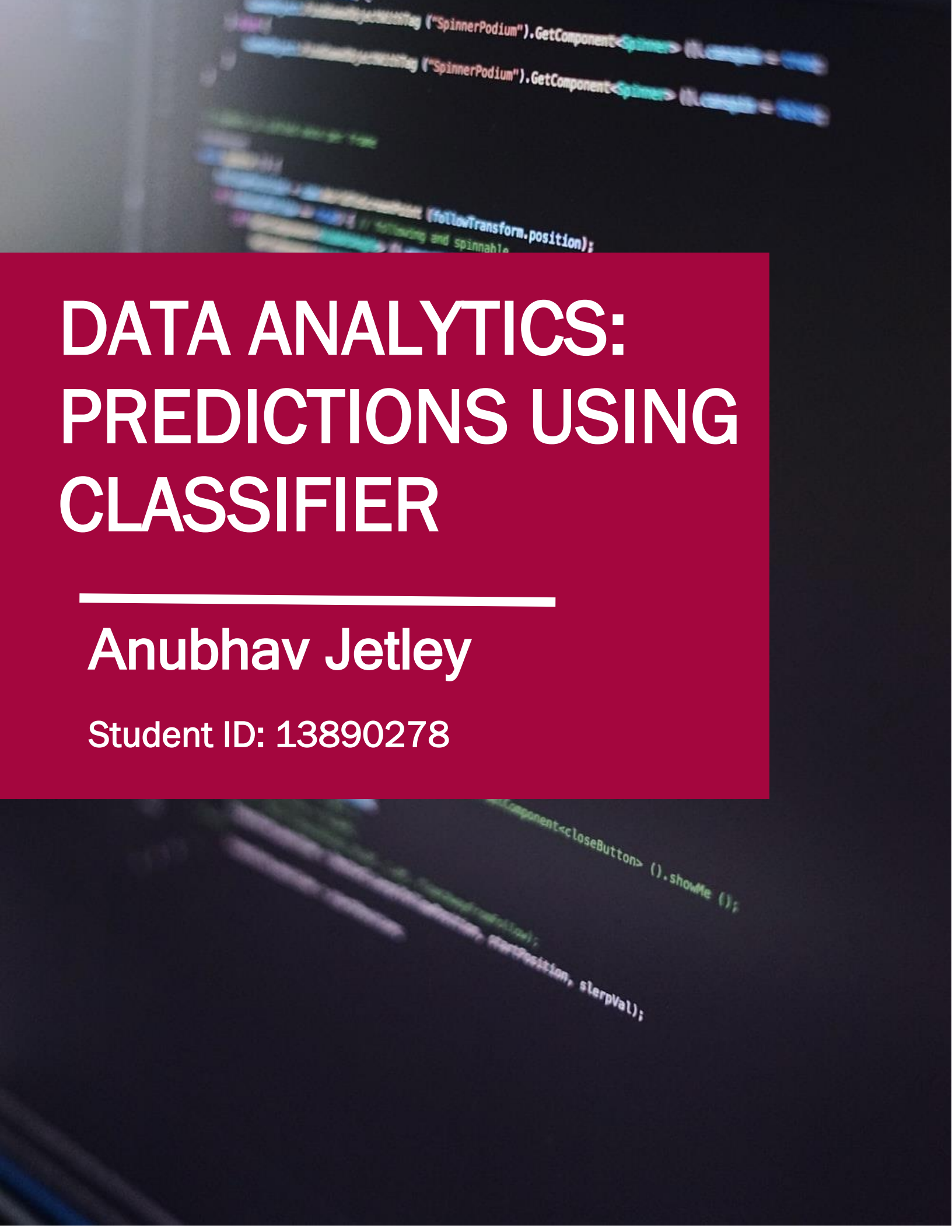


DATA ANALYTICS: PREDICTIONS USING CLASSIFIER

Anubhav Jetley

Student ID: 13890278

What is the data mining problem we are encountering?

The problem to be solved within the dataset, is a prediction of whether an individual has a salary option of either '>50K' or '<=50K'. Using the machine learning process of classification predicting the outcome for the attribute of salary can be determined using the binary representation of '0' for '<=50K' and '1' for '>50K'. And thus, within this problem lies the problem as to which machine learning classifier is the best choice for predicting salary, as well as what parameters must be set to these classifiers to provide the best results. Using **Python**, the training data can be used to make the classification predictions and the testing data can be used to implement the predictions. From which the accuracy of the classifier's predictions can be determined and analysed.

Data pre-processing steps and transformations used to make classifiers:

The following data pre-processing steps and transformations were made to allow for the data to be used to perform classifiers and get predictions for the salary attribute.

Data Pre-processing steps:

Step 1: Import Libraries

Pandas: Use for data manipulation and data analysis.
Numpy: a fundamental package for scientific computing with Python
Matplotlib and **Seaborn** for visualization
Scikit-learn used for pre-processing dataset for classifiers, as well as importing classifier algorithms and prediction calculations

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
```

```
from sklearn.preprocessing import LabelEncoder
# change categorical attribute values to numerical, with salary being in binary
# 1 and 0 form
```

```
import pandas as pd

import numpy as np

import seaborn as sns

from pandas.plotting import scatter_matrix

import matplotlib.pyplot as plt

from sklearn import model_selection

from sklearn.metrics import classification_report

from sklearn.metrics import confusion_matrix

from sklearn.metrics import accuracy_score

from sklearn.linear_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

from sklearn.naive_bayes import GaussianNB

from sklearn.svm import SVC

from sklearn.feature_selection import SelectKBest

from sklearn.model_selection import cross_val_score

from sklearn import preprocessing
```

Step 2: Read the data

Using the imported pandas library, we can read the csv file of the training and testing data

```
52
53  df_train = pd.read_csv(r'C:\Users\anubh\OneDrive\Documents\training dat.csv')
54  df_test = pd.read_csv(r"C:\Users\anubh\OneDrive\Documents\testing data.csv")
55
```

Step 3: Check for missing values

Using the .info () function, we can identify if there are any missing values within the dataset

```
#look for missing values
df_train.info()
df_test.info()
```

```
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 15 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   ID                    10000 non-null  int64
 1   Age                   10000 non-null  int64
 2   Employment class      10000 non-null  object
 3   Fnlwgt                10000 non-null  int64
 4   Education level       10000 non-null  object
 5   Education years       10000 non-null  int64
 6   Marital status        10000 non-null  object
 7   Occupation            10000 non-null  object
 8   Relationship status   10000 non-null  object
 9   Race                  10000 non-null  object
 10  Sex                   10000 non-null  object
 11  Capital gain          10000 non-null  int64
 12  Capital loss          10000 non-null  int64
 13  Work hours per week   10000 non-null  int64
 14  Native country        10000 non-null  object
dtypes: int64(7), object(8)
```

Shows the total number of entries including non-null and null entries

Shows the count of non-null entries

Hence, from the .info () table we can identify that there are no null values within this test dataset. The same was observed for the training dataset.

There is no need for any removal of NaN values.

Step 4: Convert categorical attributes into numerical values for classification

From the Scikit-learn library, we can use the preprocessing label encoder to convert categorical attribute values into numerical values.

```
lb_make = LabelEncoder()
df_train["Salary"] = lb_make.fit_transform(df_train["Salary"])
df_train["Sex"] = lb_make.fit_transform(df_train["Sex"])
df_train["Native country"] = lb_make.fit_transform(df_train["Native country"])
df_train["Marital status"] = lb_make.fit_transform(df_train["Marital status"])
df_train["Education level"] = lb_make.fit_transform(df_train["Education level"])
df_train["Occupation"] = lb_make.fit_transform(df_train["Occupation"])
df_train["Employment class"] = lb_make.fit_transform(df_train["Employment class"])
df_train["Relationship status"] = lb_make.fit_transform(df_train["Relationship status"])
df_train["Race"] = lb_make.fit_transform(df_train["Race"])

df_test["Sex"] = lb_make.fit_transform(df_test["Sex"])
df_test["Native country"] = lb_make.fit_transform(df_test["Native country"])
df_test["Marital status"] = lb_make.fit_transform(df_test["Marital status"])
df_test["Education level"] = lb_make.fit_transform(df_test["Education level"])
df_test["Occupation"] = lb_make.fit_transform(df_test["Occupation"])
df_test["Employment class"] = lb_make.fit_transform(df_test["Employment class"])
df_test["Relationship status"] = lb_make.fit_transform(df_test["Relationship status"])
df_test["Race"] = lb_make.fit_transform(df_test["Race"])
```

The following attributes in the dataset have their values converted to numerical.

Step 5: Data Splitting

Using scikit-learn library, split the original training data into training and testing. Fitting the classifier on the training set, predicting the salary attribute on the testing data and analyzing the accuracy of predictions on test set.

```
from sklearn.model_selection import train_test_split

target = pd.DataFrame(df_train['Salary'])
features = data_train
X_train, X_test, y_train, y_test = train_test_split(features,
                                                    target,
                                                    test_size = 0.3,
                                                    random_state = 0)
```

Note: Although the concat function was an option, to combine both the testing and training data. There was confusion as to how to split the data back into their form. Hence, pre-processing was done simultaneously for both datasets.

Approach to the problem:

To solve our problem, it is a measure of trial and error. In which, all possible classifiers must be trialed against the data and compared in terms of their accuracy. Within this problem, is the subsequent problem of identifying what parameters can be applied to the classifier to provide the best results. To solve this

problem, the following scikit function was used: GridSearchCV. This is a method by sklearn that allows the user to define a set of values to implement for the given classifier, and the function trains the data and finds the best possible combination of hyperparameters that we can use for the classifier.

How does it work in python?

Firstly, we need to import the GridSearchCV from the sklearn library. The estimator function of GridSearchCV required the classifier model that we are using for the hyper-parameter tuning process. The param_grid parameter variable is a list of values to choose from should be given to each hyper parameter of the model. You can change these values and experiment more to see which value ranges give better performance. The GridSearchCV than operates as a cross-validation process, which will provide the hyper-parameter set that will provide the most accurate of results.

Figure 1.1.1, is an example of its implementation in python (**Note**: the code below just an example, not implemented with our data set)

Figure 1.1

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR


gsc = GridSearchCV(
        estimator=SVR(kernel='rbf'),
        param_grid={
            'C': [0.1, 1, 100, 1000],
            'epsilon': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05,
0.1, 0.5, 1, 5, 10],
            'gamma': [0.0001, 0.001, 0.005, 0.1, 1, 3, 5]
        },
        cv=5, scoring='neg_mean_squared_error', verbose=0, n_jobs=-1)
```

For this SVR classifier, the most significant parameters required when working with the rbf kernel of the SVR model are c, gamma and epsilon. These are the parameters that require optimal cross-validation. The cross validation (cv) is to be set at 5, verbose is also set at 5 as it outputs a log to the console. And n_jobs is set at -1 so that all the cores are used on the machine.

Classification techniques used, detailing of parameters set and summary of results produced:

Random Forest Classifier:

Introduction into random forest classifier:

A Random Forest Classifier creates a set of decision trees for various randomly selected subset of training data. It than averages out the results to improve accuracy and control any over-fitting. The sub-sample size is controlled by the parameter *max_samples,* otherwise the classifier will take the whole training set instead to build the decision trees.

Understanding the parameters in the classifier:

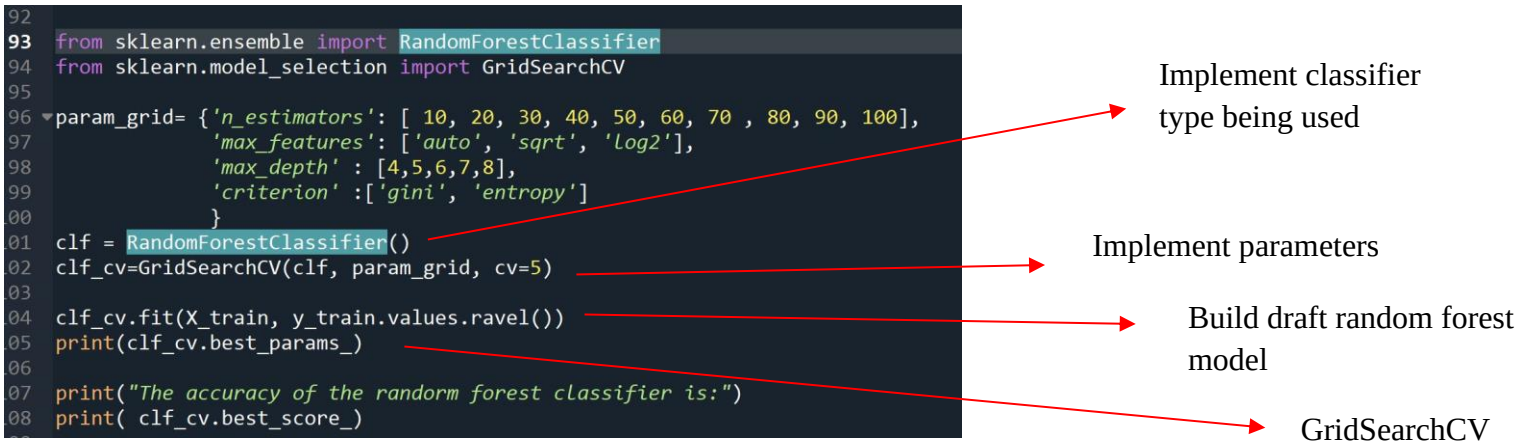
Parameter	Options	Function
N_estimators	Any int> 0, default = 10-100	The number of trees in the forest
Criterion	{“gini”, “entropy”}, default = “gini”	The parameter is used to measure the quality of the split in the decision tree, supported criteria are “gini” for Gini impurity measuring the likelihood of an incorrect classification. Entropy measures the lack of information.
Max_depth	Int, default = None	The maximum depth of the tree, the number of nodes between the first node of the decision tree and the last node. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples
Min_samples_split	Int or float, default = 2	This parameter represents the minimum number of samples required to split an internal node. The internal node is the node created in a decision tree from the original/root node.
Min_samples_leaf	Float, default = 1	The minimum number of samples required for the decision tree to reach the leaf node (end node).
Min_weighted_fraction_leaf	Float, default = 0.0	The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
Max_features	{“auto”, “sqrt”, “log2”} int or float, default=” auto”	These are the maximum number of features Random Forest is allowed to try in an individual tree. 1. *Auto/None*: This will simply take all the features which make sense in every tree. No restrictions are made on the individual tree. 2. sqrt: This option will take square root of the total number of features in the decision tree. For instance, if the total number of variables is 100, we can only take 10 of them in individual tree. 3. 0.2: This option allows the random forest to take 20% of variables in the decision tree. We

		can assign and value in a format “0.x” where we want x% of features to be considered.

Parameters Set for Random Forest Classifier:

Using the GridSearchCV function, as shown in *Figure 1.2* below, the possible parameter values have been inputted. From which the optimal value from each parameter will be selected by the GridSearchCV and printed. A sample random forest classifier has also been inputted in tuning the parameters to get an indication if the accuracy of the classification predictions are improving against the original, un-tuned parameter classifier.

Figure 1.2



After implementing the code in *Figure 1.2* to tune the hyperparameters for the classifier model and then running the code. The program should print the optimal hyperparameters to use for the classifier model, as shown in *Figure 1.3*.

Output – Figure 1.3

Classification Code:

Next, the classifier must be built, this is shown within *Figure 1.4* in which the hyperparameters selected by the function GridSearchCV are implemented to the Random Forest model build.

```
108
109 model=RandomForestClassifier(n_estimators=100, max_features='log2')
110 model.fit(X_train,y_train.values.ravel())
111 prediction=model.predict(X_test)
112 print(accuracy_score(y_test, prediction))
113 print('The accuracy of the random forest classifier is',metrics.accuracy_score(prediction,y_test))
114 print(clf.predict_proba(X_test))
```

Summary of results:

The following results were produced using the random forest classifier:

```
In [15]: runfile('C:/Users/anubh/random forest classifier
- best results so far new.py', wdir='C:/Users/anubh')
{'max_features': 'auto', 'n_estimators': 100}
0.911390977443609
0.9225438596491228
The accuracy of the random forest classifier is
0.9225438596491228
```

'auto' value from parameter and '100' from 'n_estimators' chosen for hyper tuning by GridSearchCV

Accuracy of draft Random Forest classifier

Actual accuracy of Random Forest Classifier model with hyper tuning

When compared against the original Random Forest Classifier without any hyperparameter tuning, the accuracy of its predicting was: 0.9205263157894736

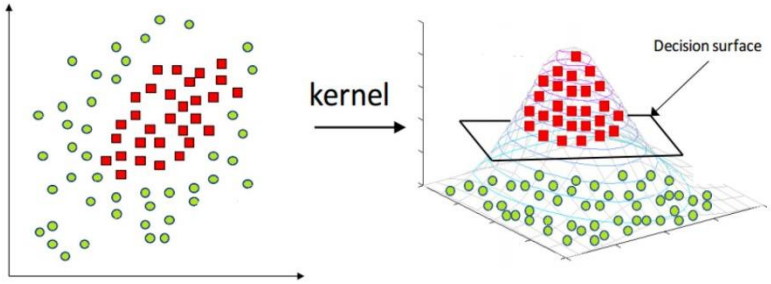
```
The accuracy of the random forest classifier is
0.9205263157894736
[[0.97 0.03]
 [0.82 0.18]
 [1.   0.  ]
 ...
 [1.   0.  ]
 [1.   0.  ]
 [1.   0.  ]]
```

SVC classifier:

Introduction into SVC:

The SVC classifier that uses the SVM (Support Vector Machine) algorithm, performing a linear classification. This involves classification decisions based on the linear value of a combination of attributes. In terms of linear classification, the SVM algorithm simply uses a line or a hyperplane so that it can classify data points into certain classes. When the data being classified has continuous attributes that cannot be linearly classified, the SVC uses a kernel trick.

Figure 1.3



As shown in *Figure 1.3*, if were to map the data from a 2 -dimensional space to a 3-dimensional space, a decision surface could be identified from which data points could be clearly separated into classes. However, the problem with this classification process is that when there are more and more dimensions, computations within that space become more and more expensive.
Hence, the kernel trick can be applied to this problem. It allows us to operate in the original feature space without having to compute coordinates of the data points in a higher dimensional space.

To better understand the SVM model, we can use a mathematical example:

We have two data points, x and y within a 3-dimensional space. We want to map the data points within a 9-dimensional space, however. The following calculations in *Figure 1.4* indicate the level of computational complexity that would occur if were to try and use a simple dimensional mapping transformation. **Note:** This calculation is only determining scalar values of x and y.

Figure 1.4

$$\phi(\mathbf{x}) = (x_1^2, x_1x_2, x_1x_3, x_2x_1, x_2^2, x_2x_3, x_3x_1, x_3x_2, x_3^2)^T$$
$$\phi(\mathbf{y}) = (y_1^2, y_1y_2, y_1y_3, y_2y_1, y_2^2, y_2y_3, y_3y_1, y_3y_2, y_3^2)^T$$

$$\mathbf{x} = (x_1, x_2, x_3)^T$$
$$\mathbf{y} = (y_1, y_2, y_3)^T$$

$$\phi(\mathbf{x})^T\phi(\mathbf{y}) = \sum_{i,j=1}^{3} x_i x_j y_i y_j$$

O (n^2) represents the computational complexity

However, if the kernel function was to be used, which is represented by k (x, y), the complicated computations of determining the positions of x and y in 9-dimensional space can be disregarded. The same scalar values can be determined within the 3-dimensional space using the kernel formula as shown in *Figure 1.5*.

Figure 1.5

$$\begin{aligned} k(\mathbf{x}, \mathbf{y}) &= (\mathbf{x}^T\mathbf{y})^2 \\ &= (x_1y_1 + x_2y_2 + x_3y_3)^2 \\ &= \sum_{i,j=1}^{3} x_i x_j y_i y_j \end{aligned}$$

Understanding the parameters in the classifier:

Parameter	Options	Functions
'C'	Float, default = 1.0	'C' is a regularization parameter. The strength of the regularization is inversely proportional to C. Hence, value must be strictly positive.
kernel	{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'	Specifies the decision boundary used to classify data points. For example, the most basic classification is if a 'linear' kernel were to be used because a straight line (or hyperplane) would be used to make the decision boundary.
Degree	Int, default = 3	Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
gamma	{'scale', 'auto'} or float, default='scale'	Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

Results:

Using the demonstrated code, the accuracy of the predicting classifier SVM is identified:

```
#SVM
from sklearn import svm

svc = svm.SVC(kernel='linear')

svc.fit(X_train, y_train)

y_pred=svc.predict(X_test)

print("Test set predictions:\n {}".format(y_pred))
print(svc.score(X_test,y_test))
```

The following result was produced for this classifier:

```
Test set predictions:
 [0 0 0 ..., 0 0 0]
0.773111979167
```

Decision Tree classifier:

What does the decision tree classifier do?

The decision tree classifier is used to classify data in the form of a tree structure. Breaking data down into smaller and smaller subsets whilst simultaneously an associated decision tree is incrementally developed through the process. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches. Leaf node represents a classification or decision. The topmost decision node in a tree which corresponds to the best predictor called root node.

Pros:
A decision tree classifier is computationally inexpensive, and thus can be easy to read for analysis purposes. This classifier can also deal with insignificant variables and outliers.

Cons:
Decisions tree classifiers are known to be prone to overfitting, meaning that the classifier is trained on the training dataset too well that any single outlier or form of noisy data can degrade the performance of the model.

Understanding the parameters in the classifier:

Parameters	Options	Functions
Criterion	{“gini”, “entropy”}, default=” gini”	The parameter is used to measure the quality of the split in the decision tree, supported criteria are “gini” for Gini impurity measuring the likelihood of an incorrect classification. Entropy measures the lack of information.
Splitter	{“best”, “random”}, default= “best”	The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
Max_depth	int, default=None	The most influential parameter for a decision tree. This parameter is the number of nodes between the root node and the last leaf node.

Exploring the parameter Max_depth:

What is the boundaries for this parameter and what provides the best accuracy for the decision tree classifier?

The depth of the tree is known as a hyperparameter, which means a parameter you need to decide before you fit the model to the data.

The Max_depth can cause significant effects to the decision tree classifier. The following outcomes could occur based on the value implemented to the classifier:

- If max_depth is too high, then the decision boundary will become very complex. This could potentially lead to the classifier becoming prone to overfit the data. Meaning that slight noises and outliers in data would degrade the performance and the accuracy of the decision tree classifier
- If max_depth was too low, the classifier could potentially underfit the data, meaning that the model does not contain enough signal to perform.

In order to identify the best integer to tune for this hyperparameter, the original training data can be split into a local training and testing dataset. You can then fit the model to your training data, make predictions on your test set and see how well your prediction does on the test set. This can be achieved from implementing the following code in *Figure 1.9*.

Figure 1.9

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42, stratify=y)
```

Now, to identify the optimal value for max_depth that will provide the best accuracy for the decision tree in both the training and testing data, we can create a plot graph.

Using values for max_depth ranging from 1-9 we can identify the optimal value that can be hyper tuned for this decision tree classifier. *Figure 2.1*, shows the code used to identify the best possible value for max_depth.

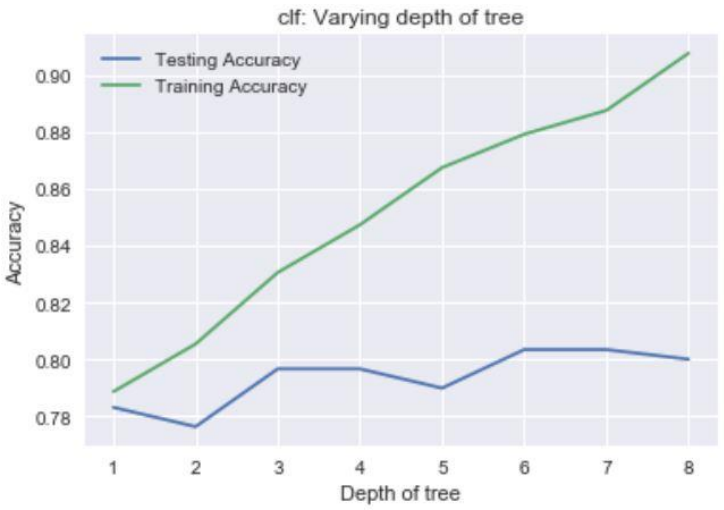
Figure 2.1

```
# Setup arrays to store train and test accuracies
dep = np.arange(1, 9)
train_accuracy = np.empty(len(dep))
test_accuracy = np.empty(len(dep))

# Loop over different values of k
for i, k in enumerate(dep):
    # Setup a Decision Tree Classifier
    clf = tree.DecisionTreeClassifier(max_depth=k)

    # Fit the classifier to the training data
    clf.fit(X_train, y_train)

    #Compute accuracy on the training set
    train_accuracy[i] = clf.score(X_train, y_train)

    #Compute accuracy on the testing set
    test_accuracy[i] = clf.score(X_test, y_test)
```

```
# Generate plot
plt.title('clf: Varying depth of tree')
plt.plot(dep, test_accuracy, label = 'Testing Accuracy')
plt.plot(dep, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.xlabel('Depth of tree')
plt.ylabel('Accuracy')
plt.show()
```

From this, the following graph was produced, *Figure 2.2*

Figure 2.2



From *2.2*, it can be identified that as the max_depth is increased the accuracy of the training data continues to increase. This is because as the depth of the decision tree continue to increase, the data will fit better and better into the classifier as more and more predictions are made that reflect the training data. But with an increase in max_depth showing increased accuracy for the training data, this also means that the classifier is learning the details of the training data too well. And thus, the noisy data and outliers are negatively affecting the models performance. Hence, the testing data plot stays at a gradually stagnant path.

Using the GridSearchCV we can find the optimal value of the hyper parameter **max_depth** using the process shown in *Figure 2.3*.

Figure 2.3

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn import metrics

param_grid= {'max_depth': np.arange(0.1,20)
             }

clf = DecisionTreeClassifier()
clf_cv=GridSearchCV(clf, param_grid, cv=5)
clf_cv.fit(X_train, y_train.values.ravel())
print(clf_cv.best_params_)

print( clf_cv.best_score_)
```

Identifies optimal max_depth values between 0.1 and 20

K-Neighbours Classifier:

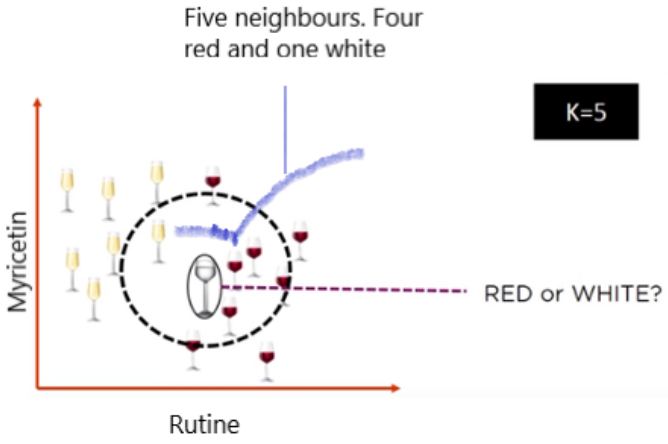
Introduction into K-Neighbours:

K-nearest classifier uses the K-nearest neighbour algorithm to classify data points. This is a very simple process that works on the basis of storing all the available cases and classifies the new data or case based on a similarity measure. It is mostly used to classifies a data point based on how its neighbours are classified. ‘k’ in KNN is a parameter that refers to the number of nearest neighbours to include in the majority of the voting process.

How does the classifier work?

Looking at an example, suppose a new wine was added to a dataset containing both white and red wine as the data points. To predict whether the newly added wine is either red or white wine the KNN classifier would use the neighbours within the dataset to predict what type of wine was newly added. For instance, if k were equal to 5 and the new data point is would be classified of its wine type based upon the nearest 5 neighbours. As shown in the *Figure 2.4*, the new point would be classified as red since four out of five neighbours are red.

Figure 2.4



Knn classifier parameter tuning:

‘k’ is the main parameter used within this classifier. It is the parameter that sets the number of neighbours to use in the classifier. The default for this parameter is 5.

To determine the optimal k value for the Knn classifier, certain details must be understood.

There is no structured method to find the best value for “K”. However, certain principles do come to good use:

- Choosing smaller values of k can result in the data becoming noisy and could significantly influence the performance of the classifier predicter
- Choosing larger k values will generally result in smoother boundaries, meaning lower variance in the data but a higher bias. Having a higher-level k value also is much more computationally expensive

Another way to choose K is though cross-validation. One way to select the cross-validation dataset from the training dataset. Take the small portion from the training dataset and call it a validation dataset, and then use the same to evaluate different possible values of K. This can be achieved using the GridSearchCV function. *Figure 2.5*, demonstrates the process of finding the optimal k value for the KNN classifier.

Figure 2.5

```
In [1]: runfile('C:/Users/anubh/knieghbour classifier.py',
wdir='C:/Users/anubh')
{'n_neighbors': 14}
```

Hence, the optimal hyperparameter for the KNN classifier is 14. This can be analysed and understood due to large amount of data, in which the k values is not small enough to overfit the data. For example, for K = 3 and a really large data set, there is a reasonable chance that there will be two noisy data points that are close enough to each other to outvote the correct data points in some region. On the other hand, if K is too large of a value, then there may be too much smoothing within the dataset and thus could eliminate important details. In the extreme case, if we choose K to be equal to the number of data points, and for example there was more red wines than white wines, then the whole distribution would be red wines.

Gaussian NB Classifier:

Naive Bayes is a statistical classification technique based on Bayes Theorem. It is one of the simplest supervised learning algorithms. Naive Bayes classifier is the fast, accurate and reliable algorithm. Naive Bayes classifiers have high accuracy and speed on large datasets.

Naive Bayes classifier assumes that the effect of a particular feature in a class is independent of other features. For example, a loan applicant is desirable or not depending on his/her income, previous loan and transaction history, age, and location. Even if these features are interdependent, these features are

still considered independently. This assumption simplifies computation, and that's why it is considered as naive.

Parameters:

For this NB classifier, parameters can be kept at default.

Training and Evaluation model:

To train the data, we can combine Gaussian naïve Bayes classifier with quantile transformer as shown in *Figure 2.6.*

Figure 2.6

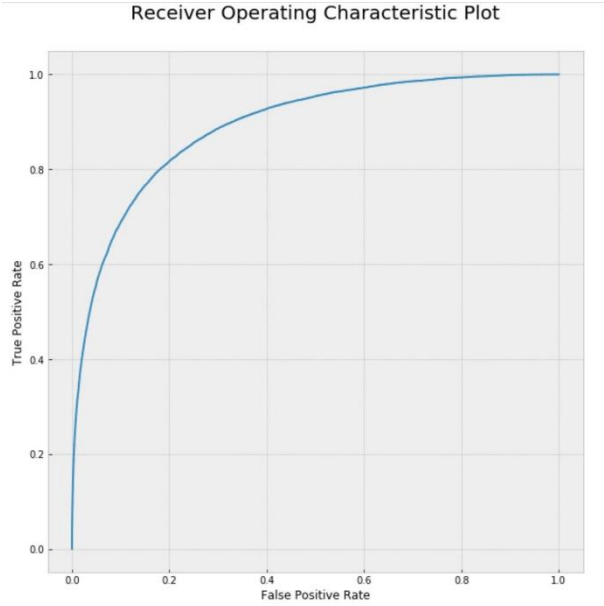
```
from sklearn.preprocessing import QuantileTransformer
from sklearn.pipeline import make_pipeline

pipeline = make_pipeline(QuantileTransformer(output_distribution='normal'), GaussianNB())
pipeline.fit(X_train, y_train)
```

ROC curve:

```
fpr, tpr, thr = roc_curve(y_train, pipeline.predict_proba(X_train)[:,1])
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Plot', **title_config)
auc(fpr, tpr)
```

```
title_config = {'fontsize': 20, 'y': 1.05}
```

After training the model, we plot the ROC curve on training data and evaluate the model by computing the training AUC and cross-validation AUC. We can use sklearn.metrics.roc_curve to obtain the values for plotting the curve and sklearn.metrics.auc for computing the AUC.



Results:

```
In [7]: runfile('C:/Users/anubh/untitled1.py', wdir='C:/
Users/anubh')
The accuracy of the random Gaussian naive bayes classier
is 0.6169298245614036
[[1.00000000e+00 2.30267592e-12]
 [9.74535880e-01 2.54641197e-02]
 [1.00000000e+00 9.34872304e-24]
 ...
 [1.00000000e+00 3.14101285e-28]
 [1.00000000e+00 3.33741560e-21]
 [1.00000000e+00 3.97436595e-14]]
```

As we can see from the ROC for the Gaussian Naïve Bayes Classifier, the curve does not continue to follow the left-hand border, this

The curve also is closer to coming to a 45-degree diagonal of the left – hand ROC space, this also tends to suggest that the model is less accurate. However, when analysing the AUC of the curve it was determined as 0.8592933235123672 that suggests that the NB classifier is well performing.

Which classifier performed the best?

After, testing each classifier against the training and testing dataset. The random forest classifier can be determined as the best performing classifier in predicting the salary attribute.

Random forests are bagged decision tree models that split on a subset of features on each split. This is a huge mouthful, so let's break this down by first looking at a single decision tree, then discussing bagged decision trees and finally introduce splitting on a random subset of features.

Random forest improves on bagging because it decorrelates the trees with the introduction of splitting on a random subset of features. This means that at each split of the tree, the model considers only a small subset of features rather than all of the features of the model. That is, from the set of available features n, a subset of m features (m=square root of n) are selected at random. This is important so that variance can be averaged away. Consider what would happen if the data set contains a few strong predictors. These predictors will consistently be chosen at the top level of the trees, so we will have very similar structured trees. In other words, the trees would be highly correlated.
In summary, this classifier goes most in depth of the data, especially breaking them into buckets of data makes the predicting process much higher quality.
The results from predicting the salary are down below:

```
In [15]: runfile('C:/Users/anubh/random forest classifier
- best results so far new.py', wdir='C:/Users/anubh')
{'max_features': 'auto', 'n_estimators': 100}
0.911390977443609
0.9225438596491228
The accuracy of the random forest classifier is
0.9225438596491228
```

Reflection:
From this assignment, I have come to better understanding of the working of Python, and the processes and the functions behind it. I have also found a deeper insight into the work of a Data Scientist, what often seemed an unclear image of what a data scientist does and why they have such a high demand in the job market has become much clearer.
In terms of the classification models, I've come to an understanding that its not just the classifier that counts to better accuracy, but the process and the parameters used.