Mini-Project2

April 8, 2024

[293]:
```
import math
import pickle
import gzip
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
%matplotlib inline
```

1 Mini-Project 2

1.) Topic

This project will explore PCA and KMeans clustering on the Darwin dataset, which is a dataset of handwriting data from subjects either diagnosed with Alzheimer's or not. We will use Logistic Regression on the dataset after doing PCA. In addition we will predict from the unsupervised K-means clustering based on our best permutation of labels.

2.) Data

Here is a link to the dataset from the UCI repository: https://archive.ics.uci.edu/dataset/732/darwin

I couldn't find clear descriptions of the features within the repository, so I had to do some additional searching to find out a little more on the features. A detailed description of the tasks the subjects completed can be found here: Link

The subjects were tasked with 25 different handwriting tasks. The study used paper overlayed on a graphic tablet to track the pen motions and pressure applied by the subjects. Below you can see the column names for the first task. There were 25 tasks, as there are 18 features per task we end up with 450 features.

[314]:
```
from sklearn.preprocessing import StandardScaler
s = StandardScaler()
X = pd.read_csv('Data/data.csv')

X.iloc[:,0:19]
```

[314]:
```
          ID  air_time1  disp_index1  gmrt_in_air1  gmrt_on_paper1  \
0       id_1       5160     0.000013    120.804174       86.853334
```

1	id_2	51980	0.000016	115.318238	83.448681
2	id_3	2600	0.000010	229.933997	172.761858
3	id_4	2130	0.000010	369.403342	183.193104
4	id_5	2310	0.000007	257.997131	111.275889
..	…	…	…	…	…
169	id_170	2930	0.000010	241.736477	176.115957
170	id_171	2140	0.000009	274.728964	234.495802
171	id_172	3830	0.000008	151.536989	171.104693
172	id_173	1760	0.000008	289.518195	196.411138
173	id_174	2875	0.000008	235.769350	178.208024

	max_x_extension1	max_y_extension1	mean_acc_in_air1	mean_acc_on_paper1	\
0	957	6601	0.361800	0.217459	
1	1694	6998	0.272513	0.144880	
2	2333	5802	0.387020	0.181342	
3	1756	8159	0.556879	0.164502	
4	987	4732	0.266077	0.145104	
..	…	…	…	…	
169	1839	6439	0.253347	0.174663	
170	2053	8487	0.225537	0.174920	
171	1287	7352	0.165480	0.161058	
172	1674	6946	0.518937	0.202613	
173	1838	6560	0.567311	0.147818	

	mean_gmrt1	mean_jerk_in_air1	mean_jerk_on_paper1	mean_speed_in_air1	\
0	103.828754	0.051836	0.021547	1.828076	
1	99.383459	0.039827	0.016885	1.817744	
2	201.347928	0.064220	0.020126	3.378343	
3	276.298223	0.090408	0.021150	5.082499	
4	184.636510	0.037528	0.018590	3.804656	
..	…	…	…	…	
169	208.926217	0.032691	0.022786	4.074893	
170	254.612383	0.032059	0.019521	4.149653	
171	161.320841	0.022705	0.022441	2.041489	
172	242.964666	0.090686	0.023634	4.385103	
173	206.988687	0.099555	0.019198	4.898606	

	mean_speed_on_paper1	num_of_pendown1	paper_time1	pressure_mean1	\
0	1.493242	22	10730	1679.232060	
1	1.517763	11	12460	1723.171348	
2	3.308866	10	6080	1520.253289	
3	3.542645	10	5595	1913.995532	
4	2.180544	8	4080	1819.121324	
..	…	…	…	…	
169	3.390491	12	5835	1470.698372	
170	4.143594	10	4595	1880.668118	
171	3.507108	14	4060	1800.671182	

```
172             3.538417                  8        4425     1881.701695
173             2.945370                 12        4340     1860.744240

     pressure_var1  total_time1
0      288285.0449        15890
1      210516.6356        64440
2      120845.8717         8680
3      100286.6032         7725
4      160061.8198         6390
..             …           …
169    235194.3615         8765
170    155216.3567         6735
171    212575.8020         7890
172    109235.0387         6185
173    131142.3171         7215

[174 rows x 19 columns]
```

3.) Data Cleaning and EDA

If we use the Pandas Dataframe describe function we get some very useful data about our dataset and features. We can see the count of each column is 174, meaning there is data for all rows and columns, so we won't need to impute. There is some interesting percentile data as well for each of the features as well as mean, min and max values.

There are some features that could use scaling, for example the pressure_var25 feature could lead to some computation issues if it is multiplied many times. We will use sklearn's standard scaler to assist us in scaling and standardizing the dataset.

Below the describe table I am going to show a few histograms of the features means, so I had to do a fun list comprehension to get the indexes of columns that should be grouped together which is what makes python so great. An example of metrics that should be grouped together across tasks would be gmrt_in_air1, gmrt_in_air2 … ,gmrt_in_air25 .

We haven't removed the label column from our data frame yet, we will do that just after this so we can shuffle the data with the labels. Before creating our list of grouped feature indexes, we should go ahead and drop the ID column because it is of no use to us.

Please read on in the next cell labeled: "3.) Data Cleaning and EDA continued"

[315]:
```
X.describe()
```

[315]:
```
          air_time1  disp_index1  gmrt_in_air1  gmrt_on_paper1  \
count    174.000000   174.000000    174.000000      174.000000
mean    5664.166667     0.000010    297.666685      200.504413
std    12653.772746     0.000003    183.943181      111.629546
min       65.000000     0.000002     28.734515       29.935835
25%     1697.500000     0.000008    174.153023      136.524742
50%     2890.000000     0.000009    255.791452      176.494494
75%     4931.250000     0.000011    358.917885      234.052560
```

```
max    109965.000000     0.000028   1168.328276      865.210522

       max_x_extension1  max_y_extension1  mean_acc_in_air1  \
count        174.000000        174.000000        174.000000
mean        1977.965517       7323.896552          0.416374
std         1648.306365       2188.290512          0.381837
min          754.000000        561.000000          0.067748
25%         1362.500000       6124.000000          0.218209
50%         1681.000000       6975.500000          0.275184
75%         2082.750000       8298.500000          0.442706
max        18602.000000      15783.000000          2.772566

       mean_acc_on_paper1  mean_gmrt1  mean_jerk_in_air1  …  mean_gmrt25  \
count          174.000000  174.000000         174.000000  …   174.000000
mean             0.179823  249.085549           0.067556  …   221.360646
std              0.064693  132.698462           0.074776  …    63.762013
min              0.096631   41.199445           0.011861  …    69.928033
25%              0.146647  161.136182           0.029523  …   178.798382
50%              0.163659  224.445268           0.039233  …   217.431621
75%              0.188879  294.392298           0.071057  …   264.310776
max              0.627350  836.784702           0.543199  …   437.373267

       mean_jerk_in_air25  mean_jerk_on_paper25  mean_speed_in_air25  \
count          174.000000            174.000000           174.000000
mean             0.148286              0.019934             4.472643
std              0.062207              0.002388             1.501411
min              0.030169              0.014987             1.323565
25%              0.107732              0.018301             3.485934
50%              0.140483              0.019488             4.510578
75%              0.199168              0.021134             5.212794
max              0.375078              0.029227            10.416715

       mean_speed_on_paper25  num_of_pendown25   paper_time25  \
count             174.000000        174.000000     174.000000
mean                2.871613         85.839080   43109.712644
std                 0.852809         27.485518   19092.024337
min                 0.950249         32.000000   15930.000000
25%                 2.401199         66.000000   32803.750000
50%                 2.830672         81.000000   37312.500000
75%                 3.335828        101.500000   46533.750000
max                 5.602909        209.000000  139575.000000

       pressure_mean25  pressure_var25  total_time25
count       174.000000      174.000000  1.740000e+02
mean       1629.585962   163061.767360  1.642033e+05
std         324.142316    56845.610814  4.969397e+05
min         474.049462    26984.926660  2.998000e+04
```

```
25%       1499.112088  120099.046800  5.917500e+04
50%       1729.385010  158236.771800  7.611500e+04
75%       1865.626974  200921.078475  1.275425e+05
max       1999.775983  352981.850000  5.704200e+06

[8 rows x 450 columns]
```

[316]:
```
X = X.drop(['ID'], axis=1)
```

[317]:
```
l1 = [[y + x*18 for x in range(25)] for y in range(18)] #
l1
```

[317]:
```
[[0,
  18,
  36,
  54,
  72,
  90,
  108,
  126,
  144,
  162,
  180,
  198,
  216,
  234,
  252,
  270,
  288,
  306,
  324,
  342,
  360,
  378,
  396,
  414,
  432],
 [1,
  19,
  37,
  55,
  73,
  91,
  109,
  127,
  145,
  163,
```

```
 181,
 199,
 217,
 235,
 253,
 271,
 289,
 307,
 325,
 343,
 361,
 379,
 397,
 415,
 433],
[2,
 20,
 38,
 56,
 74,
 92,
 110,
 128,
 146,
 164,
 182,
 200,
 218,
 236,
 254,
 272,
 290,
 308,
 326,
 344,
 362,
 380,
 398,
 416,
 434],
[3,
 21,
 39,
 57,
 75,
 93,
 111,
```

```
 129,
 147,
 165,
 183,
 201,
 219,
 237,
 255,
 273,
 291,
 309,
 327,
 345,
 363,
 381,
 399,
 417,
 435],
[4,
 22,
 40,
 58,
 76,
 94,
 112,
 130,
 148,
 166,
 184,
 202,
 220,
 238,
 256,
 274,
 292,
 310,
 328,
 346,
 364,
 382,
 400,
 418,
 436],
[5,
 23,
 41,
 59,
```

```
 77,
 95,
 113,
 131,
 149,
 167,
 185,
 203,
 221,
 239,
 257,
 275,
 293,
 311,
 329,
 347,
 365,
 383,
 401,
 419,
 437],
[6,
 24,
 42,
 60,
 78,
 96,
 114,
 132,
 150,
 168,
 186,
 204,
 222,
 240,
 258,
 276,
 294,
 312,
 330,
 348,
 366,
 384,
 402,
 420,
 438],
[7,
```

```
 25,
 43,
 61,
 79,
 97,
 115,
 133,
 151,
 169,
 187,
 205,
 223,
 241,
 259,
 277,
 295,
 313,
 331,
 349,
 367,
 385,
 403,
 421,
 439],
[8,
 26,
 44,
 62,
 80,
 98,
 116,
 134,
 152,
 170,
 188,
 206,
 224,
 242,
 260,
 278,
 296,
 314,
 332,
 350,
 368,
 386,
 404,
```

```
 422,
 440],
[9,
 27,
 45,
 63,
 81,
 99,
 117,
 135,
 153,
 171,
 189,
 207,
 225,
 243,
 261,
 279,
 297,
 315,
 333,
 351,
 369,
 387,
 405,
 423,
 441],
[10,
 28,
 46,
 64,
 82,
 100,
 118,
 136,
 154,
 172,
 190,
 208,
 226,
 244,
 262,
 280,
 298,
 316,
 334,
 352,
```

```
 370,
 388,
 406,
 424,
 442],
[11,
 29,
 47,
 65,
 83,
 101,
 119,
 137,
 155,
 173,
 191,
 209,
 227,
 245,
 263,
 281,
 299,
 317,
 335,
 353,
 371,
 389,
 407,
 425,
 443],
[12,
 30,
 48,
 66,
 84,
 102,
 120,
 138,
 156,
 174,
 192,
 210,
 228,
 246,
 264,
 282,
 300,
```

```
 318,
 336,
 354,
 372,
 390,
 408,
 426,
 444],
[13,
 31,
 49,
 67,
 85,
 103,
 121,
 139,
 157,
 175,
 193,
 211,
 229,
 247,
 265,
 283,
 301,
 319,
 337,
 355,
 373,
 391,
 409,
 427,
 445],
[14,
 32,
 50,
 68,
 86,
 104,
 122,
 140,
 158,
 176,
 194,
 212,
 230,
 248,
```

```
 266,
 284,
 302,
 320,
 338,
 356,
 374,
 392,
 410,
 428,
 446],
[15,
 33,
 51,
 69,
 87,
 105,
 123,
 141,
 159,
 177,
 195,
 213,
 231,
 249,
 267,
 285,
 303,
 321,
 339,
 357,
 375,
 393,
 411,
 429,
 447],
[16,
 34,
 52,
 70,
 88,
 106,
 124,
 142,
 160,
 178,
 196,
```

```
  214,
  232,
  250,
  268,
  286,
  304,
  322,
  340,
  358,
  376,
  394,
  412,
  430,
  448],
 [17,
  35,
  53,
  71,
  89,
  107,
  125,
  143,
  161,
  179,
  197,
  215,
  233,
  251,
  269,
  287,
  305,
  323,
  341,
  359,
  377,
  395,
  413,
  431,
  449]]
```

3.) Data Cleaning and EDA continued

If you expand the cell above you can see our 18 arrays of 25 indexes that should be grouped for the histograms.

Let's use those in the next cell to pull in the means from each of those columns,I'm just going to do an arbitrary one as I don't want to show 18 histograms, but you could easily loop through the array of arrays and print out 18 historgrams if so inclined. We can do this easily by just making a new dataframe from the existing with just the columns we want. Then we can use pandas to calculate the mean of each column. This will be the data for our mean histogram for disp_index and air_time.

[318]:
```
dspmn = X[X.columns[l1[1]]]
display(dspmn)

gmrtmn = X[X.columns[l1[2]]]
display(gmrtmn)
```

```
     disp_index1  disp_index2  disp_index3  disp_index4  disp_index5  \
0       0.000013     0.000012     0.000013     0.000027     0.000016
1       0.000016     0.000015     0.000013     0.000045     0.000034
2       0.000010     0.000010     0.000009     0.000023     0.000014
3       0.000010     0.000014     0.000014     0.000026     0.000016
4       0.000007     0.000012     0.000012     0.000027     0.000003
..           …            …            …            …            …
169     0.000010     0.000012     0.000007     0.000027     0.000015
170     0.000009     0.000010     0.000010     0.000025     0.000015
171     0.000008     0.000012     0.000011     0.000027     0.000015
172     0.000008     0.000011     0.000012     0.000027     0.000015
173     0.000008     0.000011     0.000010     0.000027     0.000015

     disp_index6  disp_index7  disp_index8  disp_index9  disp_index10  …  \
0       0.000003     0.000004     0.000006     0.000004      0.000006  …
1       0.000009     0.000007     0.000006     0.000007      0.000008  …
2       0.000005     0.000005     0.000007     0.000010      0.000005  …
3       0.000003     0.000005     0.000009     0.000007      0.000005  …
4       0.000004     0.000005     0.000006     0.000007      0.000005  …
..           …            …            …            …            …  …
169     0.000006     0.000005     0.000007     0.000009      0.000004  …
170     0.000007     0.000010     0.000007     0.000012      0.000009  …
171     0.000004     0.000007     0.000007     0.000011      0.000007  …
172     0.000004     0.000007     0.000007     0.000011      0.000007  …
173     0.000004     0.000007     0.000007     0.000011      0.000007  …

     disp_index16  disp_index17  disp_index18  disp_index19  disp_index20  \
0        0.000005      0.000025      0.000004      0.000028      0.000014
1        0.000005      0.000030      0.000006      0.000033      0.000051
2        0.000000      0.000029      0.000004      0.000030      0.000014
3        0.000005      0.000026      0.000004      0.000030      0.000016
4        0.000003      0.000019      0.000004      0.000028      0.000012
..            …             …             …             …             …
169      0.000005      0.000027      0.000005      0.000034      0.000014
170      0.000004      0.000031      0.000005      0.000034      0.000013
171      0.000004      0.000031      0.000005      0.000034      0.000018
172      0.000004      0.000031      0.000005      0.000034      0.000018
173      0.000004      0.000031      0.000005      0.000034      0.000018
```

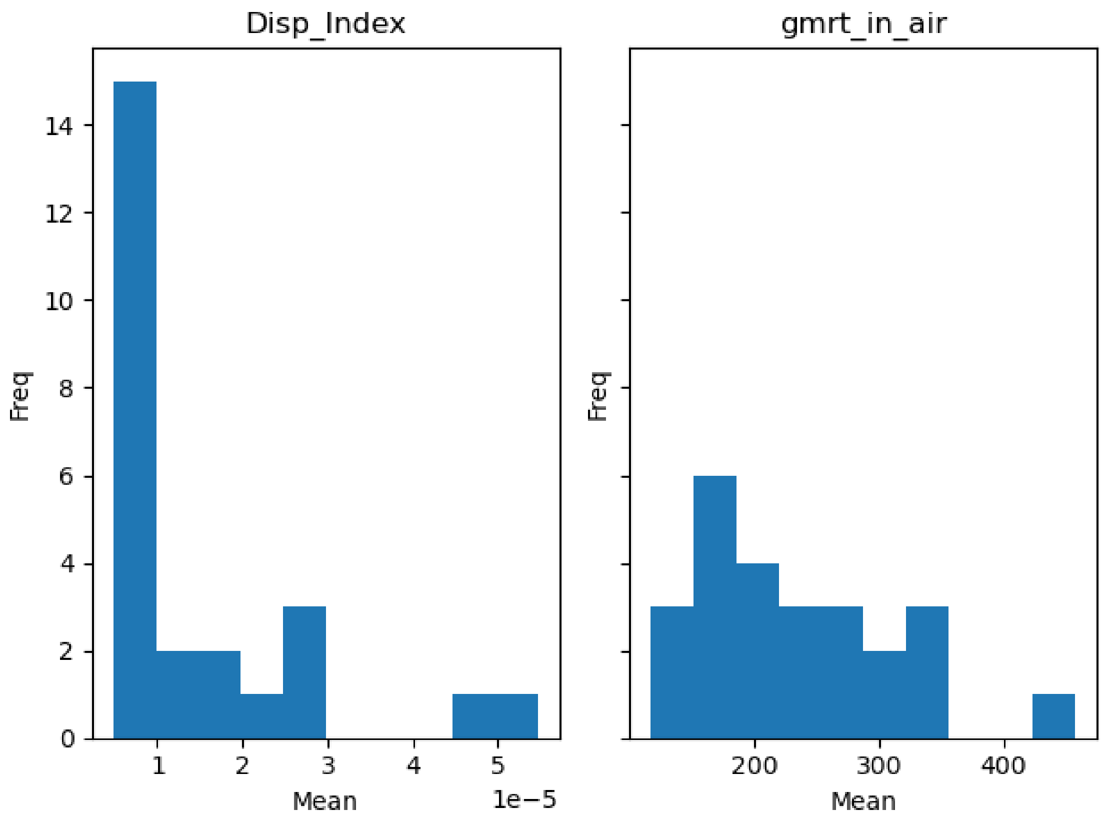
```
     disp_index21  disp_index22  disp_index23  disp_index24  disp_index25
0        0.000061      0.000009      0.000009      0.000019      0.000049
1        0.000036      0.000012      0.000011      0.000016      0.000070
2        0.000049      0.000010      0.000011      0.000022      0.000056
3        0.000058      0.000009      0.000010      0.000021      0.000058
4        0.000062      0.000008      0.000008      0.000021      0.000043
..            …            …            …            …            …
169      0.000050      0.000008      0.000009      0.000021      0.000051
170      0.000053      0.000009      0.000009      0.000020      0.000056
171      0.000053      0.000011      0.000010      0.000025      0.000059
172      0.000053      0.000011      0.000010      0.000025      0.000059
173      0.000053      0.000011      0.000010      0.000025      0.000059

[174 rows x 25 columns]
     gmrt_in_air1  gmrt_in_air2  gmrt_in_air3  gmrt_in_air4  gmrt_in_air5  \
0      120.804174    269.355789    330.487573    232.639907    125.989768
1      115.318238    272.771237    240.382372     68.602606    151.022535
2      229.933997    122.809584     82.562392    107.348186    101.628410
3      369.403342    185.278506    323.273194    219.338837    212.248278
4      257.997131    112.619685    214.958135    204.322750    134.501527
..            …            …            …            …            …
169    241.736477    244.639443    205.543659     91.302401     92.660767
170    274.728964    326.445362    461.017026    327.269006    106.507208
171    151.536989    303.522001    428.020268     63.105656    102.308276
172    289.518195    511.188113    385.069812     63.105656    102.308276
173    235.769350    511.188113    405.584055     63.105656    102.308276

     gmrt_in_air6  gmrt_in_air7  gmrt_in_air8  gmrt_in_air9  gmrt_in_air10  \
0      244.192448    390.168619     98.738919   1331.340031     136.057361
1      130.260917    133.477130     85.904279     67.345719      70.232324
2      156.512289    273.908579    936.464720    299.546106     201.996739
3      214.783769    162.139813    121.386016    237.756310     149.103145
4      399.201436    420.384148    198.886419    210.098311     127.444184
..            …            …            …            …             …
169    407.986542    319.107406    193.202877    292.564378     351.985247
170    361.434999    392.816132    193.202877    478.666791     176.547172
171     87.324640    139.288717    193.202877    178.700835     196.224808
172     87.324640    139.288717    193.202877    178.700835     196.224808
173     87.324640    139.288717    193.202877    178.700835     196.224808

     …  gmrt_in_air16  gmrt_in_air17  gmrt_in_air18  gmrt_in_air19  \
0    …     152.892221     233.520832     154.316105     161.441145
1    …     125.217229     114.279952      90.701274     103.550797
2    …     255.367051     355.428038     167.812304     160.468599
3    …     267.084079     262.309253     412.731557     134.595114
4    …     185.464449     318.563434     181.697644     119.828032
```

```
..   …          …          …          …          …
169  …   318.348482  341.258753  356.548511   95.346818
170  …   181.009497  360.743997  356.548511   95.346818
171  …   213.848238  360.743997  356.548511   95.346818
172  …   213.848238  360.743997  356.548511   95.346818
173  …   213.848238  360.743997  356.548511   95.346818

     gmrt_in_air20  gmrt_in_air21  gmrt_in_air22  gmrt_in_air23  \
0       175.234228     251.622971     141.179667     126.658709
1       185.609249     290.800361      64.153361      85.909291
2       124.719850     703.155498     147.953226     117.765304
3       235.163095     451.279786     137.080909     211.397750
4       187.584869     288.615548     129.442167     104.338067
..             …              …              …              …
169     204.739473     404.320478     220.482567     249.819905
170     198.244935     305.367999     133.312847     247.462723
171     243.500435     305.367999      97.473341     249.977087
172     243.500435     305.367999      97.473341     249.977087
173     243.500435     305.367999      97.473341     249.977087

     gmrt_in_air24  gmrt_in_air25
0       218.093767     279.628181
1        79.502263      86.117902
2       110.209716     215.379542
3       101.823953     207.557650
4        91.467337     167.510556
..             …              …
169     114.441189     232.999622
170     289.535604     250.394568
171     141.325005     183.261091
172     141.325005     183.261091
173     141.325005     183.261091

[174 rows x 25 columns]
```

[319]:
```python
dspMean = dspmn.mean(axis=0).to_numpy()
gmrtMean = gmrtmn.mean(axis=0).to_numpy()
```

[320]:
```python
fig, axs = plt.subplots(1, 2,sharey=True, tight_layout=True)


axs[0].hist(dspMean)
axs[0].set_xlabel('Mean')
axs[0].set_ylabel('Freq')
axs[0].title.set_text('Disp_Index')

axs[1].hist(gmrtMean)
```

```
axs[1].set_xlabel('Mean')
axs[1].set_ylabel('Freq')
axs[1].title.set_text('gmrt_in_air')
```



So to conclude our EDA we will shuffle our data and then pull our label data out and apply sklearn's standard scaler. We pull the target data our after shuffling so we can keep the order when we split the data into training and test data later.

[321]:
```
X[0:5]
X = X.sample(frac=1)
```

[322]:
```
npt = X['class'].to_numpy()

X = X.drop(['class'], axis=1)

X = s.fit_transform(X)

X
```

[322]:
```
array([[-0.2824812 ,  0.67005885,  0.58067291, …,  0.68059489,
         0.01161489, -0.19097171],
```

```
[-0.37362547, -0.28743257,  0.00608449, …,  0.32867771,
  1.21002067,  0.91310364],
[ 0.57823776,  2.63873278, -0.78277519, …, -0.14248587,
  0.04551975, -0.1365429 ],
…,
[-0.36966268, -0.68713303,  0.54116553, …,  0.92729398,
 -1.21182576, -0.2119602 ],
[ 0.24338163,  0.55074528,  0.1291487 , …,  0.84618712,
 -1.22320833,  0.35761497],
[ 0.00838791, -0.40674614, -0.90324148, …, -0.40951536,
  0.2262976 , -0.16547271]])
```

[323]:
```
X.shape
```

[323]:
```
(174, 450)
```

4.) Model Building

So the focus of this project for me is really the dimensionality reduction and clustering moreso than model building; however, we will still build two models. We will use logistic regression(actually classification) after we perform PCA on the dataset. After that we will use KMeans clustering on the original dataset to see how the algorithm clusters the data into two groups. We will use the implicitly as a model for predictions, we should note that this is still an unsupervised algorithm, i.e. it knows nothing about the labels so hopefully the clusters will implicitly be our labels.

Below is my PCA implementation from Homework 6. I did have to make an adjustment to it, on the homework assignment I used the Numpy.linalg.eig function to compute the eigen_pairs, when I ran this initially for the DARWIN dataset I got complex numbers in the eigen_pairs. Switching the algorithm to the numpy.linalg.eigh resolved this issue as it is meant for Real Symmetric matrices which is what our covariance matrix that we compute should be. This made the overall algorithm much faster as well.

[324]:
```
from sklearn.preprocessing import StandardScaler

class PCA1:
    def __init__(self, target_explained_variance=None):
        """
        explained_variance: float, the target level of explained variance
        """
        self.target_explained_variance = target_explained_variance
        self.feature_size = -1

    def standardize(self, X):
        """
        standardize features using standard scaler
        :param X: input data with shape m (# of observations) X n (# of␣
 ↪features)
        :return: standardized features (Hint: use skleanr's StandardScaler.)
```

```
        """
        # Q1. Standardize X using sklearn's StandardScaler. Read the␣
↪documentation's example. https://scikit-learn.org/stable/modules/generated/
↪sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.
↪StandardScaler
        # Hint: In the example, they used .fit() and .transform() methods␣
↪separately. You can use another method that has both fit and transform.
        # YOUR CODE HERE
        scaler = StandardScaler()

        return scaler.fit_transform(X)



    def compute_mean_vector(self, X_std):
        """
        compute mean vector
        :param X_std: transformed data
        :return mean vector of shape (# of features,)
        """
        # Q2. return mean vector of shape (# of features,). Hint: averaging␣
↪over the rows for each column.
        # YOUR CODE HERE
        fl = X_std.shape[1]
        mv = np.empty((fl,))
        i = 0

        for col in X_std.T:
            mv[i] = np.mean(col) #Don't need to provide axis as a col(row since␣
↪transposed) should only have one axis
            i += 1

        return mv

    def compute_cov(self, X_std, mean_vec):
        """
        Covariance using mean, (don't use any numpy.cov)
        :param X_std:
        :param mean_vec:
        :return n X n matrix:: covariance matrix
        """
        # Q3. Caculate covariance matrix https://en.wikipedia.org/wiki/
↪Covariance_matrix
        # Hint: E[(X-mu)T(X-mu)]. You can assume equal probability when␣
↪calculating the expected value.
        # YOUR CODE HERE
```

```
        CVM = (1 / (X_std.shape[0] - 1))*((X_std - mean_vec).T.dot(X_std -␣
↪mean_vec))
        print(CVM)

        return CVM

    def compute_eigen_vector(self, cov_mat):
        """
        Eigenvector and eigen values using numpy. Uses numpy's eigenvalue␣
↪function
        :param cov_mat:
        :return: (eigen_values, eigen_vector)
        """
        # Q4. Return eigenvalues and engenvectors.
        # Hint: Use appropriate function in linalg package. https://numpy.org/
↪doc/stable/reference/routines.linalg.html
        # YOUR CODE HERE
        #print(cov_mat.shape)
        eigen_vals, eigen_vecs = np.linalg.eigh(cov_mat, )

        #eigen_vals = np.array([float(x.real) for x in eigen_vals])
        #print(eigen_vecs.shape)
        return eigen_vals, eigen_vecs

    def compute_explained_variance(self, eigen_vals):
        """
        Q5. Sort eigen values and compute explained variance ratio.
        explained variance informs the amount of information (variance)
        can be attributed to each of  the principal components.
        :param eigen_vals:
        :return: explained variance ratio.
        """
        # YOUR CODE HERE
        lenEV = np.array(eigen_vals).shape[0]
        eigen_vals[::-1].sort()
        totvar = sum(eigen_vals)
        evr = np.empty((lenEV,))
        j = 0

        for v in eigen_vals:
            evr[j] = v / totvar
            j += 1

        #print(evr)
        return evr
```

```
    def cumulative_sum(self, var_exp):
        """
        return cumulative sum of explained variance.
        :param var_exp: explained variance ratio
        :return: cumulative explained variance ratio
        """
        return np.cumsum(var_exp)

    def compute_weight_matrix(self, eig_pairs, cum_var_exp):
        """
        compute weight matrix of top principal components conditioned on target
        explained variance.
        (Hint : use cumilative explained variance ratio and␣
↪target_explained_variance to find
        top components)

        :param eig_pairs: list of tuples containing eigenvalues and␣
↪eigenvectors,
        sorted by eigenvalues in descending order (the biggest eigenvalue and␣
↪corresponding eigenvectors first).
        :param cum_var_exp: cumulative expalined variance by features
        :return: weight matrix (the shape of the weight matrix is n X k)
        """
        matrix_w = np.ones((self.feature_size, 1))
        # Q6. In this function, you will implement weight matrix calculation.
        # For each iteration, check the cumulative explained variance ratio␣
↪compared to the target explained variance (see the init vairables)
        # then add the eigenvector as column or the matrix_w above.
        # matrix_w will have the dimension of (784,1) initially, but each␣
↪iteration the column will be added until
        # the cumulative explained variance reaches the target explained␣
↪variance.
        # YOUR CODE HERE
        #print(matrix_w.shape)
        #print(eig_pairs[0][1].reshape(-1,1))
        #print(eig_pairs[0][1].T.shape)
        #print(eig_pairs[0])
        #print(matrix_w)
        matrix_w = np.concatenate((matrix_w, eig_pairs[0][1].reshape(-1,1)),␣
↪axis=1)

        csi = 1
        while (cum_var_exp[csi] <= self.target_explained_variance):
            matrix_w = np.concatenate((matrix_w, eig_pairs[csi][1].
↪reshape(-1,1)), axis=1)
            csi += 1
```

```
        #print(matrix_w)
        return matrix_w


    def transform_data(self, X_std, matrix_w):
        """
        transform data to subspace using weight matrix
        :param X_std: standardized data
        :param matrix_w: weight matrix
        :return: data in the subspace
        """
        return X_std.dot(matrix_w)

    def fit(self, X):
        """
        entry point to the transform data to k dimensions
        standardize and compute weight matrix to transform data.
        The fit functioin returns the transformed features. k is the number of␣
↪features which cumulative
        explained variance ratio meets the target_explained_variance.
        :param   m X n dimension: train samples
        :return  m X k dimension: subspace data.
        """

        self.feature_size = X.shape[1]


        # Multisteps to appomplish the fit function- 16 pts
        # step 1. Standardize X to X_std using an appropriate function you␣
↪implemented above.
        # X_std = (complete this part)
        # YOUR CODE HERE
        X_std = self.standardize(X)

        # step 2. get mean_vec and cov_mat from the appropriate functions from␣
↪above implementations
        # mean_vec =
        # cov_mat =
        # YOUR CODE HERE
        mean_vec = self.compute_mean_vector(X_std)
        cov_mat =  self.compute_cov(X_std, mean_vec)

        # step 3. get eigenvalues and eigenvectors from the implemented␣
↪function above.
        # eig_vals, eig_vecs =
        # YOUR CODE HERE
```

```
        eig_vals, eig_vecs = self.compute_eigen_vector(cov_mat)

        # step 4. Sort both eig_vals and eig_vecs by descending order in␣
↪eigenvalues.
        # For example, the first 5 elements of the sorted eigenvalues would␣
↪look like array([170.57702751, 112.84212831,  45.10927112,  40.75001861, 32.
↪99731063])
        # and reorder the eigenvector list accordingly.
        # Make a list of tuple called eig_pairs
        # eig_pairs = [(170.577, the first eigenvector), (112.84, the second␣
↪eigenvector), ...] (the length of this list is 784)
        # Each eigenvector has a dimension of (784,)
        # YOUR CODE HERE
        eig_pairs = list(zip(eig_vals,eig_vecs))
        eig_pairs.sort(key=lambda x: x[0], reverse=True)


        # step 5. get explained variance ratio and cumulated explained variance␣
↪ratio using functions implemented above.
        # Use the variable names below.
        # var_exp =
        # cum_var_exp =
        # YOUR CODE HERE
        var_exp = self.compute_explained_variance(eig_vals)
        cum_var_exp = self.cumulative_sum(var_exp)

        # This step calculates the matrix_w
        matrix_w = self.
↪compute_weight_matrix(eig_pairs=eig_pairs,cum_var_exp=cum_var_exp)

        print(len(matrix_w),len(matrix_w[0]))
        return self.transform_data(X_std=X_std, matrix_w=matrix_w)
```

Next we run the PCA algorithm:

[325]:
```
pca_handler = PCA1(target_explained_variance=0.9)
X_train_updated = pca_handler.fit(X)
```

```
[[ 1.00578035  0.36334712 -0.23407708 … -0.04146479  0.09154482
   0.0188034 ]
 [ 0.36334712  1.00578035 -0.24779655 … -0.11254862  0.06697624
   0.09853628]
 [-0.23407708 -0.24779655  1.00578035 …  0.08944409 -0.12053273
  -0.08482268]
 …
 [-0.04146479 -0.11254862  0.08944409 …  1.00578035 -0.14329341
  -0.09395363]
```

```
 [ 0.09154482  0.06697624 -0.12053273 … -0.14329341  1.00578035
   0.12500912]
 [ 0.0188034   0.09853628 -0.08482268 … -0.09395363  0.12500912
   1.00578035]]
450 79
```

[326]:
```python
X_train_updated.shape
```

[326]:
```
(174, 79)
```

We have reduced our dimension from 450 to 79. As we can see above our updated data set keeps the same number of rows but the columns have decreased to 54. Order of the rows has been preserved. For a smell test on whether or not our PCA implementation is ok we can compare it to sklearn's PCA algorithm:

[327]:
```python
from sklearn.decomposition import PCA

pca = PCA(n_components=0.9)
principal_components = pca.fit_transform(X)
```

[328]:
```python
principal_components.shape
```

[328]:
```
(174, 79)
```

As we can see we get the exact same number of dimensions!

Before fitting a logistic regression model, we need to convert the Prediction labels from "H"(Healthy) to 0 and "P"(Patient(Diagnosed with Alzheimer's disease)) to 1

[329]:
```python
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y = le.fit_transform(npt)
```

Now we can split the data into train and test sets.

[330]:
```python
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X_train_updated, y,␣
 ↪test_size=.3)
```

Now that we have our splits, let's run it through a quick model to see if we can get any decent predictability. Let's use Logistic Regression as a baseline model to use after performing Unsupervised learning of some sort, in this case PCA.

5.) Model Training

Below we train a logistic regression model and use it to predict on the test set.

[331]:
```
from sklearn.linear_model import LogisticRegression

LogReg = LogisticRegression(max_iter=10000)
LogReg.fit(X_train, Y_train)

ypred = LogReg.predict(X_test)
```

[332]:
```
Y_test, ypred
```

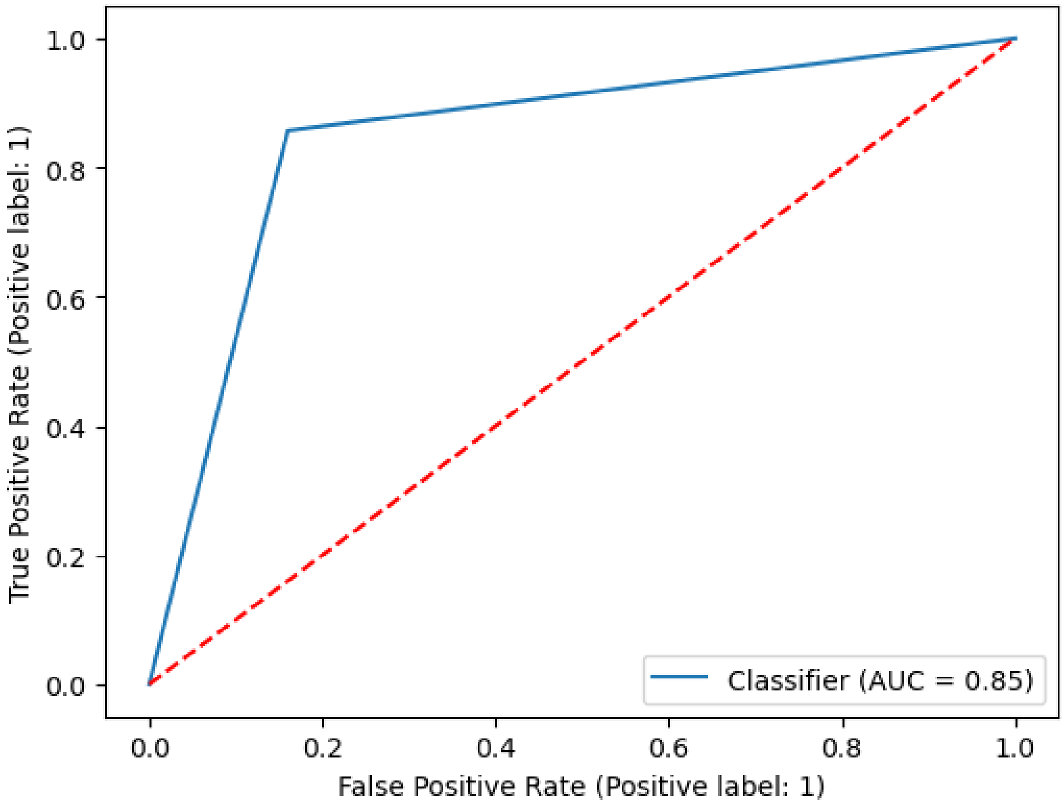
[332]:
```
(array([1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1,
        0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1,
        0, 1, 1, 1, 0, 1, 0, 0, 1]),
 array([1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1,
        0, 1, 1, 1, 0, 1, 0, 1, 1]))
```

We can see above the actual labels and the outputted prediction labels from the logistic model. Next let's use sklearn's accuracy scoring method by passing those two arrays in. As we see below we get a pretty good accuracy score of ~88.6%. Not too shabby for some handwriting data. Seems like there is definitely some indication that these handwriting tasks might be able to give a solid prediction on disease. Below that we plot the roc curve, as this is a classification problem we can use ROC to see the True positive rate(model predicted Alzheimer's and the subject did indeed have Alzheimer's) vs the false positive rate(Model Predicted Alzheimer's but subject did not have Alzheimer's).

[333]:
```
from sklearn.metrics import accuracy_score

acc = accuracy_score(Y_test, ypred)
acc
```

[333]:
```
0.8490566037735849
```

[334]:
```
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.metrics import RocCurveDisplay

fpr, tpr, thresh = roc_curve(Y_test, ypred)
rocAucScore = roc_auc_score(Y_test, ypred)

RocCurveDisplay.from_predictions(Y_test, ypred)
plt.plot(np.arange(0, 1.1, 0.1), np.arange(0, 1.1, 0.1), 'r--')
plt.show()
```

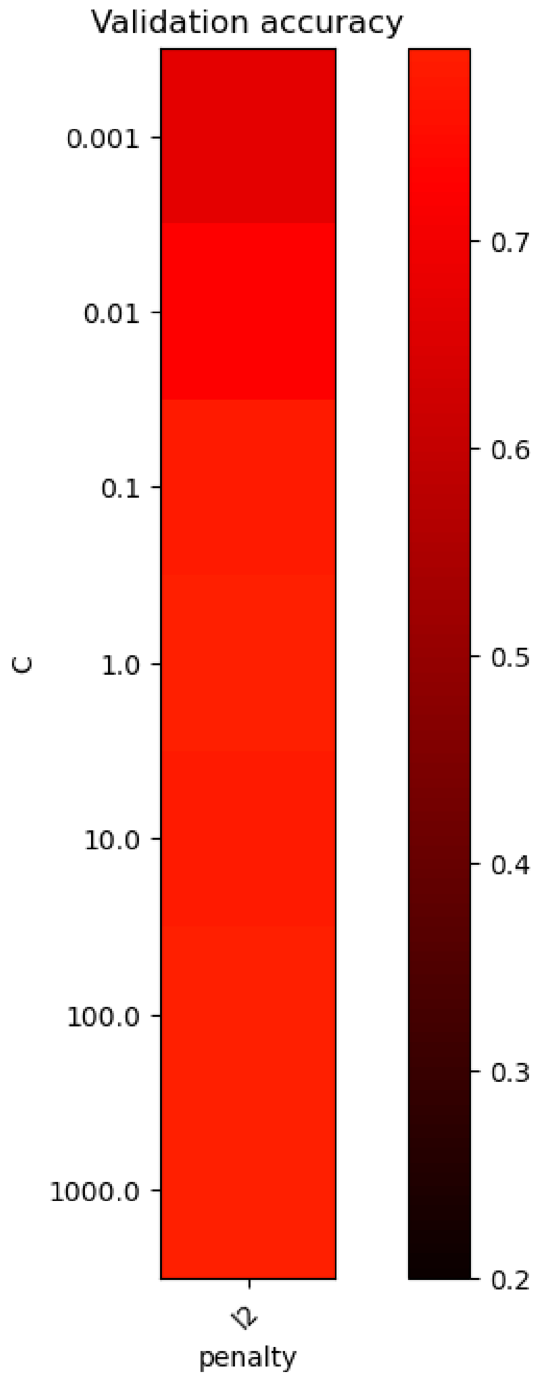


Let's do a quick GridSearch for Hyper parameter tuning so we can get our best possible model. First we will bring in the plotSearchGrid function from our Week 5 homework. Then we can display the output of our param trials on a grid.

[335]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
import seaborn as sns
import copy

%matplotlib inline

class MidpointNormalize(Normalize):

    def __init__(self, vmin=None, vmax=None, midpoint=None, clip=False):
        self.midpoint = midpoint
        Normalize.__init__(self, vmin, vmax, clip)

    def __call__(self, value, clip=None):
        x, y = [self.vmin, self.midpoint, self.vmax], [0, 0.5, 1]
```

```
        return np.ma.masked_array(np.interp(value, x, y))

def plotSearchGrid(grid):

    scores = [x for x in grid.cv_results_["mean_test_score"]]
    scores = np.array(scores).reshape(len(grid.param_grid['C']), len(grid.
 ↪param_grid["penalty"]))

    plt.figure(figsize=(10, 8))
    plt.subplots_adjust(left=.2, right=0.95, bottom=0.15, top=0.95)
    plt.imshow(scores, interpolation='nearest', cmap=plt.cm.hot,
               norm=MidpointNormalize(vmin=0.2, midpoint=0.92))
    plt.xlabel('penalty')
    plt.ylabel('C')
    plt.colorbar()
    plt.xticks(np.arange(len(grid.param_grid["penalty"])), grid.
 ↪param_grid["penalty"], rotation=45)
    plt.yticks(np.arange(len(grid.param_grid["C"])), grid.param_grid["C"])
    plt.title('Validation accuracy')
    plt.show()
```

[336]:
```
from sklearn.model_selection import GridSearchCV

params={'C':np.logspace(-3,3,7), 'penalty':['l2']} #l2 is the only penalty␣
 ↪hyper parameter that works


grid = GridSearchCV(LogisticRegression(max_iter=10000),params, cv=3)
grid.fit(X_train, Y_train)

plotSearchGrid(grid)

display(grid.best_params_, grid.best_score_)
```




```
{'C': 1.0, 'penalty': 'l2'}
```

```
0.7928861788617887
```

From the grid search it looks like the default hyperparameter settings are the best for our use. Only the l2 penalty would work for me so that's all I could test for that hyperparameter.

Now let's see what K means can tell us. We should fit a K means model with 2 clusters to see if it can possibly group the samples in either 0(Healthy) or 1(Patient)

[174]:
```python
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix

kmeans = KMeans(n_clusters=2, n_init="auto").fit(X)

print(kmeans.labels_)
print(y)
```

```
[1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1
 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 0 1 0 0 1 1 0 0 1
 1 1 1 1 0 1 0 1 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1
 1 1 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 0 0 1 1 0 1 1 1 1
 1 1 1 0 0 1 0 1 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 1 1 0]
[0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1 0 0
 1 0 0 0 0 1 1 1 1 1 0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 0 0 1 1
 0 0 1 0 1 0 1 1 0 1 1 1 0 1 0 1 0 0 1 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 1 1 0
 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 1 1 1 1 1 0 0 0 1 0 1 0 0 1 1 1 0 1 0 1 1 1
 0 0 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 1]
```

At this point K Means knows nothing about the labels. The nice thing about only having two clusters is that there are only two possible permutation interpretations of the labels, i.e. ((0,1) means (Patient, Healthy) or (0,1) means (Healthy, Patient)). We could just try either permutation and pick the best accuracy, but we built up a nice label permutor in week 7 so let's modify that to run it, sure it's overkill for only two permutations, but you could run it for more.

[197]:
```python
import itertools
import functools
import operator

def label_permute_compare(ytdf,yp,n=2):
    """
    ytdf: labels dataframe object
    yp: clustering label prediction output
    Returns permuted label order and accuracy.
    Example output: (3, 4, 1, 2, 0), 0.74
    """
# YOUR CODE HERE
    curBestAcc = 0.0
    curBestPerm = (0,0)
    #ytdf = ytdf.to_numpy()
    #actLabel = dict(map(lambda i,j : (i,j) , ['P', 'H'],[0,1]))
    #ytdf2 = list(map(lambda x: actLabel[x[0]],ytdf))
    for itp in itertools.permutations([0,1]):
        #Need to update the prediction output with different permutations
        yp2 = list(map(lambda x,y: 1 if itp[x]==y else 0, yp, ytdf))
```

```
        yp2Numerator = functools.reduce(operator.add, yp2)
        yp2Acc = yp2Numerator/174 #Updated denominator here to be number of␣
↪samples

        if(yp2Acc > curBestAcc):
            curBestPerm = itp
            curBestAcc = yp2Acc
    return  curBestPerm, curBestAcc
```

[204]:
```
bestperm,bestacc = label_permute_compare(y,kmeans.labels_)

bestperm,bestacc
```

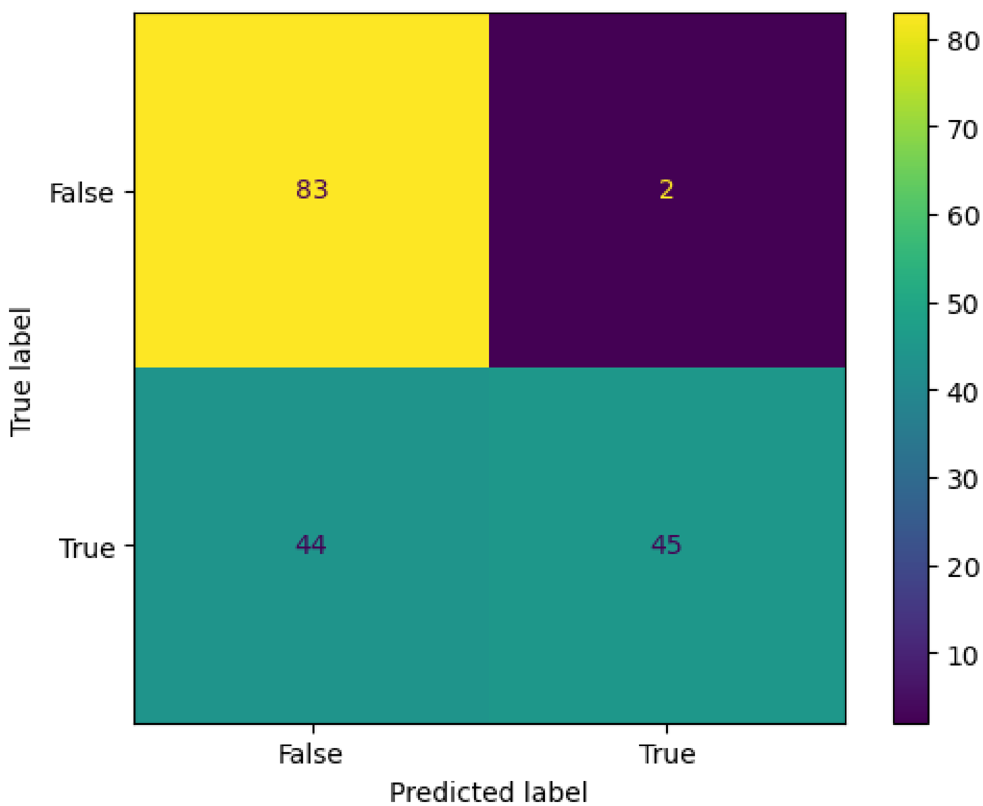
[204]: ((1, 0), 0.735632183908046)

6 Results Analysis

So we can see above that K-Means gives us accuracy around 73.5%. Let's look at a confusion matrix to see what kind of predictions the model may be good at (i.e. maybe it's better at predicting Healthy vs predicting Alzheimer's)

[275]:
```
from sklearn.metrics import confusion_matrix, f1_score
from sklearn import metrics

cmkm = confusion_matrix(y,list(map(lambda x: bestperm[x], kmeans.labels_)))

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = cmkm,␣
↪display_labels = [False, True])
plt.figure(figsize = (8,5))
cm_display.plot()
plt.show()

display(f1_score(y, list(map(lambda x: bestperm[x], kmeans.labels_))))
```

<Figure size 800x500 with 0 Axes>




```
0.6617647058823529
```

Let's analyze this confusion matrix a little bit to see how this KMeans clustering does as a predictor. The first thing I notice is that we obtained very few false positives, there were only 2, this gives us a false positive rate of 2/85 or .0235 or 2.35%. This means when the model predicts true it is very likely that the subject has Alzheimer's.

The next thing of note is there seems to be a lot of false negatives, there were 44 false negatives making the false negative rate 44/89 or .494 or 49.4%. This means if the model predicts a negative(healthy) then it's only right about half the time.

The specificity of this model is 83/85 or .9765 or 97.65%. This means that when the actual diagnosis is negative the model predicts it with 97.65% probability.

The recall of this model is 45/89 or .5056 or 50.56%. This means that if the actual diagnosis is positive the model only predicts correctly about half of the time.

Our F1 score of .661 isn't great, but it does show that the model does have some validity.

If we compare the accuracy of our two models, the first model with PCA and Logistic Classifier is clearly better than the KMeans. There is a chance that K Means has a False-Positive rate, but it's extremely unlikely that you would want to use this model over the PCA and Logistic model.

7 Conclusion

This has been a very interesting project to see if there are any unsupervised methods that can lead to insights into potential Alzheimer's diagnoses through handwriting. It seems that PCA with a simple logistic regression can provide a decently accurate prediction on whether or not a patient has Alzheimer's through their handwriting. It should be noted that it is decently accurate on THIS dataset. It has some problems, the main issue is it's a relatively small dataset, so the sample size might not be sufficient to offer truly accurate predictions. Another shortcoming is we don't actually know how we could differentiate Alzheimer's from some other disease like Parkinson's that might cause differences in handwriting, it is within the realm of possibility that this model prediction could lead to a misdiagnosis of Alzheimer's when maybe someone just had a stroke that affected their motor skills. An improved dataset could potentially have more classification classes for different diseases, that would be very interesting to see if you could differentiate different diseases based soley on handwriting. It should also be noted that there are likely better models than logistic regression that could be used; however, our focus in this project was on the unsupervised learning models and that did help us in regards to dimensionality reduction. In addition clustering was at least able to give us a better than guessing prediction knowing nothing about the labels, which is pretty neat, clearly there is likely some correlation in motor skill dysfunction with these neurological diseases and these unsupervised methods seem to bear that out. Further improvements could be made with a better model like Gradient Boosting or maybe using SVM or Random Forest. I'd also be interested in seeing how this could do in a deep learning model and may explore that for the next project. Thanks for taking the time to read my project!

Dataset citation:

`Fontanella,Francesco. (2022). DARWIN. UCI Machine Learning Repository. https://doi.org/10.2443`

[ ]:

```

```