REPORT 2 – BIOMEDICAL MACHINE LEARNING

A practical report on data handling, analysis and machine learning experiments carried out by Umi Hussain

NOVEMBER 28, 2022

NEWCASTLE UNIVERSITY'S SCHOOL OF COMPUTING

Pre-processing

Merging of files

To start things of with pre-processing and cleaning the data I decided to look at the structure of the data. I realised that there was 5 different excel spreadsheets that contained the various different visits of patients. I decided that merging all 5 files into one data frame was the best course of action, I done this by utilising pandas concat() function.

Sorting the various columns and dealing with null values

I then sorted the columns into the correct types, most of these columns where just numeric float types, however a few where set out to be string columns.

```
# string columns
complete_data[['MRI_ID', 'sex', 'hand', 'CDR']] = complete_data[['MRI_ID', 'sex', 'hand', 'CDR']].astype(str)
```

After this I made sure that the ID column in the data frame could not be N/A as this is intended to act like the primary key and help identify the various patients, I then filled out any missing information with 0.

```
# ID != 'N/A'
complete_data = complete_data[complete_data['ID'].notna()]

# any empty information = '0'
complete_data = complete_data.fillna(value=0)
```

Dealing with the patient ages issue, sorting the data frame by ID/MRI_ID and removing basic_age from the data frame

Once I had sorted out null values and successfully merged the data I noticed that the patient ages did not match up with there annual visits, to tackle this issue I took the basic_age of the patient and added visit – 1 to it. Doing so meant that the patients age correctly updated depending on what visit they where on. After this I sorted the data frame by ID and MRI_ID in ascending order to make it more presentable. I then decided to drop the basic_age column as there is no real need for it.

```
# replace complete_data age with basic_age + visit - 1 to get the accurate age of the patient
complete_data['age'] = complete_data['basic_age'] + complete_data["visit"] - 1

# sort complete_data by ID and then the ID's MRI_ID (ascending order)
complete_data = complete_data.sort_values(by=['ID', 'MRI_ID'])
```

```
# get rid of basic_age as it has no use
complete_data.drop(columns="basic_age", axis=0, inplace=True)
```

Resetting the ID column due to gaps

I then noticed that there where plenty of gaps throughout the ID column, to make the data more consistent I iterated through the data frame and used some simple if/elif statements to deal with this issue.

```
# reset ID column to get rid of gaps
for i in range(1, len(complete_data)):
    curr = complete_data.loc[i, "ID"]
    prev = complete_data.loc[i-1, "ID"]
    if curr - prev >= 1 and complete_data.loc[i, "visit"] == 1:
        complete_data.loc[i, "ID"] = prev + 1
    elif curr - prev >= 1 and complete_data.loc[i, "visit"] != 1:
        complete_data.loc[i, "ID"] = prev
```

Changing up the MRI_ID column and dropping the visit column from the data frame

Once I had sorted out the ID column, I noticed that the MRI_ID was completely incorrect and did not match the ID column, I also noticed that there was no real reason for both the MRI_ID and visit column, as whenever the patient would next visit the MRI number would increase by 1. So I decided to tackle both problems at once by getting rid of the starting digits of the MRI_ID and dropping the column visit. The digits at the start of MRI_ID where no longer needed as we had our unique specific ID column all set up. To finish up with this part of the data frame I renamed MRI_ID to MRI_visit, so that the column name correctly represented the data below, which showed what MRI visit the patient was on.

```
# drop first part of MRI_ID
for i in range(0, len(complete_data)):
    complete_data.loc[i, "MRI_ID"] = complete_data.loc[i, "MRI_ID"][5:]

# rename column MRI_ID to MRIvisit
complete_data.rename(columns = {'MRI_ID':'MRI_visit'}, inplace=True)

# get rid of visit column as it now has no use
complete_data.drop(columns="visit", axis=0, inplace=True)
```

	ID	MRI_visit
1	1	MR1
1	1	MR2
2	2	MR1
2	2	MR2
2	2	MR3
3	3	MR1
3	3	MR2
4	4	MR1
4	4	MR2
4	4	MR3
5	5	MR1
5	5	MR3
5	5	MR4
6	6	MR1
6	6	MR2

Correcting spelling errors in the CDR column and creating an error checker for the ASF column

I then noticed that there where some spelling errors in the CDR column, so to fix this I used pandas inbuilt function replace(). I also noticed that some values had been typed in wrong in the ASF column, and where 100x larger than the rest of the data. So I created a simple error checker as a solution to this problem.

```
# correct any spelling errors in the CDR column
complete_data = complete_data.replace(('very midl', 'vry mild', 'very miId'), 'very mild')
complete_data = complete_data.replace('midl', 'mild')

# error checker for ASF column
for i in range(0, len(complete_data)):
    if complete_data.loc[i, "ASF"] > 2:
        complete_data = complete_data.replace(complete_data.loc[i, "ASF"], complete_data.loc[i, "ASF"] / 100)
```

Getting rid of any duplicate values, reviewing the data frame, and saving the file as new .pkl.gz file

I noticed a few duplicate values so I got rid of them by using the inbuilt function drop_duplicates(). I then took a look over the data frame and ensured that I had cleaned and processed it to the best of my ability, before lastly saving the processed data frame to a new file in the data folder.

```
complete_data
```

	ID	MRI_visit	delay	sex	hand	age	YOE	SES	MMSE	CDR	eTIV	nWBV	ASF
0	1	MR1	0	M	R	87	14	2.0	27.0	none	1987.0	0.696	0.883
1	1	MR2	457	M	R	88	14	2.0	30.0	none	2004.0	0.681	0.876
2	2	MR1	0	M	R	75	12	0.0	23.0	very mild	1678.0	0.736	1.046
3	2	MR2	560	M	R	76	12	0.0	28.0	very mild	1738.0	0.713	1.010
4	2	MR3	1895	M	R	77	12	0.0	22.0	very mild	1698.0	0.701	1.034
5	3	MR1	0	F	R	88	18	3.0	28.0	none	1215.0	0.710	1.444
6	3	MR2	538	F	R	89	18	3.0	27.0	none	1200.0	0.718	1.462
7	4	MR1	0	M	R	80	12	4.0	28.0	none	1689.0	0.712	1.039
8	4	MR2	1010	M	R	81	12	4.0	29.0	very mild	1701.0	0.711	1.032
9	4	MR3	1603	M	R	82	12	4.0	30.0	none	1699.0	0.705	1.033
10	5	MR1	0	M	R	71	16	0.0	28.0	very mild	1357.0	0.748	1.293
11	5	MR3	518	M	R	73	16	0.0	27.0	mild	1365.0	0.727	1.286
12	5	MR4	1281	M	R	74	16	0.0	27.0	mild	1372.0	0.710	1.279
13	6	MR1	0	F	R	93	14	2.0	30.0	none	1272.0	0.698	1.380
14	6	MR2	742	F	R	94	14	2.0	29.0	none	1257.0	0.703	1.396

```
# saving the clean data to a .csv file in the data folder
complete_data.to_csv("C:/Users/umihu/OneDrive - Newcastle University/Documents/CSC3432/CW - part 2/data/pro
```

Exploratory analysis

Part 1

With both the exploratory analysis, and pre-processing, I realised that the two scripts would sort of work hand in hand. As I was analysing the data I was seeing areas that I could clean and change, therefore I started working on both scripts in tandem and ended up finishing both scripts roughly around the same time as well.

To start of with the analysis I used some simple in built analysis techniques utilising some of pandas inbuilt functions, such as info(), describe(), head() and isnull().sum(). I then ran a simple aggregate() query to see if there was any correlation between age and SES levels.

```
# see if the mean SES of each age shows any pattern
g = data.groupby("age")
g.SES.aggregate(statistics.mean)
```

I also created a very simple table to see if SES levels have any correlation with a particular sex.

```
# have a look if SES level has any correlation with sex
data.groupby("sex").SES.describe()
```

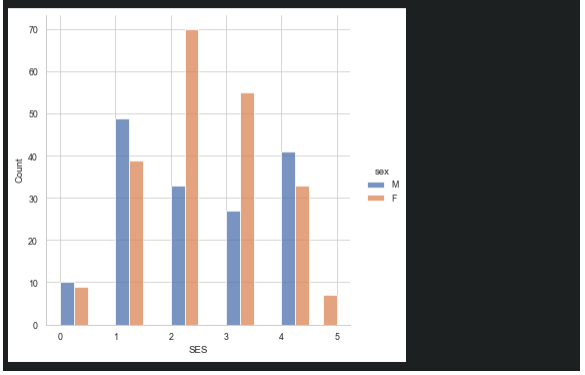
sex	count	mean	std	min	25%	50%	75%	max
F	213.0	2.399061	1.171758	0.0	2.0	2.0	3.0	5.0
M	160.0	2.250000	1.303117	0.0	1.0	2.0	4.0	4.0

Part 2

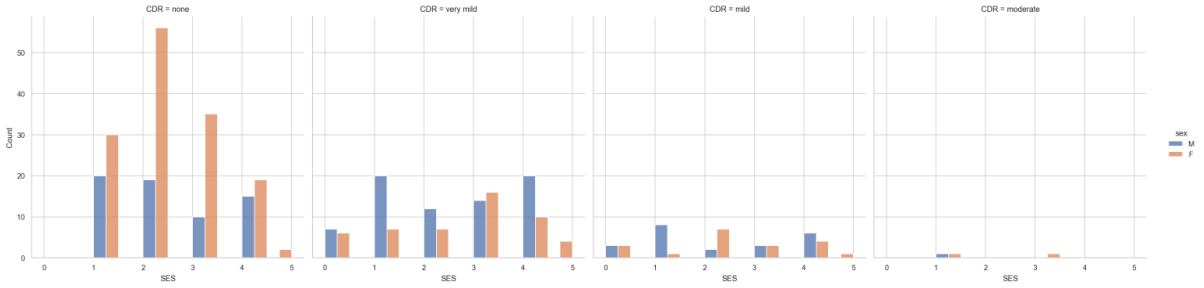
For the second part of the analysis I decided to visualise some of the query's and data comparisons that I was looking into, I done this by utilising the seaborn and matplotlib.pylot libraries.

The first diagram I generated was a histogram using Seaborn's displot functionality, I reviewed SES levels for the two separate sexes.

```
grid = seaborn.displot(x="SES", hue="sex", multiple="dodge", data=data)
```

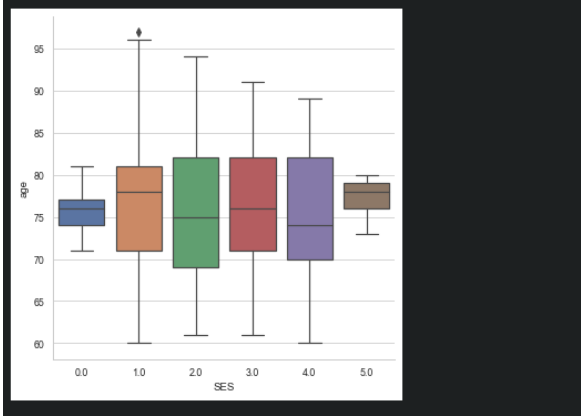


I then developed this histogram by looking at how the same graph would look across the 4 different CDR levels.



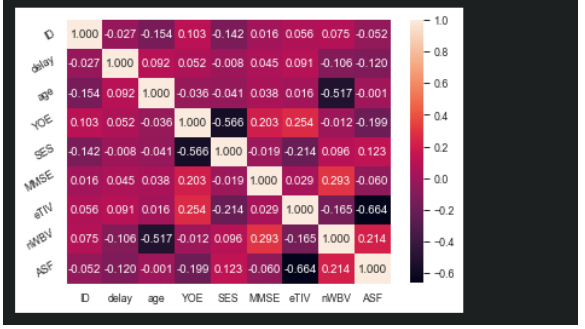
I then plotted a catplot to see the correlation between SES and age, the kind of graph is a box plot.

```
grid = seaborn.catplot(x="SES", y="age", kind="box", data=data)
```



Lastly I decided to create a heatmap which measured the different corroleation strength between each of the columns that where present in the dataframe.

```
fig, p = plt.subplots()
seaborn.heatmap(data.corr(), annot=True, fmt=".3f", ax=p)
for label in p.get_yticklabels():
    label.set_rotation(30)
```



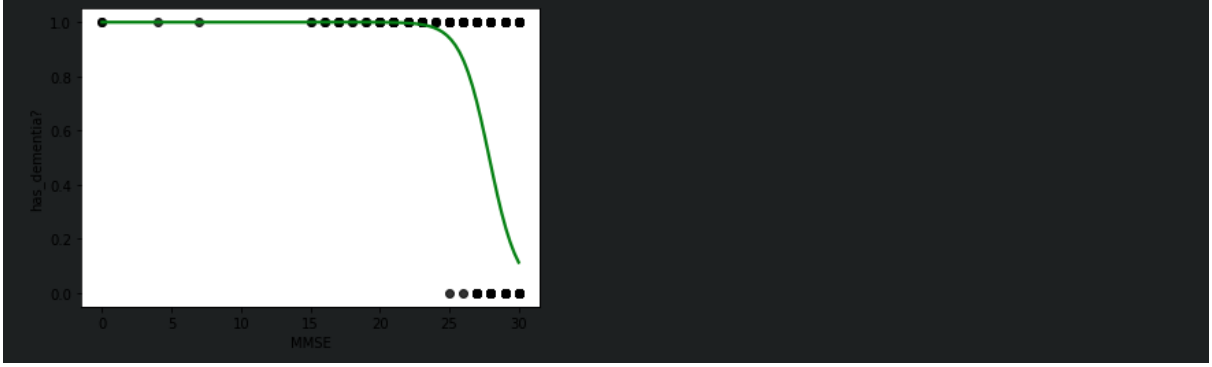
Logistic Regression

To start with, I decided to investigate Logistic Regression on the dataframe. This was because I came up with the idea to measure whether or not a patient had dementia by looking into other columns within the data. I started off with the creation of a new Boolean column has_dementia through basic python conversion of the column 'CDR'. I then decided it would be interesting to measure the relation between has_dementia and MMSE as seen below:

```
x = x_data["MMSE"]
y = data["has_dementia?"]

seaborn.regplot(x=x,y=y,data=data,logistic=True,ci=None, line_kws={'color': 'green'}, scatter_kws={'color':
```

```
<AxesSubplot:xlabel='MMSE', ylabel='has_dementia?'>
```



Here we can clearly see, through analysis of the data that a MMSE level below 25 typically means the patient has some form of dementia.

Machine learning with SKlearn

After creating the new has_dementia? column I imported LogisticRegression from sklearn and fit a new logistic regression model with the has_dementia column on the y axis and the rest of the data on the x axis. By doing this I would be able to check what can lead to dementia in patients. Logistic regression would be quite good for this as it is easy to train and would allow me to predict whether a new patient could have some form of dementia or not off the rest of the data.

I tweaked the logistic regression params a few times via trial and error until I found what appeared to be the best score by changing the max iteration to 4009. The score that I ended up achieving with this machine learning model was 0.90.

```
logistic_reg = lr(multi_class="ovr", max_iter=4009).fit(x_data, y_target)
```

```
logistic_reg.score(x_data,y)
```

```
0.8954423592493298
```

After this I decided to create a report which would show the different scores that the technique managed to get for both Boolean targets. This would further allow me to see the performance of the model for any future predictions.

```
predicted = logistic_reg.predict(x_data)

report = metrics.classification_report(y, predicted)

print(report)
```

```
              precision    recall  f1-score   support

       False       0.88      0.93      0.91       206
        True       0.91      0.85      0.88       167

    accuracy                           0.90       373
   macro avg       0.90      0.89      0.89       373
weighted avg       0.90      0.90      0.90       373
```

Lastly, I decided to use cross_val_score from sklearns library to measure different scores of the model when cross validation is applied, this is an evaluation process that allows us to measure the performance of the model on unseen data. I decided to use 8 folds for the cross validation, and used 2 different scoring methods, being accuracy and f1, to see how they both performed. I then calculated the mean for each scoring method which produced the scores below:

```
cross_val_score(lr(max_iter=4009, multi_class="ovr"), x_data, y_target, scoring="f1", cv=8).mean()
```

```
0.8681560373857444
```

```
cross_val_score(lr(max_iter=4009, multi_class="ovr"), x_data, y_target, scoring="accuracy", cv=8).mean()
```

```
0.8873149861239593
```

I made sure that the distribution of the scores was even, meaning the model is robust.

As we can see, the scores dropped slightly when compared to before, meaning more error, however they are still quite high. This shows us that the model is reproducible and can be used to predict whether a new patient could have dementia.

Decision Trees

The next machine learning model that I decided to develop was a decision tree that once again had has_dementia? as the target. The reason for this being that a decision tree would allow me to visualise exactly what element/column combinations lead to the Boolean either being true or false, predicting whether a patient might have dementia.

Fitting a model and cross validation

I started off by instantiating the decision tree classifier and fitting the data, I then created a metrics classification report just so that I could get a general gist of the scores that I was getting.

```
dt = tree.DecisionTreeClassifier(max_depth=3)

predicted = dt.fit(x,y).predict(x)

report = metrics.classification_report(y, predicted)

print(report)
```

```
              precision    recall  f1-score   support

       False       0.79      0.96      0.87       206
        True       0.93      0.69      0.79       167

    accuracy                           0.84       373
   macro avg       0.86      0.82      0.83       373
weighted avg       0.85      0.84      0.83       373
```

I then wanted a more reproducible accuracy score, one that included training/testing of the data, so I decided to utilise the cross_val_score function and divided the data up into 5 folds, then I calculated the mean. This allows me to measure how successful this model would be if new data was entered into the frame.

```
cross_val_score(dt, x, y, scoring="accuracy", cv=5).mean()
```

```
0.780072072072072
```

I made sure that the distribution of the scores was even, meaning the model is robust.

Visualisation of the tree prediction

I then was able to visualise the tree's rigging for prediction by importing tree from sklearn's library, I made a new export text function that took in the model I previously created.

```
text = tree.export_text(dt, feature_names=x.columns.tolist())

print(text)
```

```
|--- MMSE <= 27.50
|   |--- MMSE <= 26.50
|   |   |--- nWBV <= 0.76
|   |   |   |--- class: True
|   |   |--- nWBV >  0.76
|   |   |   |--- class: True
|   |--- MMSE >  26.50
|   |   |--- ASF <= 1.35
|   |   |   |--- class: True
|   |   |--- ASF >  1.35
|   |   |   |--- class: False
|--- MMSE >  27.50
|   |--- sex_num <= 0.50
|   |   |--- MMSE <= 28.50
|   |   |   |--- class: False
|   |   |--- MMSE >  28.50
|   |   |   |--- class: False
|   |--- sex_num >  0.50
|   |   |--- ASF <= 1.29
|   |   |   |--- class: False
|   |   |--- ASF >  1.29
|   |   |   |--- class: True
```

The text shows the various columns, the max depth is set to 2 so it shows the most relevant columns that the decision tree uses to predict future data. For example, one part of the tree shows that when the MMSE is larger than 26.50 and the ASF is less than or equal to 1.35, the patient may have dementia. However when the ASF is larger than 1.35, the model predicts that the patient would not have dementia.

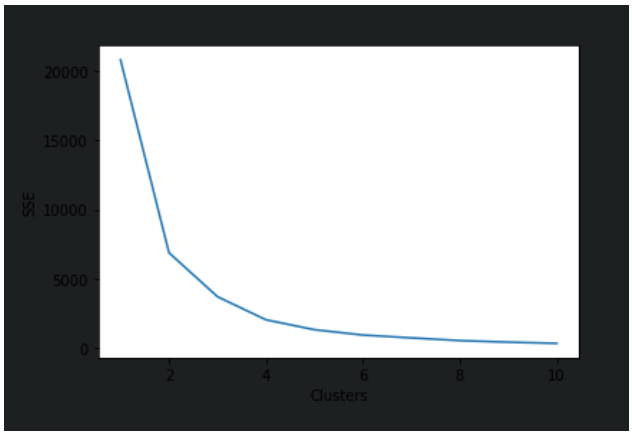
Clustering

I decided to set up a clustering model to measure the correlation between age and nWBV, as I was interested to see how age impacted patients brain volume. I believe that clustering is a good idea to visualize this as it would allow me to split the data and see the difference in nWBV levels in different groups of ages.

The elbow method

To determine the number of clusters that the data set would need, I used the elbow method. This method helps me to find the optimal amount of clusters needed in the scatterplot.

```
wcss = []
for i in range(1,11):
    k_means = KMeans(n_clusters=i,init='k-means++', random_state=42)
    k_means.fit(elbow_check)
    wcss.append(k_means.inertia_)
#plot elbow curve
plt.plot(numpy.arange(1,11),wcss)
plt.xlabel('Clusters')
plt.ylabel('SSE')
plt.show()
```



Wherever the line starts to bend is the optimal cluster number

Setting up and plotting the cluster

Once I had worked out the correct number of clusters that I would need, I began setting up the KMeans cluster and the data that was going into it using the km.fit_predict function.

```
km = KMeans(n_clusters=2)
km
```

```
KMeans(n_clusters=2)
```

```
y_predicted = km.fit_predict(data[['age','nWBV']])
data['cluster'] = y_predicted
data.head()
```

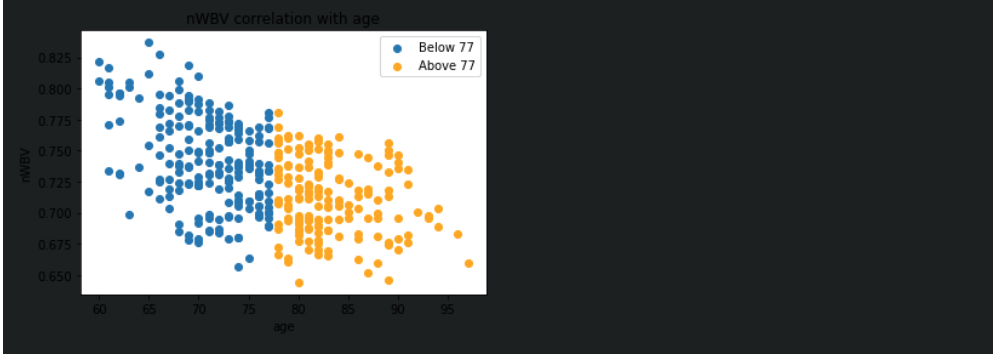
After setting up the cluster, the last thing to do was split the cluster into two separate variables, representing each cluster then plot them:

```
data1 = data[data.cluster == 0]
data2 = data[data.cluster == 1]

plt.scatter(data1.age, data1["nWBV"])
plt.scatter(data2.age, data2["nWBV"], color='orange')

plt.ylabel('nWBV')
plt.xlabel('age')
plt.legend(["Below 77", "Above 77"])
plt.title("nWBV correlation with age")
```

```
Text(0.5, 1.0, 'nWBV correlation with age')
```



After looking at this predicted data we can see that the general trend is that nWBV does decrease with age slightly, in between the ages 60 – 77 quite a few patients have a volume above 0.800, however when the patient is above 77 it's almost guaranteed that their nWB volume is not above 0.775.

Concluding thoughts

Overall I really enjoyed this part of the module, gaining an insight into such an interesting field of study has piqued my interest and I am definitely going to continue looking into machine learning and AI. Out of the three models I developed I would say that logistic regression was my favourite. This was due to how easy I found it to implement and interpret, it also didn't require much tweaking of the functions parameters. I feel as though this model would really help to identify dementia in patients. One area that I feel I could develop and improve is the cluster model. The reason I feel as though there is room for development is because it is a fairly simple model, maybe I could develop it a bit more and turn it into a 3D scatter plot that also takes in the different CDR levels in patients. This would help to see how nWBV levels correlate to the different CDR levels alongside the age of the patient.