CITS5508 Lab Sheet 2: Classifying Forests

Name: James Mok
Student number: 21120365
Date created: March 10th 2019
Last modified: March 22th 2019

The objective of this Lab is to investigate the process of classifying forest types from areas in Japan. This will go through the following areas such as data visualisation, normalisation, classifiers and more.

1. Data Visualisation

First step is tmporting and visualising the data set to help provide insight into the next steps.

In [1]:

```
# Importing common libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Reading in the data

train =  pd.read_csv('training.csv')
test =  pd.read_csv('testing.csv')
train.shape

#Display first 5 rows
train.head(5)
```

Out[1]:

	class	b1	b2	b3	b4	b5	b6	b7	b8	b9	...	pred_minus_obs_H_b9	pred_minu
0	d	39	36	57	91	59	101	93	27	60	...	-2.36	
1	h	84	30	57	112	51	98	92	26	62	...	-2.26	
2	s	53	25	49	99	51	93	84	26	58	...	-1.46	
3	s	59	26	49	103	47	92	82	25	56	...	2.68	
4	d	57	49	66	103	64	106	114	28	59	...	-2.94	

5 rows × 28 columns

In [2]:

```
#Understanding data size

print('Below are the dimensions of data (rows,columns)')
print('Training set: '+ str(train.shape))
print('Test set: '+ str(test.shape))
```

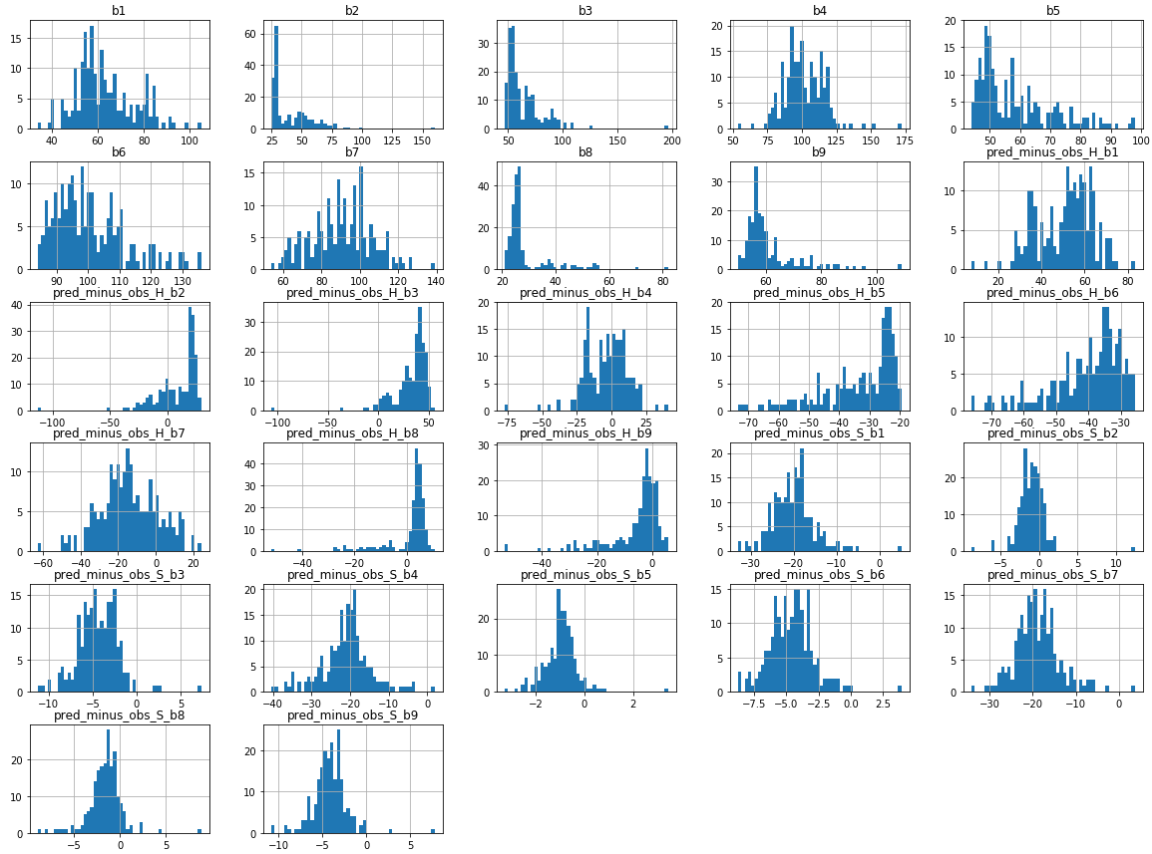
```
Below are the dimensions of data (rows,columns)
Training set: (198, 28)
Test set: (325, 28)
```

1.1 Data Observations

Using this information we can consider plotting this, where 50 bins should provide a clear enough representation of the dataset.

In [3]:

```
#Visualising data by creating histograms

train.hist(bins=50, figsize=(20,15))
plt.show()
```



1.2 Observations from Plots

From the plots, some observations can be made:

- Most plots have the shape that looks like it is normally distributed
- Plots skewed to the left or right
- Fairly consistent with few outliers

2. Simplifying and Manipulating the Data

Here we will simplify the data to only include the 9 features (b1, b2 ... b9).
Also we will extract the class labels column.

In [4]:

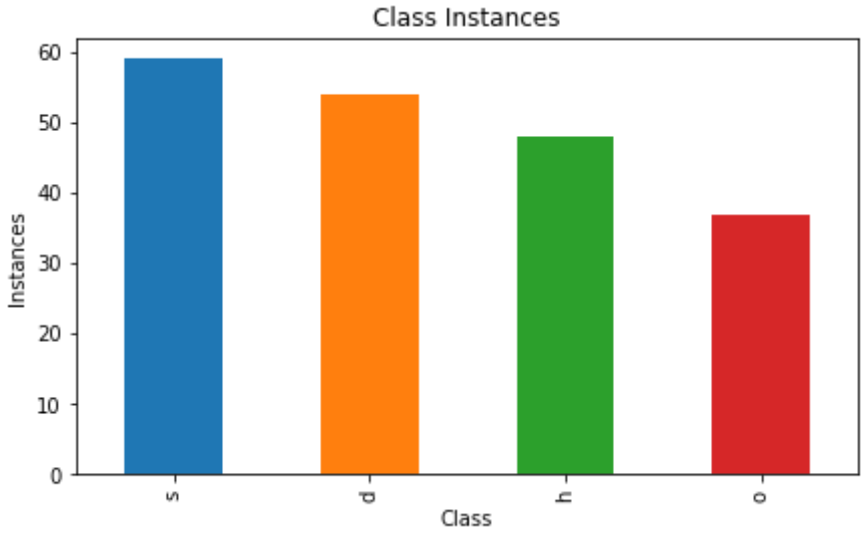
```
#Taking only columns 1(Class) and 2 to 10 (b1,b2...b9)

#Therefore removing columns to exclude pred_minus_obs

train_split = np.split(train, [1,10], axis=1)
test_split = np.split(test, [1,10], axis=1)

#Uncomment to display training set classes column (=)
#train_split[0]

#Uncomment to display training set features column
#train_split[1]
```

3. Imbalanced training set?

Now we will plot the instances for classes to verify if training set is imbalanced.

In [5]:

```
#Index [0] takes the first split which is the classes column
train_class = train_split[0]
test_class = train_split[0]

#Plotting class instances in a bar plot
plt.figure(figsize=(7, 4))

class_instances = train_class['class'].value_counts().plot.bar()

class_instances.set_title('Class Instances');
class_instances.set_xlabel('Class');
class_instances.set_ylabel('Instances');
```



3.1 Is this Imbalanced?

From the plot it can be seen that this training set is **not** imbalanced.

The instances are within the same order of magnitude and therefore not imbalanced. This allows us to move on to soon attempt classifcation

4. Normalising Data

Before we use the classification algorithms, here we will compare two normalisation methods - `MinMaxScaler` , and `StandardScaler`

4.1 Which normalisation method?

First, to understand which method is more suitable, details of each function which are taken from the documentation are as follows:

> `MinMaxScaler` - This method uses the minimum and divides the data set which scales the data from 0 to 1.
>
> - More suitable for data that has a smaller standard deviation and also this normalisation method is adversely affected by outliers.
>
> `StandardScaler` - This method assumes the data set is normally distributed, which scales your distribution of data to a mean of 0 and a standard deviation of 1.
>
> - Suitable for normally distributed data.
>
> `RobustScaler` - Similar to MinMaxScaler however utilies less of the data for scaling.
>
> - Resembles `MinMaxScaler` but is less prone to the adverse affects of outliers.

From the above mentioned points, `StandardScaler` seems more suitable based on the fact that the data is mostly normally distributed.

In [6]:

```
#Normalising data

#Utilising StandardScaler
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings("ignore")

scaler = StandardScaler()

train_array = scaler.fit_transform(train_split[1])
test_array = scaler.transform(test_split[1])
```

4.2 Normalisation using `StandardScaler`

It is important to note that the training set uses `fit_transform` , fits the data and then standardises this by scaling and centering.
The test set only uses `transform` which scales and centers the data but does not fit the data which is important as we want to leave the test set unaltered.

5. One-Versus-All Binary Classification with Stochastic Gradient Descent

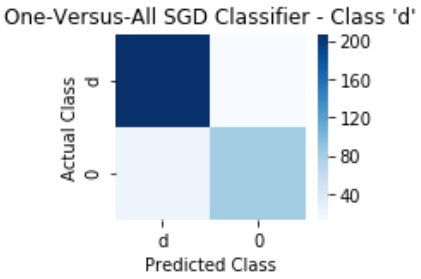
In this section, a One-Versus-All Binary SGD Classifier is used. However, firstly our data needs to be in binary form which has been done using `LabelBinarizer` .

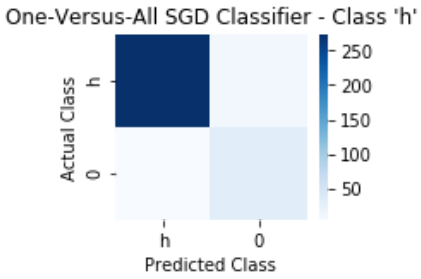
After the data has been split into each class which we define as the 'one' and then is classified against the other classes ('all') through using a `for` loop.
The results are shown below through a confusion matrix and a metric score.

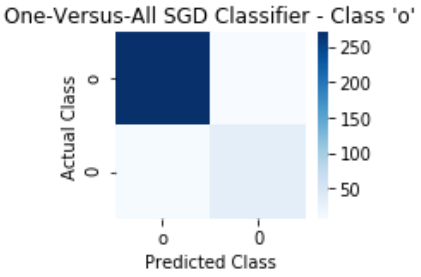
In [7]:

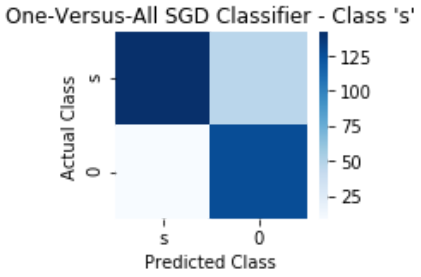
```
#One-versus-all binary classification with Stocastic Gradient Descent
from sklearn.preprocessing import LabelBinarizer
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score
import seaborn as sns

#Using LabelBinarizer to provide binary for to One-Versus-All classification
lb = LabelBinarizer()

#Training set binarised
y_train = train_split[0]
X_train = train_array
y_train_binary = lb.fit_transform(y_train)
print('Binarized classes are ordered:' + str(lb.classes_) + '\n')

#Test set binarised
y_test = test_split[0]
X_test = test_array
y_test_binary = lb.fit_transform(y_test)

#Creating plot figure and labels
labels=['d','h','o','s']
fig = plt.figure(figsize=(10,6))
fig.subplots_adjust(hspace = 1, wspace=2)

#for loop to cycle through each class and classify using SGD Classifier
for i in range(4):

    #Selecting class to train
    y_train_class = y_train_binary[:,i]
    y_test_class = y_test_binary[:,i]

    #Training in SGDClassifier
    sgd_clf = SGDClassifier(random_state=1)
    sgd_clf.fit(X = X_train, y = y_train_class)
    prediction = sgd_clf.predict(X_test)

    #Creating confusion matrix
    ax = fig.add_subplot(2, 2, i+1)
    conf_mx = confusion_matrix(y_test_class, prediction)
    sns.heatmap(conf_mx, xticklabels= [labels[i], '0'], yticklabels= [labels[i], '0'],
cmap='Blues')

    #Plot labels
    plt.title('One-Versus-All SGD Classifier - Class ' +'\''+ str(labels[i]) +'\'')
    plt.xlabel('Predicted Class');
    plt.ylabel('Actual Class');

    #Accuracy Score
    print('For class: \''+ str(labels[i])+ '\'')
    print('Accuracy Score: ' + str("%.2f" % (accuracy_score(y_test_class, prediction)*1
00)) + '%')
    print('F1 Score: '+ str("%.2f" % (f1_score(y_test_class, prediction)*100)) + '%' +'
\n')
```

```
Binarized classes are ordered:['d' 'h' 'o' 's']

For class: 'd'
Accuracy Score: 88.92%
F1 Score: 82.00%

For class: 'h'
Accuracy Score: 94.77%
F1 Score: 80.00%

For class: 'o'
Accuracy Score: 94.15%
F1 Score: 78.65%

For class: 's'
Accuracy Score: 82.77%
F1 Score: 81.94%
```









5.1 Observations of One-Versus-All SGD Classifier

To explain F1 Score and Accuracy score, first we will define each and then determine which metric is the most suitable

`accuracy_score` - is the correctly predicted observations divded by the total observations
 Gives the percantage of total observations that were predicted correctly. However, this is not so useful for imbalanced data sets, as it does not account for false positives and false negatives

`f1_score` - Is the weighted average of Precision and Recall

> 'Precision' - Is the ratio of correctly predicted positive observations to the total predicted positive observations.
> For example, for all that were classified in class 's', how were actually class 's'.
>
> 'Recall' - is the ratio of correctly predicted positive observations to the all observations in the actual class.
> For example, for all forest the were actually in class 's', how many were identified as class 's'.

In this case the classes have been binarised, which creates more of an imbalanced dataset. This renders `accuracy_score` less useful.
F1 score will be much better as it takes into account false positives and false negatives which is a better metric for this situation.

5.2 Hyperparameter Tuning

Optimisation of parameters through `GridSearchCV'

The following hyperparameters have been chosen with reasoning shown below:

> `loss` - The `loss` parameter defines the loss function used, where `hinge` uses a linear SVM and `log` uses a logistic regression classifier.
>
> `alpha` - The learning rate, important to consider to converge to the minima in the loss function.
>
> `tol` - Works in conjunction with the learning rate where `tol` is the stopping criterion. This attempts to identify where the minima point is, where the loss > previous_loss - tol.
>
> `max_iter` - Maximum passes over the training data.

Some of the key parameters to consider is how the `alpha` , `tol` and `max_iter` work together to converge towards the optimal result in the loss function.

In [8]:

```
#Hyperparamter tuning
from sklearn.metrics import make_scorer
from sklearn.model_selection import GridSearchCV

#List of parameters used in tuning
parameters = {
    'loss':('hinge', 'log'),
    'alpha':[0.0001, 0.001,0.01],
    'l1_ratio': [0.01, 0.05, 0.1, 0.2],
    'tol': [1,0.5,0.1,0.01],
    'max_iter': [5,10,25, 50, 100]
}


# Using F1 score as performance metric
f1_score_method = make_scorer(f1_score, average = 'macro')

#Cross-validation hyperparameter tuning
clf = GridSearchCV(sgd_clf, parameters, cv=5, scoring = f1_score_method)
clf.fit(X_train, y_train)

#Print F1 score and best parameters
print('Optimal score and parameters from tuning:')
print(clf.best_score_)
print(clf.best_params_)
```

```
Optimal score and parameters from tuning:
0.9593092024587759
{'alpha': 0.001, 'l1_ratio': 0.01, 'loss': 'hinge', 'max_iter': 10, 'tol':
1}
```

6. Logistic Regression with Softmax Function

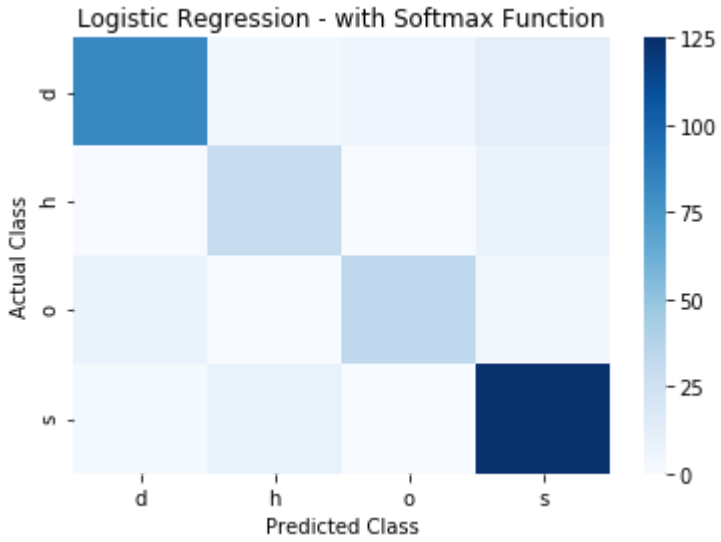
The steps for section 5. will now be repeated - Classifying Forests using Logistic Regression with Softmax function.

> Where `softmax` function calculates probabilities over all the possible target classes. This is utilised in the prediction process, taking the class with the highest probability as the prediction - great for multi-class classification.

In [9]:

```
#Logistic Regression with Softmax function
from sklearn.linear_model import LogisticRegression

#Training set
y_train = train_split[0]
X_train = train_array

#Test set
y_test = test_split[0]
X_test = test_array

#Logistic Regression with Softmax Function (multinomial)
#C = regularisation factor, prevents overfitting
#solver = lbfgs , an optimissation algorithm that supports in softmax and l2 penalty

softmax_reg = LogisticRegression(multi_class="multinomial",solver="lbfgs")
softmax_reg.fit(X = X_train, y = y_train)
prediction = softmax_reg.predict(X_test)

#Creating figure
fig = plt.figure()
labels=['d','h','o','s']

#Creating confusion matrix
conf_mx = confusion_matrix(y_test, prediction)
sns.heatmap(conf_mx, xticklabels= labels, yticklabels= labels,cmap = 'Blues')

#Plot labels
plt.title('Logistic Regression - with Softmax Function')
plt.xlabel('Predicted Class');
plt.ylabel('Actual Class');

#Score
#print('Accuracy Score:' + str("%.2f" % (accuracy_score(y_test, prediction)*100))+'%')
print('F1 Score: '+ str("%.2f" % (f1_score(y_test, prediction,average = 'macro')*100))
+'%' +'\n')
```

```
F1 Score: 81.33%
```



6.1 Observations

An F1 Score of 83.69% is not that great, however this needs to be taken into context once the hyperparameters are tuned

6.2 Hyperparameter Tuning

Some of these parameters have been explained in section 5.2, here it is important to consider `C` which is the regularisation factor.

> `C` - Helps to prevent overfitting of the training set through regularisation.

The tuned hyperparameters have been investigated below.

In [10]:

```
#Hyperparameter Tuning

parameters = {
    'tol':[1,0.5,0.1 , 0.05,0.01],
    'C': [2, 5, 10,15,20,30],
    'max_iter': [1,2,5,10,15,20,50]
}

#Cross Validation of 5 segments using F1 score
clf = GridSearchCV(softmax_reg, parameters, cv=5,scoring=f1_score_method)
clf.fit(X_train, y_train);

#Print F1 score and best parameters
print('Optimal score and parameters from tuning:')
print(clf.best_score_)
print(clf.best_params_)
```

```
Optimal score and parameters from tuning:
0.9440841325844355
{'C': 5, 'max_iter': 10, 'tol': 0.5}
```

7. Conclusion

Which classifier gave better performance?

Using the two classifiers with the F1 score metric, the results are shown below after hyperparameter tuning.

> One-Versus-All classifier with Stochastic Gradient Descent:
> F1 Score: 95.9%
>
> Multi-class Logistic Regression:
> F1 Score: 94.4%

Purely from this data set, we can conclude that the One-Versus-All classifier performed better. However, it will be interesting to see the comparison of the two classifiers on different data sets.

General conclusions

This lab was a fantastic introduction into machine learning. In summary of my learning:

1. Data manipulation and data visualisation to deal with pre-processing.
2. Details of the classifiers utilising metrics and visual diagrams to depict and understand the results.
3. Tuning the classifiers through adjusting hyperparameters.
4. Lastly, interpretation of data and the resuts.

Thanks for reading!

In [1]:

```
pip install pandoc
```

```
The following command must be run outside of the IPython shell:

    $ pip install pandoc

The Python package manager (pip) can only be used from outside of IPython.
Please reissue the `pip` command in a separate terminal or command prompt.

See the Python documentation for more information on how to install packag
es:

    https://docs.python.org/3/installing/
```

In [ ]:

```

```