Chapter 3 - End-to-end Supervised learning

Solution Workbook for Student Practical Classes

Dr Rob Collins 2023

2025-01-11

Table of contents

List of Figures

Figure 1: Introduction of the first 'Pecuniary Judex' machine into the Pennyworth Provident Trust Bank of London. A most remarkable machine capable of determining good prospects from bad based entirely on scientific measurement

3 End-to-end Supervised Learning

3.1 Introduction

In this workshop session we will be creating a loan credit decision model. That is, if we lend money to somebody - then what is the probability that the money will be paid back?

The model uses a very well known data-set called the "German Credit Data" which is widely available on the Internet. The version of the data provided to complete this workshop contains some omissions - so it will require some feature engineering before it can be used..

The model we will build is a 'logistic regression'. Logistic Regression is a common choice when it is required to predict probabilities from regression models - since the results of the model are in the range 0 to 1.

3.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

3.3 Load the required libraries

For this practical session we will need the following libraries:

- 'pandas' : Conventionally given the reference name 'pd'
- 'matplotlib.pyplot' : Conventionally given the reference name 'plt'
- 'missingno' : Conventionally given the reference name 'msno'
- 'LogisticRegression' : from sklearn.linear_model
- 'train_test_split' : from sklearn.model_selection
- 'confusion_matrix' : from sklearn.metrics
- 'seaborn' : Conventionally given the reference name 'sns'
- 'classification_report' : from sklearn.metrics
- 'pickle'

sklearn 'LogisticRegression' will be used to build a classification model. This is documented within the 'sklearn' library documentation (See:

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

'train_test_split' will be used to divide the data-set into two parts - one for model building and the other for model testing.

'confusion_matrix' is used to calculate and display classification test results.

'seaborn' is used here to provide an attractive display of the confusion matrix.

'classification_report' provides another option for summarising and displaying test results.

Create a cell to import all of these libraries.

```
import pandas as pd
import matplotlib.pyplot as plt
import missingno as msno
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import seaborn as sns
from sklearn.metrics import classification_report
import pickle
```

3.4 Load the data

The data-file for this workshop is called `'german credit data unclean.csv'`. Create a cell to load it into a Pandas dataframe called 'credit_data'

```
credit_data = pd.read_csv("german credit data unclean.csv")
```

3.5 Do a basic review of the data

Let's get a feel for what this data looks like in Python. List the columns names and brief descriptions using the Pandas '.info()' method.

```
print("credit_data.info() =")
credit_data.info()
```

```
credit_data.info() =
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 21 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   check_account_status       1000 non-null   object
 1   duration                   1000 non-null   int64
 2   credit_history             1000 non-null   object
 3   purpose                    1000 non-null   object
 4   credit_amount              987 non-null    float64
 5   savings_account            1000 non-null   object
 6   employment_duration        1000 non-null   object
 7   percent_disposable_income  1000 non-null   int64
 8   gender_marriage            1000 non-null   object
 9   other_debtors              1000 non-null   object
 10  recident_since             1000 non-null   int64
 11  property                   1000 non-null   object
 12  age                        988 non-null    float64
 13  other_plans                1000 non-null   object
 14  housing                    1000 non-null   object
 15  num_credits                1000 non-null   int64
 16  job                        1000 non-null   object
 17  dependents                 1000 non-null   int64
 18  telephone                  1000 non-null   object
 19  foreign_worker             1000 non-null   object
 20  default                    1000 non-null   int64
dtypes: float64(2), int64(6), object(13)
memory usage: 164.2+ KB
```

Some of those column names are quite long and will make the display of dataframes rather wide. We can change them to something shorter…

Create two lists:

- A list of current feature names you want to change
- A list of the updated feature names

Zip those two lists into a single dictionary that links 'old name' -> 'new name'

Then use the Pandas 'rename' method to change the feature names. In that method set 'inplace = True' so that you don't create a completely new dataframe.

```
old_column_names = ['check_account_status', 'credit_history',
                    'credit_amount','savings_account',
                    'employment_duration', 'percent_disposable_income',
                    'gender_marriage', 'other_debtors',
                    'recident_since', 'foreign_worker', 'other_plans',
                    'num_credits', 'dependents', 'duration',
                    'purpose' , 'housing', 'telephone', 'property'   ]

new_column_names = ['cs', 'ch', 'ca', 'sa', 'ed', 'pd', 'gm', 'od',
                   'rs', 'fw', 'op', 'nc', 'dp', 'dr', 'pr', 'hs',
                   'tp', 'pp'  ]

column_mapping = dict(zip(old_column_names, new_column_names))

credit_data.rename(columns=column_mapping, inplace=True)

print("Changing feature (column) names:")
for old, new in zip(old_column_names, new_column_names):
    print(f"   {old} --> {new}")

```

```
Changing feature (column) names:
   check_account_status --> cs
   credit_history --> ch
   credit_amount --> ca
   savings_account --> sa
   employment_duration --> ed
   percent_disposable_income --> pd
   gender_marriage --> gm
   other_debtors --> od
   recident_since --> rs
   foreign_worker --> fw
   other_plans --> op
   num_credits --> nc
   dependents --> dp
   duration --> dr
   purpose --> pr
   housing --> hs
   telephone --> tp
   property --> pp
```

Let's now take a look at the actual data ...

Here I have decided to display only the first 12 rows of data .. you may experiment with this to display different portions of the data. I have also separated the display of my table over several cells - this is so that it formats correctly when exported into the pdf book format. In your case, you can simply display this in one cell.

```
credit_data.iloc[0:12, 0:10]
```

	cs	dr	ch	pr	ca	sa	ed	pd	gm	od
0	A11	6	A34	A43	1169.0	A65	A75	4	A93	A101
1	A12	48	A32	A43	5951.0	A61	A73	2	A92	A101
2	A14	12	A34	A46	2096.0	A61	A74	2	A93	A101
3	A11	42	A32	A42	7882.0	A61	A74	2	A93	A103
4	A11	24	A33	A40	4870.0	A61	A73	3	A93	A101
5	A14	36	A32	A46	9055.0	A65	A73	2	A93	A101
6	A14	24	A32	A42	2835.0	A63	A75	3	A93	A101
7	A12	36	A32	A41	6948.0	A61	A73	2	A93	A101
8	A14	12	A32	A43	3059.0	A64	A74	2	A91	A101
9	A12	30	A34	A40	5234.0	A61	A71	4	A94	A101
10	A12	12	A32	A40	1295.0	A61	A72	3	A92	A101
11	A11	48	A32	A49	NaN	A61	A72	3	A92	A101

```
credit_data.iloc[0:12, 10:]
```

	rs	pp	age	op	hs	nc	job	dp	tp	fw	default
0	4	A121	67.0	A143	A152	2	A173	1	A192	A201	0
1	2	A121	22.0	A143	A152	1	A173	1	A191	A201	1
2	3	A121	49.0	A143	A152	1	A172	2	A191	A201	0
3	4	A122	NaN	A143	A153	1	A173	2	A191	A201	0
4	4	A124	53.0	A143	A153	2	A173	2	A191	A201	1
5	4	A124	35.0	A143	A153	1	A172	2	A192	A201	0
6	4	A122	53.0	A143	A152	1	A173	1	A191	A201	0
7	2	A123	35.0	A143	A151	1	A174	1	A192	A201	0
8	4	A121	61.0	A143	A152	1	A172	1	A191	A201	0
9	2	A123	28.0	A143	A152	2	A174	1	A191	A201	1
10	1	A123	25.0	A143	A151	1	A173	1	A191	A201	1
11	4	A122	24.0	A143	A151	1	A173	1	A191	A201	1

You may notice immediately there there is some missing data in this table (Row 11 of the 'credit_amount' (ca) feature). This is important and gives us a clue that we need to clean and tidy the data.

3.6 Step 1 : Clean and Tidy Data

3.6.1 Visual Check of Missing Data

We are going to visualize missing data using the 'missingno' library. You should be familiar with this library from the earlier 'Feature Engineering' workbook.

Add a cell to create a graphic display that allows you to visualize missing data in the 'credit_data' dataframe.

```
msno.matrix(credit_data)
plt.show()
```

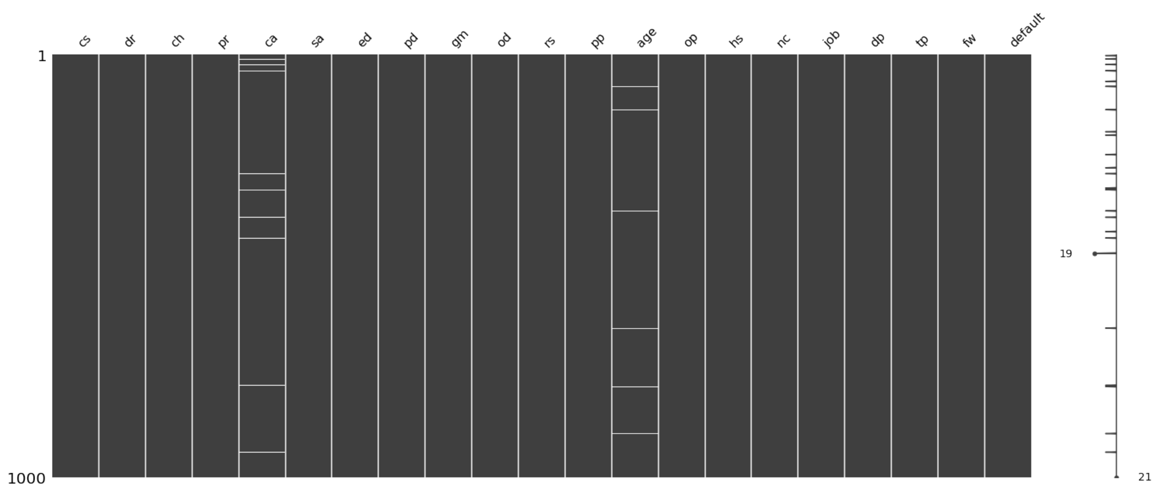

Figure 2: missingno matrix of the credit_data dataframe to identify missing data

Use a second missingno function to display a bar-chart of the number of data items in each feature of the credit_data data-set.

```
msno.bar(credit_data)
plt.show()
```

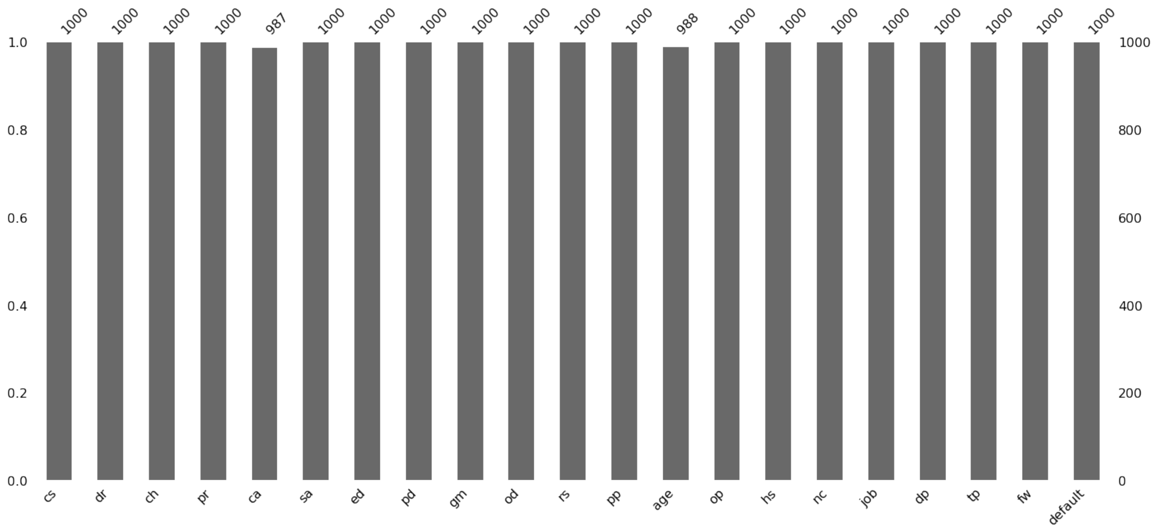

Figure 3: Missingno bar chart

3.6.2 Repair missing data

You should apply two different strategies to 'repair' the missing data:

1. Removing any data records with a missing 'credit_amount', and
2. Imputing missing values for the 'age' feature - using the average age from the data-set

Add Python code that removes any row of data that has a 'NaN' in the 'credit_amount' feature.

Hint: There reference for the Pandas 'dropna' function is here:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html]

```
credit_data.dropna(axis='index', subset= ['ca'],inplace=True)
credit_data.iloc[0:12, 0:6]
```

	cs	dr	ch	pr	ca	sa
0	A11	6	A34	A43	1169.0	A65
1	A12	48	A32	A43	5951.0	A61
2	A14	12	A34	A46	2096.0	A61
3	A11	42	A32	A42	7882.0	A61
4	A11	24	A33	A40	4870.0	A61
5	A14	36	A32	A46	9055.0	A65
6	A14	24	A32	A42	2835.0	A63
7	A12	36	A32	A41	6948.0	A61
8	A14	12	A32	A43	3059.0	A64

	cs	dr	ch	pr	ca	sa
9	A12	30	A34	A40	5234.0	A61
10	A12	12	A32	A40	1295.0	A61
12	A12	12	A32	A43	1567.0	A61

Check the above table ... Row 11, amongst others, should have been deleted and there should be 987 rows remaining in the data-set. The 'Age' feature will still contain 'NaN' values.

```python
print("credit_data.info() =")
credit_data.info()
```

```
credit_data.info() =
<class 'pandas.core.frame.DataFrame'>
Int64Index: 987 entries, 0 to 999
Data columns (total 21 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   cs       987 non-null    object 
 1   dr       987 non-null    int64  
 2   ch       987 non-null    object 
 3   pr       987 non-null    object 
 4   ca       987 non-null    float64
 5   sa       987 non-null    object 
 6   ed       987 non-null    object 
 7   pd       987 non-null    int64  
 8   gm       987 non-null    object 
 9   od       987 non-null    object 
 10  rs       987 non-null    int64  
 11  pp       987 non-null    object 
 12  age      976 non-null    float64
 13  op       987 non-null    object 
 14  hs       987 non-null    object 
 15  nc       987 non-null    int64  
 16  job      987 non-null    object 
 17  dp       987 non-null    int64  
 18  tp       987 non-null    object 
 19  fw       987 non-null    object 
 20  default  987 non-null    int64  
dtypes: float64(2), int64(6), object(13)
memory usage: 169.6+ KB
```

3.6.2.1 A short digression : Panda and nan (not-a-number) values

At this point we would like to be filter the dataframe to display only those rows that contain 'nan' (not-a-number) values. The temptation (and I would say, 'obvious', choice) would be to use a comparison operator. Something like the following for example:

```
credit_data[credit_data['cs'] == np.nan]
```

However, python and pandas specifically treats 'nan' as unrecognizable / undefined.

The results of the following might surprise you:

```
import numpy as np
if (np.nan == np.nan):
    print("True")
else:
    print("False")
```

```
False
```

Thus, if you wish to match data in a pandas dataframe with 'nan' you have to instead use a dedicated function provided by pandas called '.isnull()'.

3.6.2.2 Back to the main thread of this section ...

Add some code that prints out a few of the rows where the age feature is NaN

```
age_null_df = credit_data[ credit_data['age'].isnull()]
print("For display reasons, I only printed a few columns:")
age_null_df.iloc[0:10, 5:15]
```

```
For display reasons, I only printed a few columns:
```

	sa	ed	pd	gm	od	rs	pp	age	op	hs
3	A61	A74	2	A93	A103	4	A122	NaN	A143	A153
76	A61	A72	4	A93	A101	3	A123	NaN	A143	A152
131	A61	A73	4	A93	A101	3	A122	NaN	A142	A152
183	A64	A73	4	A93	A101	4	A121	NaN	A143	A152
268	A61	A75	1	A91	A101	4	A122	NaN	A143	A152
316	A61	A73	2	A93	A103	3	A122	NaN	A143	A152
370	A65	A73	4	A93	A101	4	A121	NaN	A143	A152
419	A65	A73	4	A92	A101	2	A122	NaN	A143	A152
647	A63	A73	2	A92	A101	2	A122	NaN	A143	A152
785	A64	A73	4	A93	A101	2	A122	NaN	A143	A152

Note that row 3 of 'age' is a NaN .. we can check that in a moment after fixing.

Add Python code that replaces any missing ('NaN') values in the 'Age' feature with the average of all ages in the data-set.

hint : The following may be useful:

[https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html]

```
mean_age = credit_data['age'].mean()

credit_data['age'] = credit_data['age'].fillna(mean_age)
```

Show the results by displaying a few rows of the relevant features of the dataframe.

```
credit_data.iloc[0:5,5:15]
```

	sa	ed	pd	gm	od	rs	pp	age	op	hs
0	A65	A75	4	A93	A101	4	A121	67.000000	A143	A152
1	A61	A73	2	A92	A101	2	A121	22.000000	A143	A152
2	A61	A74	2	A93	A101	3	A121	49.000000	A143	A152
3	A61	A74	2	A93	A103	4	A122	35.580943	A143	A153
4	A61	A73	3	A93	A101	4	A124	53.000000	A143	A153

3.6.3 Change all of the category data into numbers

The file includes a number of features that are coded into categories. For example 'sa' is coded as A65, A61 etc. We need to convert those categories into numbers ...

Use the Pandas 'get_dummies' method introduced in a previous workshop to fix this.

Create a list called 'categorical_features' : a list of all of the categorical features in your dataframe that you wish to convert to one-hot-encoded.

Pass this to the pd.get_dummies() function along with the name of your dataframe to convert each feature named in the list to to one-hot-encoded.

```
pd.set_option('display.max_columns', None)

categorical_features = ['cs', 'ch',
                        'pr', 'sa', 'ed',
                        'gm', 'od', 'op',
                        'hs', 'job', 'tp', 'fw', 'pp' ]
```

```python
credit_data = pd.get_dummies(credit_data, columns=categorical_features)
```

This obviously creates a lot more features .. 62 in total:

```python
print("credit_data.info() =")
credit_data.info()
```

```
credit_data.info() =
<class 'pandas.core.frame.DataFrame'>
Int64Index: 987 entries, 0 to 999
Data columns (total 62 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   dr        987 non-null    int64
 1   ca        987 non-null    float64
 2   pd        987 non-null    int64
 3   rs        987 non-null    int64
 4   age       987 non-null    float64
 5   nc        987 non-null    int64
 6   dp        987 non-null    int64
 7   default   987 non-null    int64
 8   cs_A11    987 non-null    uint8
 9   cs_A12    987 non-null    uint8
 10  cs_A13    987 non-null    uint8
 11  cs_A14    987 non-null    uint8
 12  ch_A30    987 non-null    uint8
 13  ch_A31    987 non-null    uint8
 14  ch_A32    987 non-null    uint8
 15  ch_A33    987 non-null    uint8
 16  ch_A34    987 non-null    uint8
 17  pr_A40    987 non-null    uint8
 18  pr_A41    987 non-null    uint8
 19  pr_A410   987 non-null    uint8
 20  pr_A42    987 non-null    uint8
 21  pr_A43    987 non-null    uint8
 22  pr_A44    987 non-null    uint8
 23  pr_A45    987 non-null    uint8
 24  pr_A46    987 non-null    uint8
 25  pr_A48    987 non-null    uint8
 26  pr_A49    987 non-null    uint8
 27  sa_A61    987 non-null    uint8
 28  sa_A62    987 non-null    uint8
 29  sa_A63    987 non-null    uint8
 30  sa_A64    987 non-null    uint8
```

```
 31  sa_A65    987 non-null    uint8
 32  ed_A71    987 non-null    uint8
 33  ed_A72    987 non-null    uint8
 34  ed_A73    987 non-null    uint8
 35  ed_A74    987 non-null    uint8
 36  ed_A75    987 non-null    uint8
 37  gm_A91    987 non-null    uint8
 38  gm_A92    987 non-null    uint8
 39  gm_A93    987 non-null    uint8
 40  gm_A94    987 non-null    uint8
 41  od_A101   987 non-null    uint8
 42  od_A102   987 non-null    uint8
 43  od_A103   987 non-null    uint8
 44  op_A141   987 non-null    uint8
 45  op_A142   987 non-null    uint8
 46  op_A143   987 non-null    uint8
 47  hs_A151   987 non-null    uint8
 48  hs_A152   987 non-null    uint8
 49  hs_A153   987 non-null    uint8
 50  job_A171  987 non-null    uint8
 51  job_A172  987 non-null    uint8
 52  job_A173  987 non-null    uint8
 53  job_A174  987 non-null    uint8
 54  tp_A191   987 non-null    uint8
 55  tp_A192   987 non-null    uint8
 56  fw_A201   987 non-null    uint8
 57  fw_A202   987 non-null    uint8
 58  pp_A121   987 non-null    uint8
 59  pp_A122   987 non-null    uint8
 60  pp_A123   987 non-null    uint8
 61  pp_A124   987 non-null    uint8
dtypes: float64(2), int64(6), uint8(54)
memory usage: 121.4 KB
```

At this point you can display the whole dataframe in a single cell. As previously, I have truncated my display somewhat so that it fits the workbook format.

```
credit_data.iloc[0:5, 0:11]
```

	dr	ca	pd	rs	age	nc	dp	default	cs_A11	cs_A12	cs_A13
0	6	1169.0	4	4	67.000000	2	1	0	1	0	0
1	48	5951.0	2	2	22.000000	1	1	1	0	1	0
2	12	2096.0	2	3	49.000000	1	2	0	0	0	0
3	42	7882.0	2	4	35.580943	1	2	0	1	0	0

	dr	ca	pd	rs	age	nc	dp	default	cs_A11	cs_A12	cs_A13
4	24	4870.0	3	4	53.000000	2	2	1	1	0	0

3.7 Step 2 Select the Algorithm

The first algorithm we will be using for this classification problem will be Logistic Regression .. so we can go ahead and build the model..

3.8 Step 3 Build the Model

In this section we be using the logistic regression classifier provided by sklearn. Initially we will build a demonstration example of model building using all of the data in the data-set. In the next section we will split the data into two parts to enable testing of the model.

Add a cell with code to split the data into two parts. use 'X' to represent the known 'inputs' to the model. Set 'Y' to represent the 'label' or known (expected) output from the model.

```
X = credit_data.drop(columns=['default'])
Y = credit_data['default']
```

Those new dataframes have the following shapes:

```
print(f"X.shape = {X.shape}")
```

```
X.shape = (987, 61)
```

```
print(f"Y.shape = {Y.shape}")
```

```
Y.shape = (987,)
```

That is, 'Y' is a single dimensional vector of 987 elements.

Create an instance of the specific modelling algorithm called 'logModel'

```
logModel = LogisticRegression(solver='liblinear')
```

Then use logModel.fit(X,Y) to build the model:

```
logModel = logModel.fit(X,Y)
```

Print the 'score' attribute from the logModel to show that the model has 'worked'.

Note This is not really legitimate since it scores (provides one quality measure of) the model - but it does so using exactly the same data that the original model was built from. This would be a bit like setting a student an exam consisting of questions they had already practiced in class.

However, for now it provides an indication that we have, at least, built a model!

```
print (f"logModel.score(X,Y) = {logModel.score(X, Y):.3f}")
```

```
logModel.score(X,Y) = 0.785
```

3.9 Step 4 Check Model Quality

In practice, models will always require a quality check. A common way of achieving this is to split the original data into two parts. One part will be used for building the model, the other part will be used to test how good it is at making predictions. There are various strategies regarding how to make this split - some of which we will cover in later sessions. However, a common and easy way to split data is simply to make a random selection of around 20% of the data and to retain this for testing. The remaining 80% can be used for testing.

Sklearn provides a simple method for random splitting of data called 'train_test_split':

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Use this function to split the data into two parts in the ratio 80%:20%:

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2)
```

Then print the first few rows of the test data-set.

```
print(f"y_test[0:20] = \n{y_test[0:20]}")
```

```
y_test[0:20] =
486    0
744    0
977    0
593    1
338    0
797    0
641    0
478    0
461    0
572    0
```

```
402    1
972    1
477    0
51     0
773    0
939    0
580    1
747    1
696    0
672    0
Name: default, dtype: int64
```

Now we can build a model as above. However this time based only on the training data not the test data.

```
logModel = logModel.fit(X_train,y_train)
```

We then want to test this model. That means generating a set of predictions based on the **test** data. To do this you will need to pass the test data you generated to the 'logModel.predict' method:

```
predictions = logModel.predict(X_test)
print (f"predictions = \n{predictions}")
```

```
predictions =
[0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 0 1 1 1 1
 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
 0 1 0 0 0 0 0 1 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 1 0
 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1 0
 1 0 0 1 0 0 1 0 0 1 0 0 0]
```

Now we are in a position to get a real 'score' for the model, this time based on data the model has not seen before. The 'score' method of logModel what fraction of the time did did the model predict the correct answer.

There are two things to note at this point.

1. 'score' is only one of several quality measures for classifiers and on its own can be misleading. So you will generally need to employ other metric.
2. Because the data we used is split randomly every time the model is built .. you may get different results to the ones shown below.

Add a line of code to print the 'score' for the model based on the test data obtained above.

```
print(f"LogModel.Score = {logModel.score(X_test,y_test):.3f}")
```

```
LogModel.Score = 0.742
```

A more useful tool for measuring quality is the ‘confusion matrix’. The confusion matrix will be described in the lecture portion of this course.

You can access this tool by importing the ‘confusion_matrix’ method from sklearn.metrics.

```
cm = confusion_matrix(y_test, predictions)
print (f"Confusion matrix =\n {cm}")
```

```
Confusion matrix =
 [[119  21]
 [ 30  28]]
```

This output might be useful to a person who is familiar with the confusion matrix but in this ‘vanilla’ form it is somewhat difficult to interpret. In most cases you will want to ‘wrap’ this in a more graphically attractive output format and this can be achieved using a seaborn heatmap:

```
plt.figure(figsize=(4, 4))
sns.set(font_scale=1.2)
sns.heatmap(cm, annot=True,
            fmt="d",
            cmap="Blues",
            annot_kws={"size": 14},
            cbar=False,
            xticklabels=["No Default", "Default"],
            yticklabels=["No Default", "Default"])
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

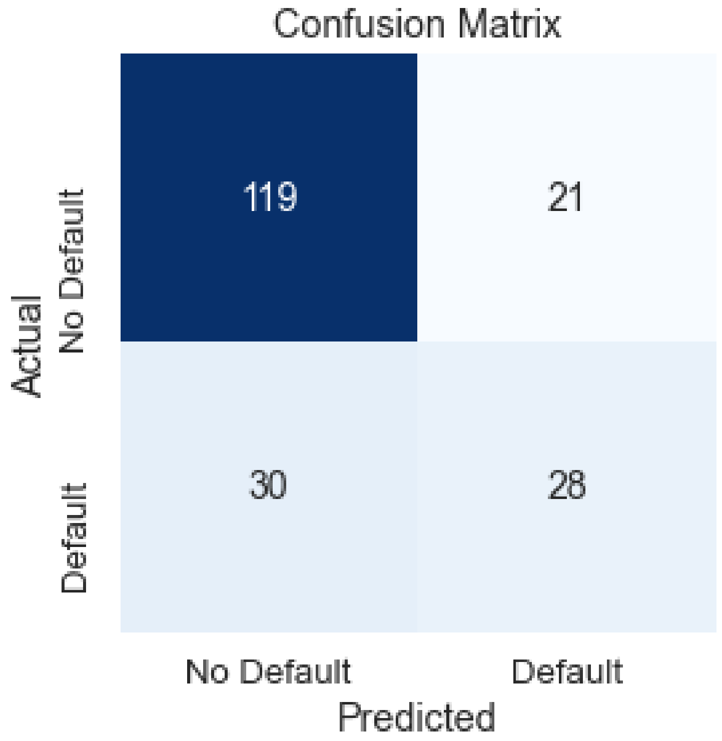

Figure 4: Confusion Matrix display using Seaborn library

Alternatively, you might use decide to use the ‘classification_report’ method from the ‘sklearn.metrics’ library to provide another view of the quality of your classification:

- Import ‘sklearn.metrics’
- Define the names of each classification in a list
- Apply the sklearn ‘classification_report’ method - passing ‘y_test’, ‘predictions’ and the list of classification names as parameters

```
target_names = ['No Default', 'Default']
```

```
print(classification_report(y_test, predictions,
                            target_names=target_names,
                            digits = 4))
```

```
              precision    recall  f1-score   support

  No Default     0.7987    0.8500    0.8235       140
     Default     0.5714    0.4828    0.5234        58

    accuracy                         0.7424       198
```

```
   macro avg     0.6850     0.6664     0.6734        198
weighted avg     0.7321     0.7424     0.7356        198
```

3.10 Step 5: Build the model into an application

In the case of Logistic Regression the 'model' is just a list of coefficients for an equation. You can access the values of these coefficients using the '.intercept_' and 'coef_' attributes of logModel.

```
print( f"logModel.intercept_ = {logModel.intercept_}")
print( f"logModel.coef_  = \n{logModel.coef_}")
```

```
logModel.intercept_ = [-0.70039774]
logModel.coef_  =
[[ 2.74314388e-02  1.48840514e-04  3.44578393e-01  7.74924472e-02
  -1.84819321e-02  1.63568286e-01  1.44643494e-01  5.25822915e-01
   9.30451210e-02 -2.92114516e-01 -1.02715126e+00  3.15894039e-01
   3.90508750e-01 -1.10955363e-01 -4.70238631e-01 -8.25606532e-01
   4.72747454e-01 -1.01495590e+00 -8.90429152e-02 -1.45045068e-01
  -3.87908642e-01 -7.85913890e-02  9.46174970e-02  7.21364776e-01
  -1.46238321e-01 -1.27345225e-01  3.94563897e-01 -1.47411572e-01
  -2.08287629e-01 -2.38065852e-01 -5.01196580e-01 -5.92158761e-02
  -1.04405968e-02 -4.41171430e-02 -4.86958351e-01 -9.96657698e-02
   1.78303956e-01 -9.59815216e-02 -6.36250537e-01 -1.46469634e-01
  -6.00110266e-02  1.88472374e-01 -8.28859084e-01 -7.99132960e-02
  -2.49612458e-02 -5.95523195e-01 -6.64136894e-03 -4.26629465e-01
  -2.67126903e-01 -8.47735472e-02 -1.15654395e-01 -1.16004507e-01
  -3.83965288e-01 -1.55970873e-01 -5.44426864e-01  2.01631857e-01
  -9.02029594e-01 -4.06265977e-01 -2.56444208e-02 -2.59970412e-01
  -8.51692790e-03]]
```

It will be practically useful when building our application to have a record of each of the feature names and their corresponding indices (column numbers).

Write code to iterate through the 'X_train' dataframe. For each column in the dataframe create one element of the dictionary composed of the feature name and the column number.

Hint: the Python 'enumerate' function is useful here. 'Enumerate' is an iterable object which, when given a list returns pairs of values - an index (position) in the list, and the item at that position.

To aid re-use of this information, format the output from the loop so that it can be directly copy-pasted into the planned 'Credit Evaluation' application. This information could easily be written to a file for later use .. but in this case we are simply going to copy-paste the resulting string into our application code.

```
print("Feature_dict = {")
for ind, col in enumerate(X_train.columns):
    print(f"'{col}':{ind}", end=", ")
print("}")
```

```
Feature_dict = {
'dr':0, 'ca':1, 'pd':2, 'rs':3, 'age':4, 'nc':5, 'dp':6, 'cs_A11':7, 'cs_A12':8, 'cs_A13':
```

```
Feature_dict = {
'dr':0, 'ca':1, 'pd':2, 'rs':3, 'age':4, 'nc':5, 'dp':6, 'cs_A11':7, 'cs_A12':8, 'cs_A13':
```

```
print(f"Len feature dict = {len(Feature_dict)}")
```

```
Len feature dict = 61
```

Rather than build our application within this workbook it is more natural to save the model parameters at this point, then have the application load these parameters when required. The following code will enable the model to be saved as a ‘pickle’ file.

```
import pickle
model_filename = 'germ_cred_model.pkl'
# Open the file to save as pkl file
the_file = open(model_filename, 'wb')
pickle.dump(logModel, the_file)
# Close the pickle instances
the_file.close()
```