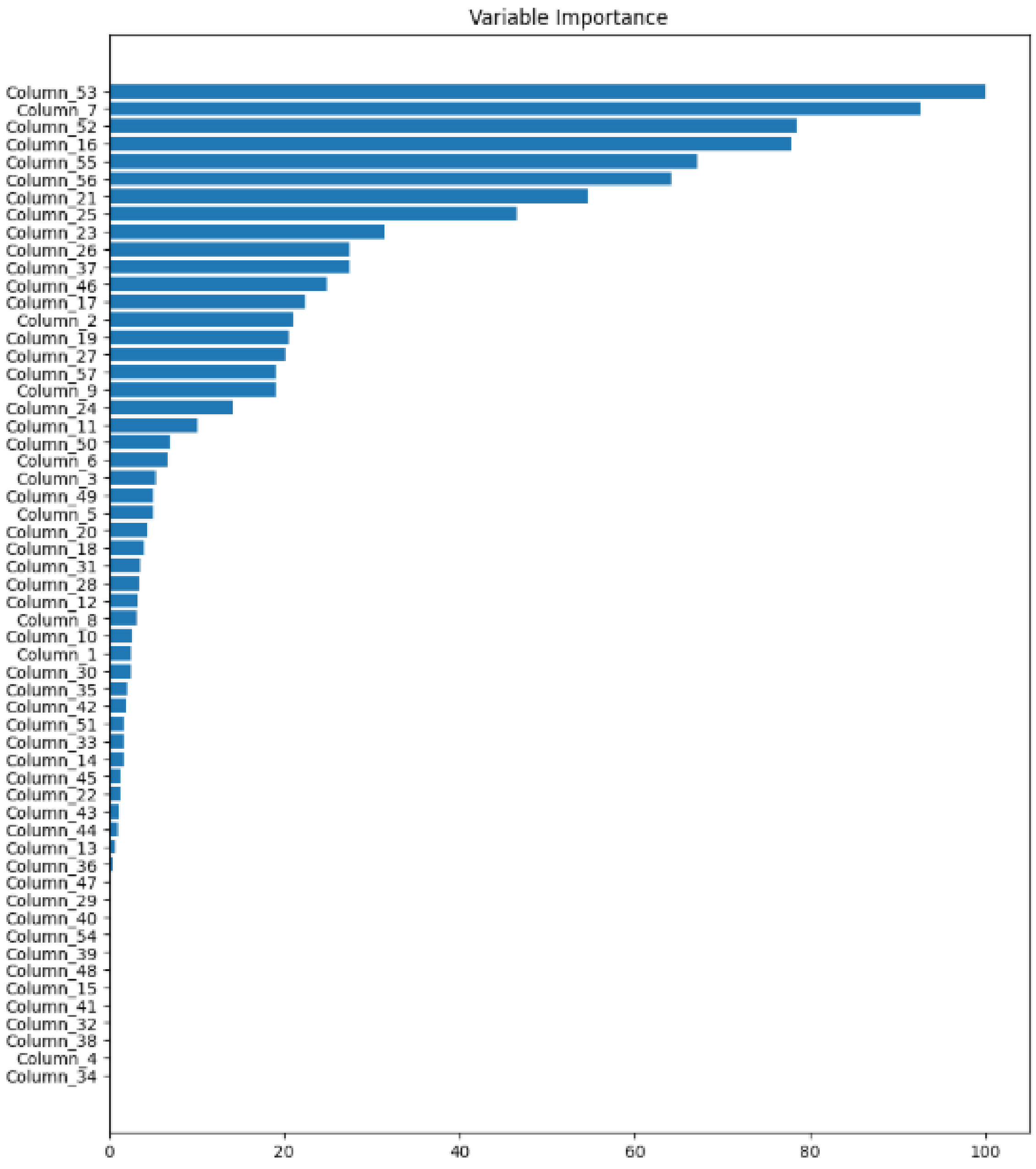
```
confusion_matrix = metrics.confusion_matrix(y_test, get_prediction(
    predictions_test, count=-1))

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
    confusion_matrix)

cm_display.plot()
plt.show()
```



4.4 Fit a Random Forest Model

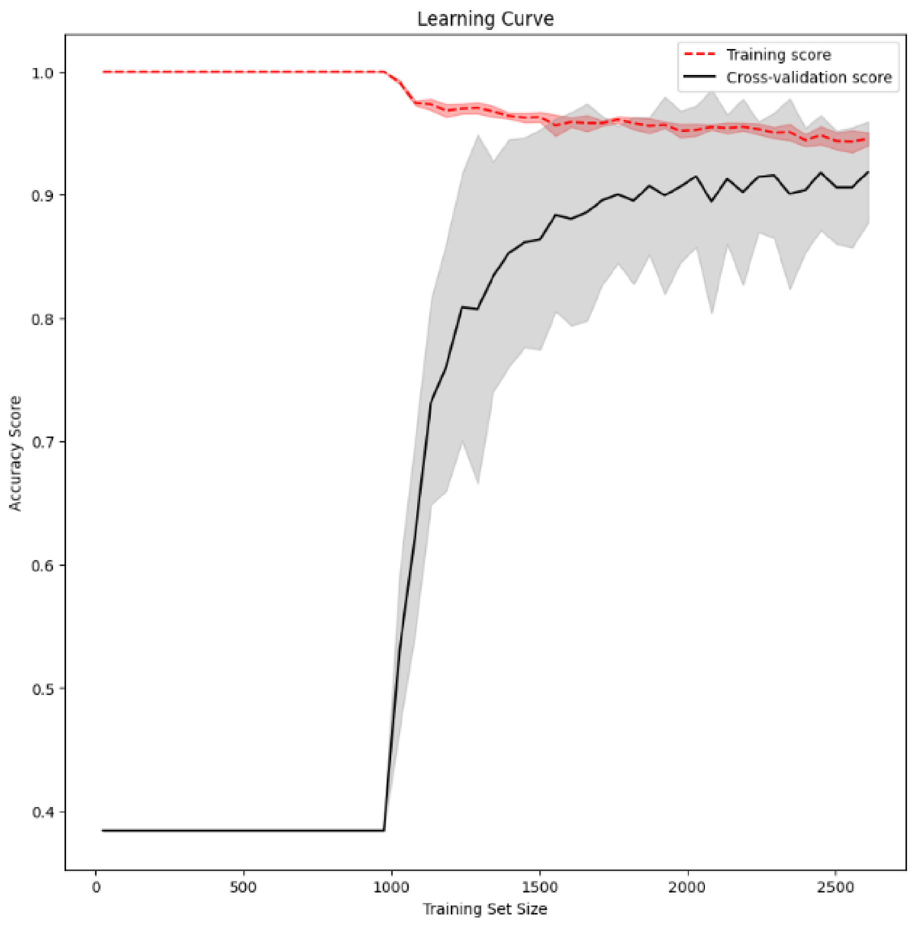
```python
# Training
model = RandomForestClassifier(n_estimators=int(math.sqrt(x_train.shape[1])
    ), max_depth=best_depth)
model.fit(x_train, y_train)

# Predict
y_pred_train = model.predict(x_train)
y_pred_test = model.predict(x_test)

# Performance metrics
train_score = accuracy_score(y_train, y_pred_train)*100
test_score = accuracy_score(y_test, y_pred_test)*100

print("Accuracy, Training Set :",str(train_score)+'%')
print("Accuracy, Testing Set :",str(test_score)+'%')
```

```python
# Top Features
feature_importance = model.feature_importances_
feature_importance = 100.0 * (feature_importance / feature_importance.max()
    )
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5

# Plot
plt.figure(figsize=(10,12))
plt.barh(pos, feature_importance[sorted_idx], align='center')
plt.yticks(pos, x_train.columns[sorted_idx])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
```



```python
# Plot Learning Curve for training
estimator = RandomForestClassifier(n_estimators=int(math.sqrt(x_train.shape
    [1])), max_depth=best_depth)
title = "Learning Curves (Random Forest)"

train_sizes, train_scores, test_scores = learning_curve(estimator, x_train,
     y_train, cv=5,scoring="accuracy", train_sizes=np.linspace(0.01, 1.0,
    50))


train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.subplots(1, figsize=(10,10))
plt.plot(train_sizes, train_mean, '--', color="red",  label="Training score
    ")
```

```
plt.plot(train_sizes, test_mean, color="black", label="Cross-validation
    score")

plt.fill_between(train_sizes, train_mean - train_std, train_mean +
    train_std, color="red", alpha=0.3)
plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std,
    color="gray",alpha=0.3)

plt.title("Learning Curve")
plt.xlabel("Training Set Size"), plt.ylabel("Accuracy Score"), plt.legend(
    loc="best")
plt.show()
```



Chapter 5

Multiclass Lab

In this session, we will build a 3-classes classification by using the ”iris flower” dataset. This is a multivariate data set introduced by Ronald Fisher in the 1936 as an example of discriminant analysis.

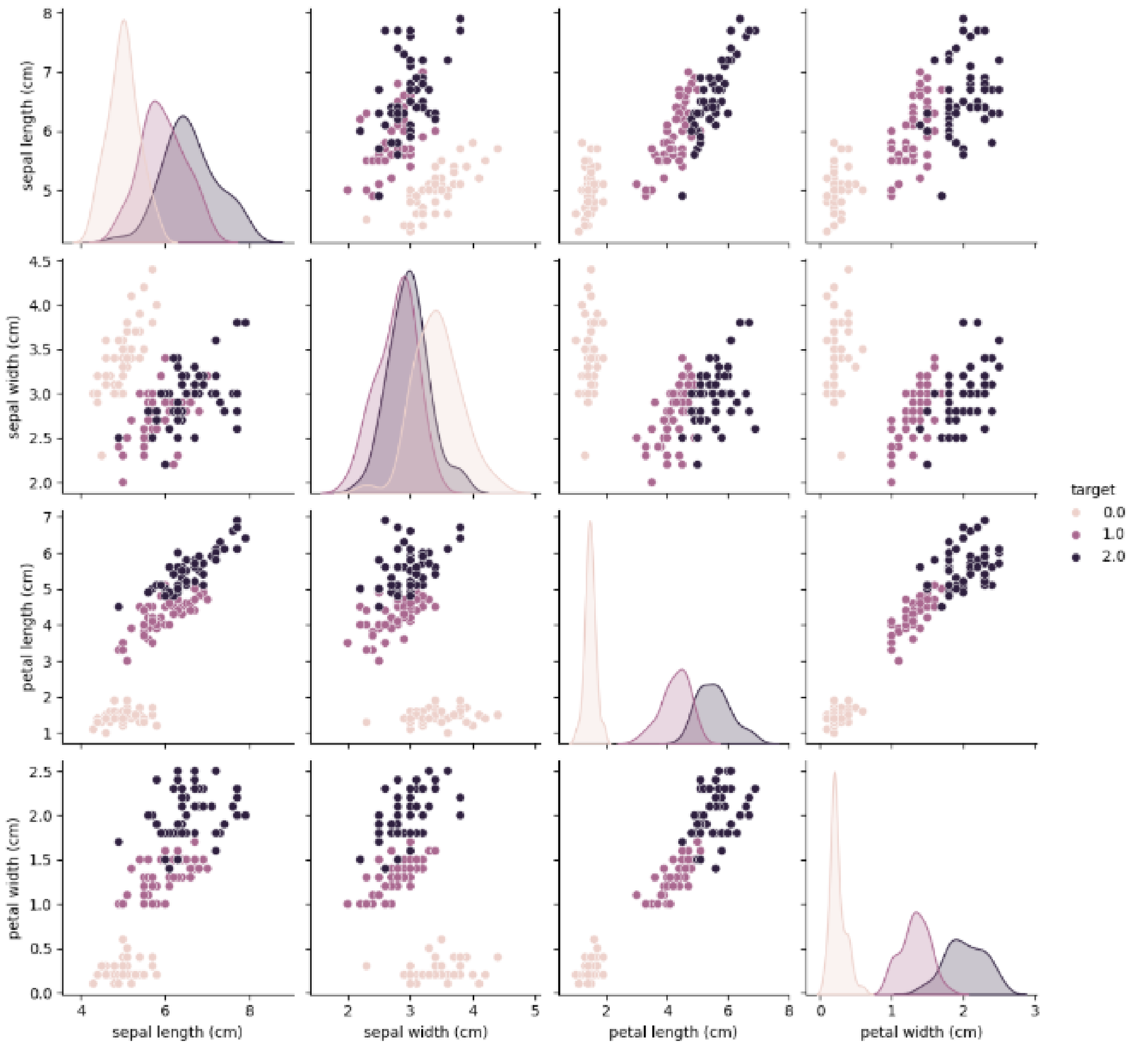
The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.

5.1 Prepare and Explore the dataset

```python
import numpy as np
import pandas as pd
import math
import matplotlib
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
import seaborn as sns
import sklearn.metrics as metrics
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import auc,roc_curve
from sklearn.model_selection import learning_curve
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import GridSearchCV
from mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
import itertools

%matplotlib inline
```

```python
from sklearn.datasets import load_iris

iris = load_iris()

X = iris.data
y = iris.target

df = pd.DataFrame(data= np.c_[iris['data'], iris['target']],
                     columns= iris['feature_names'] + ['target'])

```

```
sns.pairplot(df,hue='target')
```



```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
    test_size=0.25, random_state=1)

# normalize data
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print(X_train.shape)
print(X_test.shape)
```

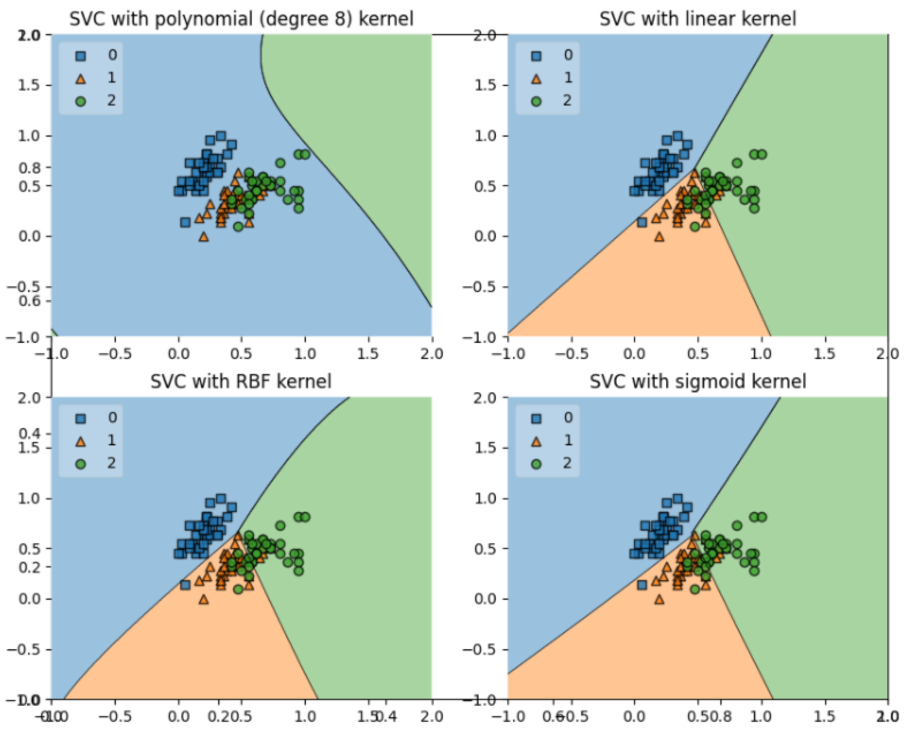
5.2 Define kernels to Transform the data to a higher dimension

```
kernels = ['Polynomial', 'RBF', 'Sigmoid','Linear']
def getClassifier(ktype):
    if ktype == 0:
        # Polynomial kernal
        return SVC(kernel='poly', degree=8, gamma="auto")
    elif ktype == 1:
```

```python
        # Radial Basis Function kernal
        return SVC(kernel='rbf', gamma="auto")
    elif ktype == 2:
        # Sigmoid kernal
        return SVC(kernel='sigmoid', gamma="auto")
    elif ktype == 3:
        # Linear kernal
        return SVC(kernel='linear', gamma="auto")
```

```python
X_train1, X_test1, y_train1, y_test1 = train_test_split(X_train, y_train,
    test_size=0.15)
for i in range(4):
    svclassifier = getClassifier(i)
    svclassifier.fit(X_train, y_train)
    y_pred = svclassifier.predict(X_test)
    print("Evaluation:", kernels[i], "kernel")
    print(classification_report(y_test,y_pred))
```

```python
clf1 = SVC(kernel='poly', degree=8, gamma="auto")
clf2 = SVC(kernel='linear', gamma="auto")
clf3 = SVC(kernel='rbf', gamma="auto")
clf4=SVC(kernel='sigmoid', gamma="auto")

plt.subplots(figsize=(10, 8))
count = 1

# title for the plots
titles = (
    "SVC with polynomial (degree 8) kernel",
    "SVC with linear kernel",
    "SVC with RBF kernel",
    "SVC with sigmoid kernel",
)

for clf, lab in zip([clf1, clf2, clf3, clf4],titles):

    clf.fit(X_train[:, :2], y_train)
    plt.subplot(2,2,count)
    fig = plot_decision_regions(X=X_train[:, :2], y=y_train, clf=clf,
        legend=2)
    plt.title(lab)
    count+=1

plt.show()
```

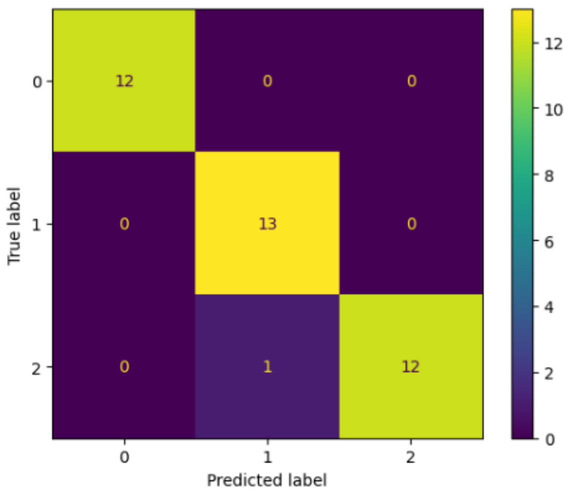


5.3 Create a GridSearchCV object and fit it to the training data

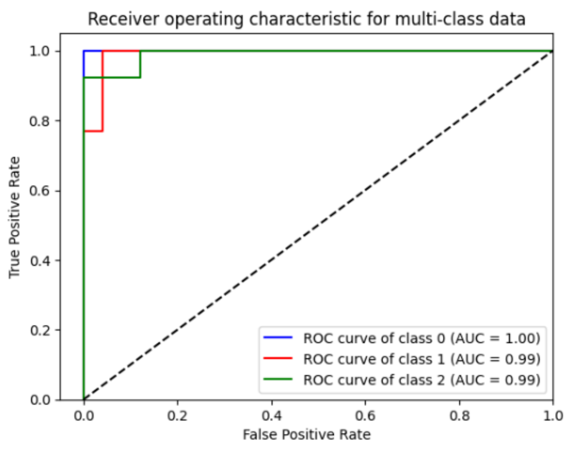
```
C_range = np.logspace(-2, 10, 13)
gamma_range = np.logspace(-9, 3, 13)
kernels = ['linear','rbf', 'poly', 'sigmoid']
param_grid = dict(gamma=gamma_range, C=C_range, kernel=kernels)
cv = StratifiedShuffleSplit(n_splits=3, test_size=0.2, random_state=42)
grid = GridSearchCV(SVC(probability = True), param_grid=param_grid, cv=cv,
    verbose=3, return_train_score=True)
grid.fit(X_train, y_train)

print(
    "The best parameters are %s with a score of %0.2f"
    % (grid.best_params_, grid.best_score_)
)
```

5.4 Predict the outcomes for the test set

```
grid_predictions = grid.predict(X_test)

# Confusion Matrix
confusion_matrix = metrics.confusion_matrix(y_test, grid_predictions)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
    confusion_matrix)

cm_display.plot()
plt.show()
```



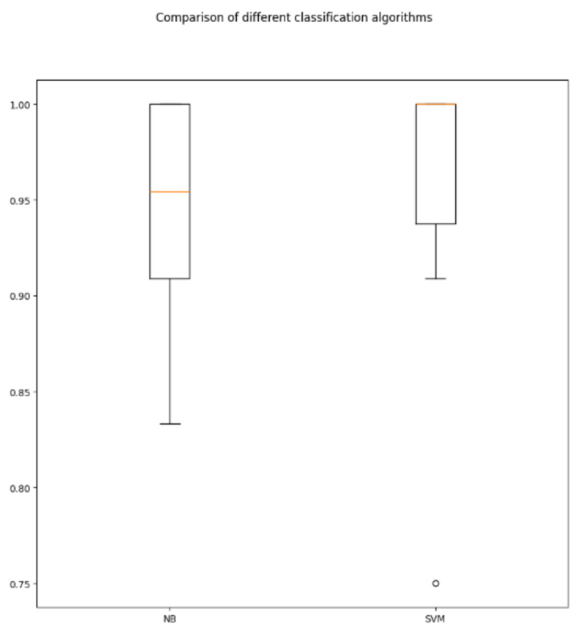
```
y_score = grid.predict_proba(X_test)
from sklearn.preprocessing import label_binarize
from itertools import cycle
y_test1 = label_binarize(y_test, classes=[0, 1, 2])
n_classes = y_test1.shape[1]


fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test1[:, i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
colors = cycle(['blue', 'red', 'green'])
for i, color in zip(range(n_classes), colors):
    plt.plot(fpr[i], tpr[i], color=color, lw=1.5,
             label='ROC curve of class {0} (AUC = {1:0.2f})'
             ''.format(i, roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--', lw=1.5)
plt.xlim([-0.05, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic for multi-class data')
plt.legend(loc="lower right")
plt.show()
```



5.5 Compare different ML models

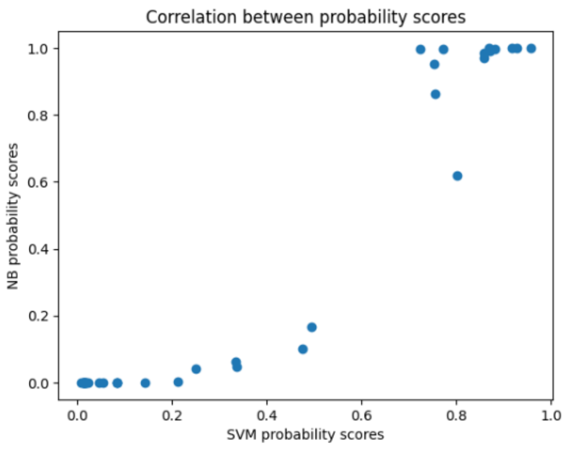
```
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB

models = []
#models.append(('LR', LogisticRegression()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(kernel='rbf')))

seed=10

results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, shuffle=True, random_state=
        seed)
    cv_results = model_selection.cross_val_score(model, X_train, y_train,
        cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
fig = plt.figure(figsize=(10,10))
fig.suptitle('Comparison of different classification algorithms')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```



```
model = GaussianNB()

model.fit(X_train,y_train)
y_score_NB = model.predict_proba(X_test)
print(y_score_NB)
plt.scatter(y_score[:,1], y_score_NB[:,1])
```

```python
plt.xlabel('SVM probability scores')
plt.ylabel('NB probability scores')
plt.title('Correlation between probability scores')

plt.show()
```



Chapter 6

Neural Networks

This Colab uses the California Housing Dataset:
https://www.kaggle.com/datasets/camnugent/california-housing-prices

6.1 Classification

6.1.1 Import relevant modules, Load and Process the dataset

```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from matplotlib import pyplot as plt
import seaborn as sns

# mount google drive
from google.colab import drive
drive.mount('/content/drive')
```

This exercise uses the California Housing Dataset. The following code cell loads the separate .csv files and creates the following two pandas DataFrames:
df_train, which contains the training set df_test, which contains the test set

```python
df_train = pd.read_csv("/content/drive/MyDrive/Lab_5/Lab5_dataset_train.csv
    ")
df_test = pd.read_csv("/content/drive/MyDrive/Lab_5//Lab5_dataset_test.csv"
    )
```

The objective of this example is to predict median_house_value according to the following three features:
median_income X housing_median_age (a feature cross) total_rooms population We want to treat this problem as a classification task. Since median_house_value is a continuous variable, we need to make it binary.

```python
df_train.head(5)
```

Feature cross is a machine learning technique for generating new features by combining existing ones. This approach helps to capture interactions between features that might otherwise go unnoticed by a model.

```python
# Create a temporary dataframe to hold data
data = pd.DataFrame()

# Discretize the features median_income
income_boundaries = [0, 5, 10, 15, 100]
```

```python
income_labels = ["low","medium","high","really high"];
data["median_income"] = pd.cut(df_train['median_income'],
                               bins=income_boundaries,
                               labels=income_labels)

# Discretize the features housing_median_age
age_boundaries = [0, 10, 30, 100];
age_labels = ["new","modern","old"];
data["housing_median_age"] = pd.cut(df_train['housing_median_age'],
                                    bins=age_boundaries,
                                    labels=age_labels)

# Put features together as strings
data["feature_cross"] = data['median_income'].astype(str) + "_" + data['
    housing_median_age'].astype(str)

# Insert the new feature in the dataset after mapping to integers
df_train['median_income_x_housing_median_age'] = data["feature_cross"].
    astype('category').cat.codes
```

Repeat the feature cross computation for the test set.

```python
# Create a temporary dataframe to hold data
data = pd.DataFrame()

# Discretize the features median_income
data["median_income"] = pd.cut(df_test['median_income'],
                               bins=income_boundaries,
                               labels=income_labels)

# Discretize the features housing_median_age
data["housing_median_age"] = pd.cut(df_test['housing_median_age'],
                                    bins=age_boundaries,
                                    labels=age_labels)

# Put features together as strings
data["feature_cross"] = data['median_income'].astype(str) + "_" + data['
    housing_median_age'].astype(str)

# Insert the new feature in the dataset after mapping to integers
df_test['median_income_x_housing_median_age'] = data["feature_cross"].
    astype('category').cat.codes
```

Check the new features

```python
df_train.head(5)
```

Select features and apply standardization (z-score).

```python
# List of selected features:
selected_features = ["median_house_value", "
    median_income_x_housing_median_age", "total_rooms", "population"];

# Filter columns:
df_train = df_train[selected_features];
df_test = df_test[selected_features];

# shuffle the examples
df_train = df_train.reindex(np.random.permutation(df_train.index))

# Calculate the Z-scores of each column in the training set:
df_train_mean = df_train.mean()
df_train_std = df_train.std()
df_train_norm = (df_train - df_train_mean)/df_train_std

```

```python
# Calculate the Z-scores of each column in the test set.
df_test_norm = (df_test - df_train_mean)/df_train_std

print("Normalized the values.")
```

6.1.2 Create binary labels

```python
# Create Binary label
#75th percentile of median house value
print(df_train["median_house_value"].quantile(q=0.75))
threshold = 265000.0

df_train_norm["median_house_value_is_high"] = (df_train["median_house_value
    "] > threshold).astype(float)
df_test_norm["median_house_value_is_high"] = (df_test["median_house_value"]
     > threshold).astype(float)

# Drop hold column
df_train_norm = df_train_norm.drop("median_house_value", axis = 1)
df_test_norm = df_test_norm.drop("median_house_value", axis = 1)

# Show new label
df_train_norm["median_house_value_is_high"].head(5)
```

6.1.3 Build the NN

```python
# Define the plotting function
def plot_curve(epochs, hist, list_of_metrics):
    """Plot a curve of one or more classification metrics vs epoch"""
    plt.figure()
    plt.xlabel("Epoch")
    plt.ylabel("Value")

    for m in list_of_metrics:
        x = hist[m]
        plt.plot(epochs[1:], x[1:], label=m)

    plt.legend()
```

```python
def create_model(my_learning_rate, n_inputs, my_metrics):
  """Create and compile a simple neural network model."""
  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Add one linear layer
  model.add(tf.keras.layers.Dense(units=1, input_shape=(n_inputs,),
      activation=tf.sigmoid),)

  # Construct the layers into a model that TensorFlow can execute.
  model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=
      my_learning_rate),
                loss=tf.keras.losses.BinaryCrossentropy(),
                metrics=my_metrics)

  return model


def train_model(model, dataset, epochs, batch_size, label_name):
  """Feed a dataset into the model in order to train it."""

  # Split the dataset into features and label.
  label = np.array(dataset[label_name])
  dataset = dataset.drop(label_name, axis = 1);
  input_data = np.array(dataset);

  # Store your model.fit results in a 'history' variable.
  history = model.fit(x=input_data, y=label, batch_size=batch_size,
                      epochs=epochs, shuffle=True)

  # Get details that will be useful for plotting the loss curve.
  epochs = history.epoch
  # Convert the history.history dictionary to a pandas dataframe.
  hist = pd.DataFrame(history.history)


  return epochs, hist

print("Defined the create_model and train_model functions.")
```

```
# Utility function to plot the model architecture
def plot_my_model(model):
  return tf.keras.utils.plot_model(my_model, show_shapes=True,
     show_layer_activations=True, show_layer_names=True, dpi = 100)
```

Run the following code cell to invoke the functions defined in the preceding two code cells. (Ignore the warning messages.)

Note: Depending on the version of TensorFlow, running this cell might generate WARNING messages. Please ignore these warnings.

```
# Hyperparameters
n_inputs = 3
learning_rate = 0.001
epochs = 20
batch_size = 100
label_name = "median_house_value_is_high"
classification_threshold = 0.50


# Establish the metrics the model will measure
METRICS = [
           tf.keras.metrics.BinaryAccuracy(name='accuracy', threshold=
               classification_threshold),
           tf.keras.metrics.Precision(thresholds=classification_threshold,
               name='precision'),
           tf.keras.metrics.Recall(thresholds=classification_threshold,
               name='recall'),
          ]

my_model = create_model(learning_rate, n_inputs, METRICS)

epochs, hist = train_model(my_model, df_train_norm, epochs, batch_size,
    label_name)


list_of_metrics_to_plot = ['accuracy','precision','recall']
plot_curve(epochs, hist, list_of_metrics_to_plot)

plot_my_model(my_model)
```

6.1.4 Evaluate the model

```python
# Prepare data
test_label = np.array(df_test_norm[label_name])
test_input_data = df_test_norm.drop(label_name, axis = 1);
test_input_data = np.array(test_input_data);

print("\n Evaluate the NN model against the test set:")
my_model.evaluate(x = test_input_data, y = test_label, batch_size=
    batch_size)
```

6.2 Regression

6.2.1 Import relevant modules, Load and Process the dataset

```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from matplotlib import pyplot as plt
import seaborn as sns

# mount google drive
from google.colab import drive
drive.mount('/content/drive')
```

```python
df_train =pd.read_csv("content/drive/Il mio Drive/Data set/
    Lab5_dataset_train.csv")
df_test =pd.read_csv("/content/drive/MyDrive/Lab5_dataset_test.csv")
```

The objective of this example is to predict median_house_value according to the following three features:

median_income X housing_median_age (a feature cross) total_rooms population

```python
df_train.head(5)
df_train.shape
```

Feature cross is a machine learning technique for generating new features by combining existing ones. This approach helps to capture interactions between features that might otherwise go unnoticed by a model.

```python
# Create a temporary dataframe to hold data
data = pd.DataFrame()

# Discretize the features median_income
income_boundaries = [0, 5, 10, 15, 100]
income_labels = ["low","medium","high","really high"]
data["median_income"] = pd.cut(df_train['median_income'],
                               bins=income_boundaries,
                               labels=income_labels)

# Discretize the features housing_median_age
age_boundaries = [0, 10, 30, 100];
age_labels = ["new","modern","old"];
data["housing_median_age"] = pd.cut(df_train['housing_median_age'],
                                    bins=age_boundaries,
                                    labels=age_labels)

# Put features together as strings
data["feature_cross"] = data['median_income'].astype(str) + "_" + data['
    housing_median_age'].astype(str)

```

```python
# Insert the new feature in the dataset after mapping to integers
df_train['median_income_x_housing_median_age'] = data["feature_cross"].
    astype('category').cat.codes
```

Repeat the feature cross computation for the test set.

```python
# Create a temporary dataframe to hold data
data = pd.DataFrame()

# Discretize the features median_income
data["median_income"] = pd.cut(df_test['median_income'],
                               bins=income_boundaries,
                               labels=income_labels)

# Discretize the features housing_median_age
data["housing_median_age"] = pd.cut(df_test['housing_median_age'],
                                    bins=age_boundaries,
                                    labels=age_labels)

# Put features together as strings
data["feature_cross"] = data['median_income'].astype(str) + "_" + data['
    housing_median_age'].astype(str)

# Insert the new feature in the dataset after mapping to integers
df_test['median_income_x_housing_median_age'] = data["feature_cross"].
    astype('category').cat.codes
```

Check the new feature.

```python
df_train.head(5)
```

Select features and apply standardization (z-score).

```python
# List of selected features:
selected_features = ["median_house_value", "
    median_income_x_housing_median_age", "total_rooms", "population"];

# Filter columns:
df_train = df_train[selected_features];
df_test = df_test[selected_features];

# shuffle the examples
df_train = df_train.reindex(np.random.permutation(df_train.index))

# Calculate the Z-scores of each column in the training set:
df_train_mean = df_train.mean()
df_train_std = df_train.std()
df_train_norm = (df_train - df_train_mean)/df_train_std

# Calculate the Z-scores of each column in the test set.
df_test_norm = (df_test - df_train_mean)/df_train_std

print("Normalized the values.")
```

6.2.2 Build a Linear Regression Model as baseline

Before creating a deep neural net, find a baseline loss by running a simple linear regression model that uses the feature layer you just created.

```python
def plot_the_loss_curve(epochs, mse):
  """Plot a curve of loss vs. epoch."""

  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Mean Squared Error")

  plt.plot(epochs, mse, label="Loss")
  plt.legend()
  plt.ylim([mse.min()*0.95, mse.max() * 1.03])
  plt.show()

print("Defined the plot_the_loss_curve function.")
```

```python
def create_model(my_learning_rate, n_inputs):
  """Create and compile a simple linear regression model."""
  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Add one linear layer to the model to yield a simple linear regressor.
  model.add(tf.keras.layers.Dense(units=1, input_shape=(n_inputs,), name =
      "neuron"))

  # Construct the layers into a model that TensorFlow can execute.
  model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=
      my_learning_rate),
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.MeanSquaredError()])

  return model


def train_model(model, dataset, epochs, batch_size, label_name):
  """Feed a dataset into the model in order to train it."""

  # Split the dataset into features and label.
  label = np.array(dataset[label_name])
  dataset = dataset.drop(label_name, axis = 1);
  input_data = np.array(dataset);

  # Store your model.fit results in a 'history' variable.
  history = model.fit(x=input_data, y=label, batch_size=batch_size,
                      epochs=epochs, shuffle=True)

  # Get details that will be useful for plotting the loss curve.
  epochs = history.epoch
  # Convert the history.history dictionary to a pandas dataframe.
  hist = pd.DataFrame(history.history)
  mse = hist["mean_squared_error"]

  return epochs, mse

print("Defined the create_model and train_model functions.")
```

6.2.3 Define a plotter function

Define an utility function to plot the architecture of the model.

```python
def plot_my_model(model):
  return tf.keras.utils.plot_model(my_model, show_shapes=True,
      show_layer_activations=True, show_layer_names=True, dpi = 100)
```

6.2.4 Train the linear model

Run the following code cell to invoke the functions defined in the preceding two code cells. (Ignore the warning messages.)

Note: Depending on the version of TensorFlow, running this cell might generate WARNING messages. Please ignore these warnings.

```
# The following variables are the hyperparameters.
n_inputs = 3
learning_rate = 0.01
epochs = 15
batch_size = 1000
label_name = "median_house_value"

# Establish the model's topography.
my_model = create_model(learning_rate, n_inputs)

# Train the model on the normalized training set.
epochs, mse = train_model(my_model, df_train_norm, epochs, batch_size,
    label_name)
plot_the_loss_curve(epochs, mse)

# isolate the label
test_label = np.array(df_test_norm[label_name])
test_input_data = df_test_norm.drop(label_name, axis = 1);
test_input_data = np.array(test_input_data);

print("\n Evaluate the linear regression model against the test set:")
my_model.evaluate(x = test_input_data, y = test_label, batch_size=
    batch_size)

plot_my_model(my_model)
```

6.2.5 Define a Deep Neural Net model

The create_model function defines the topography of the deep neural net, specifying the following:

The number of layers in the deep neural net. The number of nodes in each layer. The create_model function also defines the activation function of each layer.

```
def create_model(my_learning_rate, n_inputs):

  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Describe the topography of the model by calling the tf.keras.layers.
    Dense
  # method once for each layer. We've specified the following arguments:
  #   * units specifies the number of nodes in this layer.
  #   * activation specifies the activation function (Rectified Linear Unit
    ).
  #   * name is just a string that can be useful when debugging.

  # Define the first hidden layer with 20 nodes.
  model.add(tf.keras.layers.Dense(units=20,
                                  input_shape = (n_inputs,),
                                  activation='relu',
                                  name='Hidden1'))

  # Define the second hidden layer with 12 nodes.
  model.add(tf.keras.layers.Dense(units=12,
                                  activation='relu',
                                                name='Hidden2'))

   # Define the output layer.
   model.add(tf.keras.layers.Dense(units=1,
                                                name='Output'))

   model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=
       my_learning_rate),
                 loss="mean_squared_error",
                 metrics=[tf.keras.metrics.MeanSquaredError()])

   return model
```

6.2.6 Define a training function

The tf.keras.Model.fit method performs the actual training. The x parameter of the fit method is very flexible, enabling you to pass feature data in a variety of ways. The following implementation passes a Python numpy array in which.

Note: Numpy arrays don't have name of columns, they are plain matrices. When working with this method, make sure to always pass to the model an array which came from a dataframe with a constant order of features.

```
def train_model(model, dataset, epochs, label_name,
                batch_size=None):


   # Split the dataset into features and label.
   label = np.array(dataset[label_name])
   dataset = dataset.drop(label_name, axis = 1);
   input_data = np.array(dataset);

   # Train the model
   history = model.fit(x=input_data, y=label, batch_size=batch_size,
                       epochs=epochs, shuffle=True)

   # The list of epochs is stored separately from the rest of history.
   epochs = history.epoch

   # To track the progression of training, gather a snapshot
   # of the model's mean squared error at each epoch.
   hist = pd.DataFrame(history.history)
   mse = hist["mean_squared_error"]

   return epochs, mse
```

6.2.7 Call the functions to build and train a deep neural net

```
# The following variables are the hyperparameters.
learning_rate = 0.01
epochs = 20
batch_size = 1000
n_inputs = 3

# Specify the label
label_name = "median_house_value"

# Establish the model's topography.
my_model = create_model(learning_rate, n_inputs)

# Train the model on the normalized training set. We're passing the entire
# normalized training set, but the model will only use the features
# defined by the feature_layer.
epochs, mse = train_model(my_model, df_train_norm, epochs,
                          label_name, batch_size)
plot_the_loss_curve(epochs, mse)

# After building a model against the training set, test that model
# against the test set.
test_label = np.array(df_test_norm[label_name])
test_input_data = df_test_norm.drop(label_name, axis = 1);
test_input_data = np.array(test_input_data);

print("\n Evaluate the new model against the test set:")
my_model.evaluate(x = test_input_data, y = test_label, batch_size=
    batch_size)

plot_my_model(my_model)
```

6.3 Exercise

6.3.1 Compare the two models

How did the deep neural net perform against the baseline linear regression model?

6.3.2 Optimize the deep neural network's topography

Experiment with the number of layers of the deep neural network and the number of nodes in each layer. Aim to achieve both of the following goals:

Lower the loss against the test set. Minimize the overall number of nodes in the deep neural net. The two goals may be in conflict.

6.3.3 Regularize the deep neural network

Notice that the model's loss against the test set is much higher than the loss against the training set. In other words, the deep neural network is overfitting to the data in the training set. To reduce overfitting, regularize the model. We can use:

L1 regularization L2 regularization Dropout regularization Your task is to experiment with one or more regularization mechanisms to bring the test loss closer to the training loss (while still keeping test loss relatively low).

Note: When you add a regularization function to a model, you might need to tweak other hyperparameters.

Implementing L1 or L2 regularization

To use L1 or L2 regularization on a hidden layer, specify the kernel_regularizer argument to tf.keras.layers.Dense. Assign one of the following methods to this argument:

tf.keras.regularizers.l1 for L1 regularization tf.keras.regularizers.l2 for L2 regularization Each of the preceding methods takes an l parameter, which adjusts the regularization rate. Assign a decimal value between 0 and 1.0 to l; the higher the decimal, the greater the regularization. For example, the following applies L2 regularization at a strength of 0.01.

```
model.add(tf.keras.layers.Dense(units=20,
                                activation='relu',
                                kernel_regularizer=tf.keras.regularizers.l2(l=0.01),
                                name='Hidden1'))
```

Implementing Dropout regolarization

You implement dropout regularization as a separate layer in the topography. For example, the following code demonstrates how to add a dropout regularization layer between the first hidden layer and the second hidden layer:

```
model.add(tf.keras.layers.Dense( *define first hidden layer*)

model.add(tf.keras.layers.Dropout(rate=0.25))

model.add(tf.keras.layers.Dense( *define second hidden layer*)
```

The rate parameter to tf.keras.layers.Dropout specifies the fraction of nodes that the model should drop out during training.

6.3.4 Possible Solution

```
#@title Possible solution

# The following "solution" uses L2 regularization to bring training loss
# and test loss closer to each other. Many, many other solutions are
    possible.


def create_model(my_learning_rate, n_inputs):
  """Create and compile a simple linear regression model."""

  # Discard any pre-existing version of the model.
  model = None

  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Describe the topography of the model.

  # Implement L2 regularization in the first hidden layer.
  model.add(tf.keras.layers.Dense(units=20,
                                  input_shape = (n_inputs,),
                                  activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.
                                      l2(0.04),
                                  name='Hidden1'))

  # Implement L2 regularization in the second hidden layer.
  model.add(tf.keras.layers.Dense(units=12,
                                  activation='relu',
                                  kernel_regularizer=tf.keras.regularizers.
                                      l2(0.04),
                                  name='Hidden2'))

  # Define the output layer.
  model.add(tf.keras.layers.Dense(units=1,
                                  name='Output'))

  model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=
      my_learning_rate),
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.MeanSquaredError()])

  return model

```

```python
# Call the new create_model function and the other (unchanged) functions.

# The following variables are the hyperparameters.
n_inputs = 3
learning_rate = 0.007
epochs = 140
batch_size = 1000

label_name = "median_house_value"

# Establish the model's topography.
my_model = create_model(learning_rate, n_inputs)

# Train the model on the normalized training set.
epochs, mse = train_model(my_model, df_train_norm, epochs,
                          label_name, batch_size)
plot_the_loss_curve(epochs, mse)

# Prepare test data
test_label = np.array(df_test_norm[label_name])
test_input_data = df_test_norm.drop(label_name, axis = 1);
test_input_data = np.array(test_input_data);

print("\n Evaluate the new model against the test set:")
my_model.evaluate(x = test_input_data, y = test_label, batch_size=
    batch_size)

plot_my_model(my_model)
```

Chapter 7

Unsupervised

7.1 Hierarchical Clustering

Divisive method

In this method we assign all of the observations to a single cluster and then partition the cluster to two least similar clusters. Finally, we proceed recursively on each cluster until there is one cluster for each observation.

Agglomerative method

In this method we assign each observation to its own cluster. Then, compute the similarity (e.g., distance) between each of the clusters and join the two most similar clusters. Finally, repeat steps 2 and 3 until there is only a single cluster left.

Linkage or distance matrix Before any clustering is performed, we need to determine the proximity matrix containing the distance between each point using a distance function. Then, the matrix is updated to display the distance between each cluster. The following three methods differ in how the distance between each cluster is measured.

Single Linkage In single linkage hierarchical clustering, the distance between two clusters is defined as the shortest distance between two points in each cluster.

Complete Linkage In complete linkage hierarchical clustering, the distance between two clusters is defined as the longest distance between two points in each cluster.

Average Linkage In average linkage hierarchical clustering, the distance between two clusters is defined as the average distance between each point in one cluster to every point in the other cluster.

7.2 Clustering with a shopping trend data set

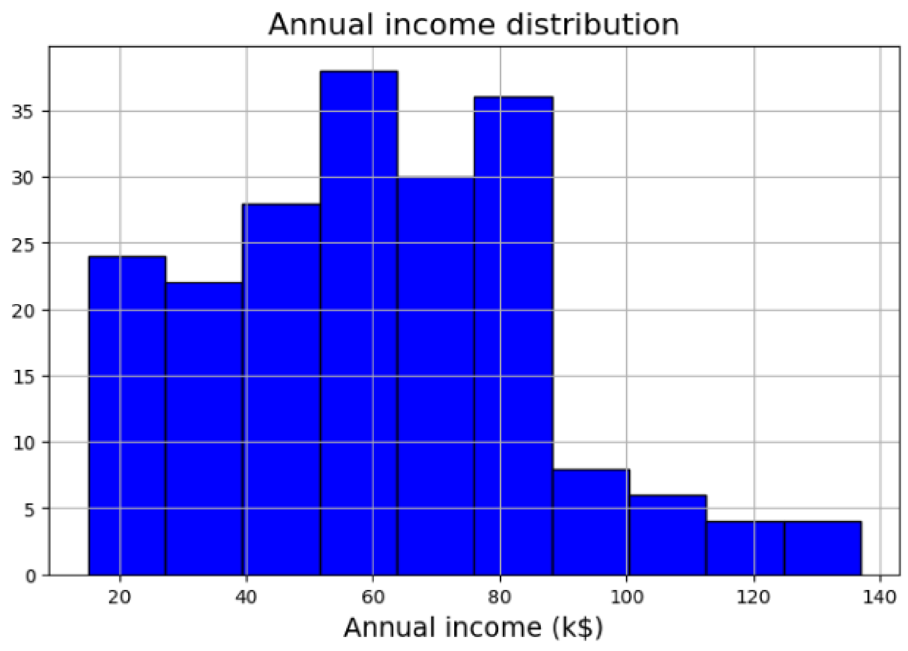
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.cluster.hierarchy as sci
from sklearn.cluster import AgglomerativeClustering
from sklearn.cluster import KMeans
%matplotlib inline

#mount google drive
from google.colab import drive
drive.mount('/content/drive')
```

7.2.1 Step 1: read the dataset

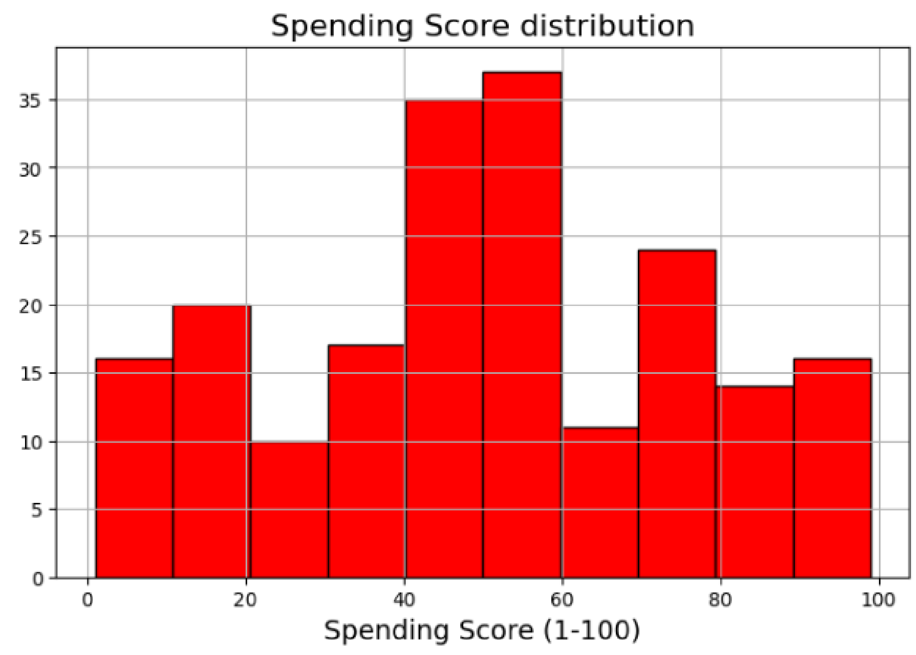
```
df = pd.read_csv('drive/My Drive/Colab Notebooks/Lab6_dataset.csv')
df.head(5)
```

7.2.2 Step 2: exploratory data analysis

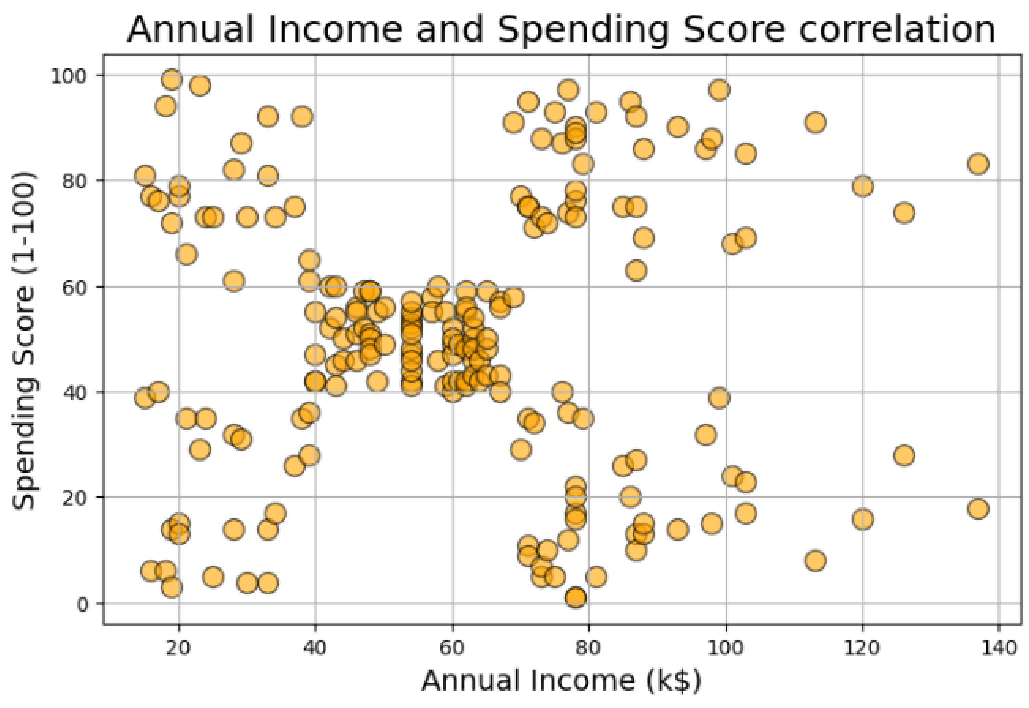
```
plt.figure(figsize=(8,5))
plt.title("Annual income distribution",fontsize=16)
plt.xlabel ("Annual income (k$)",fontsize=14)
plt.grid(True)
plt.hist(df['Annual Income (k$)'],color='blue',edgecolor='k')
plt.show()
```



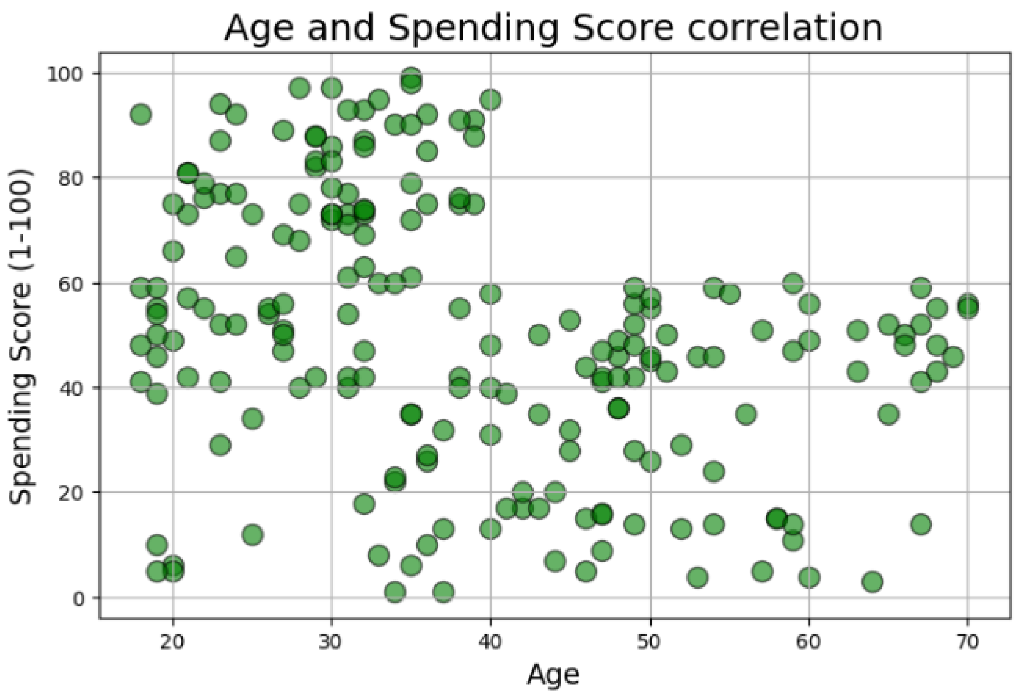
```
plt.figure(figsize=(8,5))
plt.title("Spending Score distribution",fontsize=16)
plt.xlabel ("Spending Score (1-100)",fontsize=14)
plt.grid(True)
plt.hist(df['Spending Score (1-100)'],color='red',edgecolor='k')
plt.show()
```



```
plt.figure(figsize=(8,5))
plt.title("Annual Income and Spending Score correlation",fontsize=18)
plt.xlabel ("Annual Income (k$)",fontsize=14)
plt.ylabel ("Spending Score (1-100)",fontsize=14)
plt.grid(True)
plt.scatter(df['Annual Income (k$)'],df['Spending Score (1-100)'],color='
    orange',edgecolor='k',alpha=0.6, s=100)
plt.show()
```



```
plt.figure(figsize=(8,5))
plt.title("Age and Spending Score correlation",fontsize=18)
plt.xlabel ("Age",fontsize=14)
plt.ylabel ("Spending Score (1-100)",fontsize=14)
plt.grid(True)
plt.scatter(df['Age'],df['Spending Score (1-100)'],color='green',edgecolor=
    'k',alpha=0.6, s=100)
plt.show()
```



7.2.3 Step 3: explore clusters of customers

a) Extract features: annual income and spending score

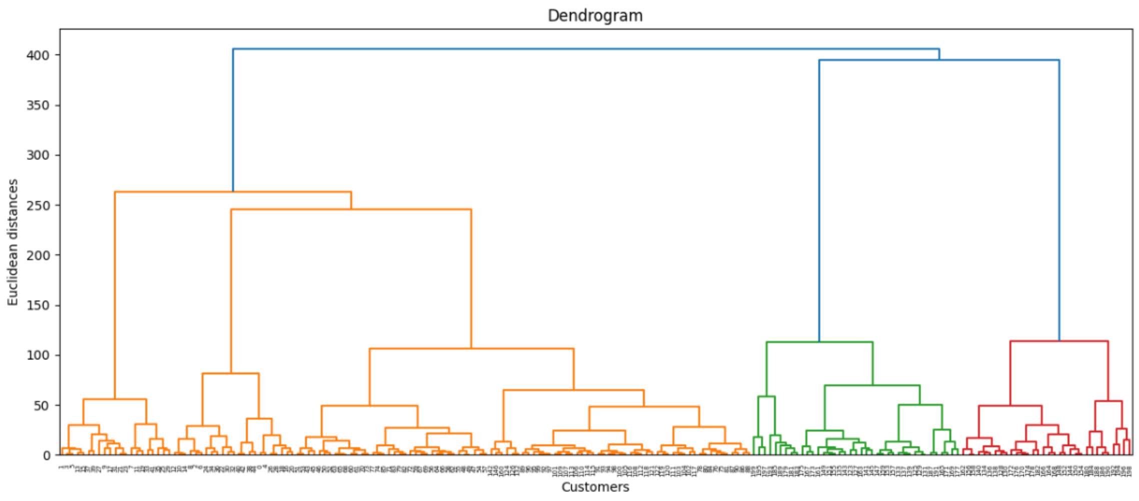
```
X = df.iloc[:,[3,4]].values
```

b) Construct the Ward distance matrix for the dendogram:

$$d(u,v) = \sqrt{\frac{|v|+|s|}{T}d(v,s)^2 + \frac{|v|+|t|}{T}d(v,t)^2 - \frac{|v|}{T}d(s,t)^2}$$

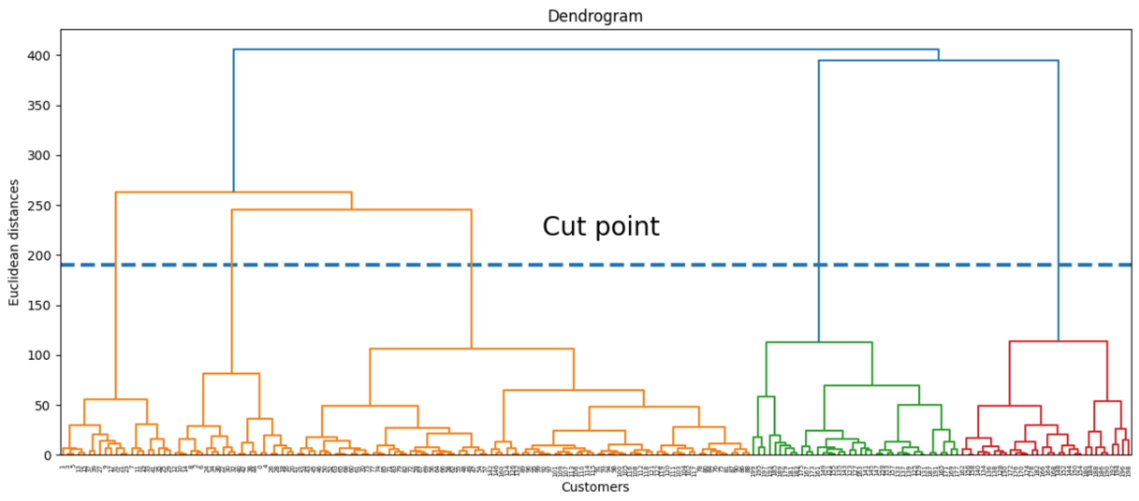
where u is the newly joined cluster consisting of clusters s and t; v is an unused cluster in the forest, $T = |v| + |s| + |t|$ in which $| * |$ is the *cardinality* of its argument.

```
plt.figure(figsize=(15,6))
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Euclidean distances')
dendrogram = sci.dendrogram(sci.linkage(X, method = 'ward'))
plt.show()
```



c) Find the optimal number of clusters

```
plt.figure(figsize=(15,6))
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Euclidean distances')
plt.hlines(y=190,xmin=0,xmax=2000,lw=3,linestyles='--')
plt.text(x=900,y=220,s='Cut point',fontsize=20)
#plt.grid(True)
dendrogram = sci.dendrogram(sci.linkage(X, method = 'ward'))
plt.show()
```



d) Build the clustering model

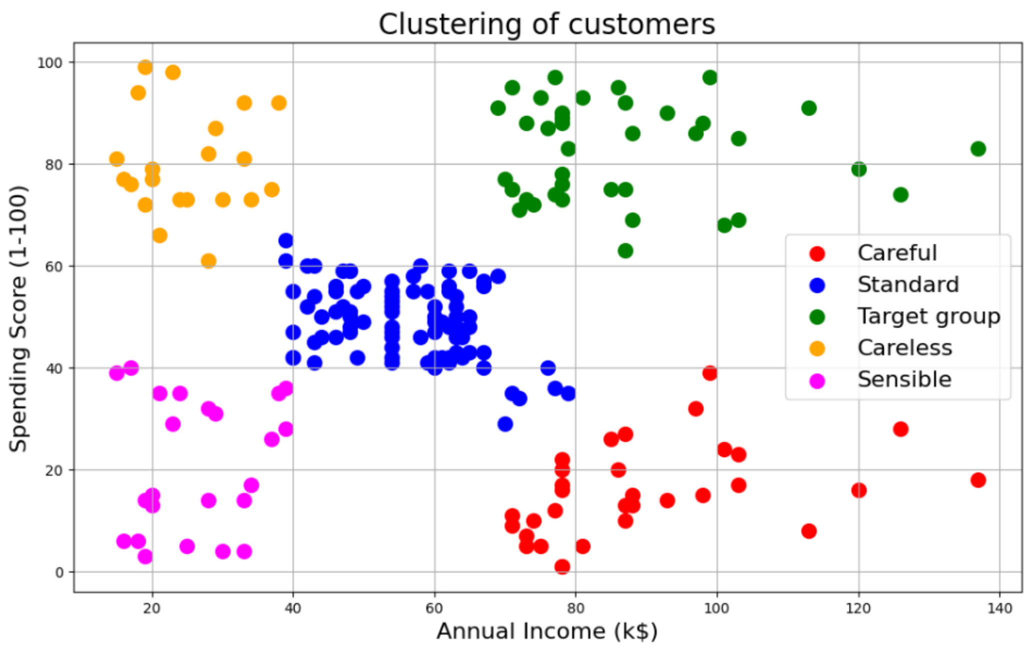
```
hc = AgglomerativeClustering(n_clusters = 5, linkage = 'ward') # Remove
    affinity argument
y_hc = hc.fit_predict(X)
```

e) Plot the clusters and find the customer behaviour

- **Careful**: high income but low spenders
- **Standard**: middle income and middle spenders
- **Target group**: high income and high spenders
- **Careless**: low income but high spenders

- **Sensible**: low income and low spenders

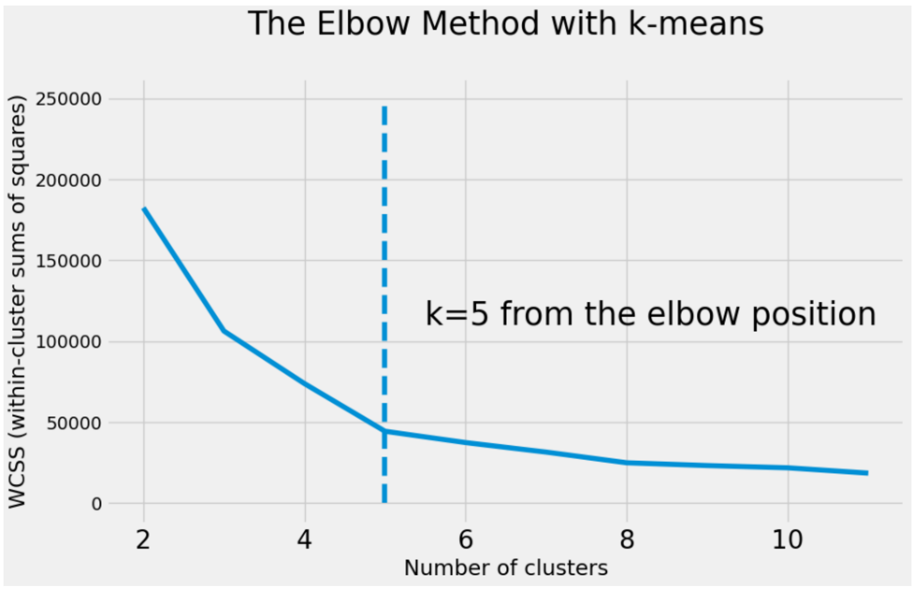
```
plt.figure(figsize=(12,7))
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100, c = 'red', label = '
    Careful')
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100, c = 'blue', label =
    'Standard')
plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s = 100, c = 'green', label =
     'Target group')
plt.scatter(X[y_hc == 3, 0], X[y_hc == 3, 1], s = 100, c = 'orange', label
    = 'Careless')
plt.scatter(X[y_hc == 4, 0], X[y_hc == 4, 1], s = 100, c = 'magenta', label
     = 'Sensible')
plt.title('Clustering of customers',fontsize=20)
plt.xlabel('Annual Income (k$)',fontsize=16)
plt.ylabel('Spending Score (1-100)',fontsize=16)
plt.legend(fontsize=16)
plt.grid(True)
plt.show()
```



f) Check the optimal number of clusters by using the k-means algorithm

We will iterate the model over a range number of clusters (2 to 12) and plot the within-cluster-sum-of-squares (WCSS) matric to determine the optimum number of cluster by elbow method.

```
wcss = []
min_k=2
max_k=12
for i in range(min_k, max_k):
    kmeans = KMeans(n_clusters = i, init = 'k-means++')
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

with plt.style.context(('fivethirtyeight')):
    plt.figure(figsize=(10,6))
    plt.plot(range(min_k, max_k), wcss)
    plt.title('The Elbow Method with k-means\n',fontsize=25)
    plt.xlabel('Number of clusters')
    plt.xticks(fontsize=20)
    plt.ylabel('WCSS (within-cluster sums of squares)')
    plt.vlines(x=5,ymin=0,ymax=250000,linestyles='--')
    plt.text(x=5.5,y=110000,s='k=5 from the elbow position',
             fontsize=25)
    plt.show()
```


The Elbow Method with k-means
WCSS (within-cluster sums of squares)
k=5 from the elbow position
Number of clusters