CA4015 – Assignment 2

- Kian Sweeney / 18306226 / kian.sweeney27@mail.dcu.ie

- Code can be viewed here: https://github.com/kiansweeney11/ca4015-second-assignment

Introduction

For our second assignment of CA4015: Advanced Machine Learning we were tasked with the analysing of data related to the analysis of volatile organic compounds (VOC). The data set provided contained four sheets, 2 (Long and Wide) focused on the VOCs that were successfully identified across all the bacteria and across differing nutritional environments. The other two sheets contain OD600 measurements for a specific time point. We will only focus on the first two mentioned sheets most specifically the 'wide' sheet. We were tasked with using an Automated Machine Learning (AutoML) approach for the classification of our data using an appropriate platform and then comparing our results to a random forest approach. We also needed to combine this with use of cross validation in our random forest and AutoML approaches to validate the success of our respective results. As per Wikipedia, AutoML is defined as "the process of automating the tasks of applying machine learning to real-world problems. AutoML covers the complete pipeline from the raw dataset to the deployable machine learning model". In my experiments I tried predicting the strain of bacteria using the compounds columns as features. I tried varying levels of test to training split on both my AutoML and random forest approaches. Firstly, I would have to take the dataset and clean it as required.

Data Cleaning

Initially, I loaded my dataset in on excel to have a quick look at it before processing it on Jupyter Notebook. Here is what it looked like:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2	The data in this sheet was used for the formulation of the PCA scores plots shown in Figure 1													
3														
4	Species	Strain	Samples	Ethyl Acetat	Ethanol	Propanoic ac	2-Pentanone	Decane	Methyl Isobu	Amylene hyc	Butanoic aci	Isobutyl acet	Methyl isova	1-Propanol
5	SA	SA_A	SA.A_TSB_A	465374	1027715		1289650	800581	324424	73015			0	
6	SA	SA_A	SA.A_TSB_B	193151	1050974		504113	294680	189630	0			0	
7	SA	SA_A	SA.A_TSB_C	403286	1850391		1169501	15	228163	73558			0	
8	SA	SA_A	SA.A_TSB_D	129833	5140770		1926072	124282	0	188367			0	
9	SA	SA_A	SA.A_TSB_E	117105	3422557		246751	0	0	0			0	
10	SA	SA_B	SA.B_TSB_A	316764	914667		560337	456376	167165	90157			0	
11	SA	SA_B	SA.B_TSB_B	141636	801684		497068	408709	152829	139948			0	
12	SA	SA_B	SA.B_TSB_C	292548	1037912		1102211	710674	228457	53152			0	
13	SA	SA_B	SA.B_TSB_D	0	7091491		1336035	470055	0	210357			0	
14	SA	SA_B	SA.B_TSB_E	588447	8155560		1737916	201138	0	295857			0	
15	SA	SA_A	SA.A_BHI_A	0	3054748		832334	0	0	0			0	
16	SA	SA_A	SA.A_BHI_B	0	909704		444238	0	0	0			0	
17	SA	SA_A	SA.A_BHI_C	0	2419195		540199	0	0	0			0	

As we can see this data is very messy. The headings of the columns appear to be in the fourth row and the text in the second row explaining the contents of the sheet is unnecessary as is typical of unclean dataset. We can also see in some of our columns null values (no values present not even 0). Initially, I interpreted this as no readings of these compounds present so they could be denoted as a 0 value. However, I changed my

approach and “imputed” the data instead. I used the sklearn library’s “SimpleImputer” implementation to replace all missing values in the data with the mean in each column. This is regarded as a better approach when the dataset is small (we have 84 rows only albeit with high dimensionality). It can add bias and variance to the data but most other approaches appear liable to this also. I looked at applying linear regression to predict values but felt this approach may be complicated to implement on so many columns due to high dimensionality. Our code and results are as follows:

```
In [20]: from sklearn.impute import SimpleImputer
         imp = SimpleImputer(missing_values=np.nan, strategy='mean')
         num = data1.iloc[:, 3:]
         imp.fit(num)
         imp.transform(num)
```

	Ethyl Acetate	Ethanol	Propanoic acid, ethyl ester	2-Pentanone	Decane	Methyl Isobutyl Ketone	Amylene hydrate	Butanoic acid, 2-methyl-, methyl ester	Isobutyl acetate	Methyl isovalerate	...	1-Dodecanol
0	465374.0	1027715.0	17989.407407	1289650.0	800581.0	324424.000000	73015.0	287247.703704	46016.203704	0.000000	...	2.098782e+06
1	193151.0	1050974.0	17989.407407	504113.0	294680.0	189630.000000	0.0	287247.703704	46016.203704	0.000000	...	2.098782e+06
2	403286.0	1850391.0	17989.407407	1169501.0	15.0	228163.000000	73558.0	287247.703704	46016.203704	0.000000	...	2.098782e+06
3	129833.0	5140770.0	17989.407407	1926072.0	124282.0	0.000000	188367.0	287247.703704	46016.203704	0.000000	...	2.098782e+06
4	117105.0	3422557.0	17989.407407	246751.0	0.0	0.000000	0.0	287247.703704	46016.203704	0.000000	...	2.098782e+06

We also tidy up our headings and remove the first four null rows from the dataset on the ‘wide’ sheet. This leaves us with our cleaned dataset and we can get ready to classify it using AutoML and RF. I did actually implement AutoML using the original dataset with the missing values. However, I couldn’t implement cross validation methods as the sklearn module I was using for cross-validation didn’t support the use of null values. I also adapted my approach with regards to categorical values too. I originally used the string values connected to each strain but changed this to number each value then using the following code:

```
In [29]: x = features['Strain']
         exit_status_map = {'SA_A': 0, 'SA_B': 1 ,'EC_A': 2, 'PA_A': 3, 'EC_B': 4, 'PA_B': 5}
         xyz = x.map(exit_status_map)

In [30]: xyz

Out[30]: 3     0
         4     0
         5     0
         6     0
         7     0
              ..
         82    4
         83    4
         84    4
         85    4
         86    4
         Name: Strain, Length: 84, dtype: int64
```

Once this was done, I set about implementing my AutoML classification task. We would be trying to predict the strain given the values of compounds contained in the bacteria. We would try this at 10:90, 20:80, 25:75, 30:70, 40:60 and 50:50 training to test split.

AutoML Classification

There are a lot of different AutoML platforms available today such as: Google colab, teapot amongst many others. I decided to use the supervised AutoML package imported on Jupyter Notebook as so:

```
from supervised.automl import AutoML
```

This package is installed on the command line using the following command:

```
pip install mljar-supervised
```

This package creates directory with reports on the AutoML command ran and also computes a baseline for our data assessing the need for machine learning at all. It also imputes missing values which would be useful in the data provided. However, this imputes the data while on the AutoML run so our normal dataset doesn't actually get the imputed data values. This is why initially I used my unimputed dataset and ran AutoML and then re-ran it on a separate file with the imputed data already present. This is to allow us to compare our results. Initially, we look at how AutoML preformed on the unimputed dataset. We split our data up accordingly:

```
In [187]: y1
Out[187]: 3      SA_A
          4      SA_A
          5      SA_A
          6      SA_A
          7      SA_A
                 ...
          82     EC_B
          83     EC_B
          84     EC_B
          85     EC_B
          86     EC_B
          Name: Strain, Length: 84, dtype: object
```

This will be the column we are trying to predict, the strain column. We then take all the numeric data columns:

```
In [188]: X1.head()
```

Out[188]:

	Ethyl Acetate	Ethanol	Propanoic acid, ethyl ester	2-Pentanone	Decane	Methyl Isobutyl Ketone	Amylene hydrate	Butanoic acid, 2-methyl-, methyl ester	Isobutyl acetate	Methyl isovalerate	...	1-Dodecanol	Methyl tetradecanoate	2-Pentadecanone
3	465374	1027715	NaN	1289650	800581	324424	73015	NaN	NaN	0	...	NaN	NaN	NaN
4	193151	1050974	NaN	504113	294680	189630	0	NaN	NaN	0	...	NaN	NaN	NaN
5	403286	1850391	NaN	1169501	15	228163	73558	NaN	NaN	0	...	NaN	NaN	NaN
6	129833	5140770	NaN	1926072	124282	0	188367	NaN	NaN	0	...	NaN	NaN	NaN
7	117105	3422557	NaN	246751	0	0	0	NaN	NaN	0	...	NaN	NaN	NaN

5 rows × 67 columns

Firstly, I'll look at how the AutoML classification performed imputing the NaN values by itself. The following table summarises the results on each training to test split:

Training to Test Split	Accuracy
0.1	0.7777777777777778
0.2	0.6470588235294118

0.25	0.6190476190476191
0.3	0.7307692307692307
0.4	0.6470588235294118
0.5	0.6190476190476191

We couldn't cross validate our results due to the presence of NaNs in the data. So now I look at AutoML classification of bacteria strain using our imputed data from earlier. These are our results:

Training Test Split	Accuracy
0.15	0.9230769230769231
0.2	0.8235294117647058
0.25	0.8095238095238095
0.3	0.8461538461538461
0.4	0.6470588235294118
0.5	0.6904761904761905

As we can see our data with imputed values based on the mean of values already in a column is significantly higher than before. However, this time we can cross validate our accuracy values. I implemented a couple of different metrics including cross validation to test the accuracy of our results. I tried C-support vector classification and using the built-in cross validation score library from sklearn like so:

```
In [89]: from sklearn.model_selection import cross_val_score
         clf = svm.SVC(kernel='linear', C=1, random_state=42)
         scores = cross_val_score(clf, X_test20, y_test20, cv=4)
         print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))

         0.24 accuracy with a standard deviation of 0.18
```

This splits the data, fits a model and computes a cross validation score four times, we take the mean of these values as our cross-validation score for each x and y test splits. This is the most simplistic way of implementing cross-validation and the one we use for all our data. I also used the aforementioned C-support vector classification. It works well with smaller datasets which is of benefit to us with the dataset we are using. Using support vector machines like this is highly beneficial in high dimensional datasets like ours and is also memory efficient as it uses a subset of training points in the decision function. The SVC implementation fit time scales at minimum quadratically with the number of samples in the data to predict new values. I felt by combining these two we would get a very clear picture of how accurate our predictions were. Here are our scores for both of these:

Training to Test Split	SVC	Cross-Validation Mean Score
0.15	0.46153846153846156	0.33
0.2	0.7058823529411765	0.24
0.25	0.7142857142857143	0.45
0.3	0.7692307692307693	0.54

0.4	0.6470588235294118	0.59
0.5	0.5952380952380952	0.57

Looking at our scores here combined with our accuracy scores from earlier there appears to be a sweet spot between 0.25 and 0.3 test to training data split. The cross-validation scores increase with the larger splits but the SVC scores suffer directly as a result. There certainly appears to be the risk of over fitting the data with test to training splits in excess of 40:60 here. This leads us on to our implementation of random forests and see how our results contrast with an AutoML approach.

Random Forest Comparison

Next, we implement a random forest approach to compare with our results from earlier. Again, we couldn't use our original cleaned data with null values as the sklearn library we were using didn't allow for null values. This meant we had to use our imputed data based on the mean values of columns. We use the sklearn ensemble's RandomForestRegressor package to predict our values for the strain given the numeric compounds present. This is an example of the code we ran:

```
In [195]: X_trainrf30, X_testrf30, y_trainrf30, y_testrf30 = train_test_split(imputed_data1, xyz, test_size=0.3, random_state=0)

In [197]: from sklearn.ensemble import RandomForestRegressor

          regressor = RandomForestRegressor(n_estimators=20, random_state=0)
          regressor.fit(X_trainrf30, y_trainrf30)
          y_predrf30 = regressor.predict(X_testrf30)
```

We ran the tests at 10:90, 20:80, 25:75, 30:70, 40:60 and 50:50 test to training splits. Here were our results for accuracy, SVC and cross-validation mean scores:

Training to Test Split	Accuracy	SVC	Cross-Val Mean Score
0.1	0.6667	0.6667	0.54
0.2	0.5294117647058824	0.7058823529411765	0.64
0.25	0.5238095238095238	0.6667	0.62
0.3	0.34615384615384615	0.6923076923076923	0.65
0.4	0.47058823529411764	0.7647058823529411	0.61
0.5	0.42857142857142855	0.6190476190476191	0.74

As we can see our accuracy is significantly lower than our AutoML approach. This is to be expected as AutoML trains and tests the data on a selected collection of algorithms finding the best fitting algorithm. When I ran AutoML on the imputed mean data it predominately returned Xg Boost as the best performing algorithm on the data despite random forest being tested each time. One similarity appears on the implementation of our random forest algorithm in that the best split across our three metrics here appears to be around 25:75 training to test data split (here the threshold seems to be between 0.20 and 0.25 not 0.25 and 0.3 with AutoML). However, one thing that is interesting to note is the much higher cross-validation scores on our different data splits. The lowest cross validation score with our RF

approach would be the third highest score on our AutoML cross-validation scores. All our other cross-validation scores for RF are higher than the highest AutoML score also. While the SVC scores are at a similar level the different in cross-validation scores using the cross-evaluation score function is stark. This suggests that maybe our RF approach would adapt better at unseen values than our AutoML approach and our accuracy scores for the AutoML approach are potentially inflated, with our data suffering from overfitting. It would be interesting to implement our own version of XgBoost and see does the cross validation scores align with the poor AutoML cross-validation scores as this was regularly denoted as the best performing algorithm.