Linear Classification and Logistic Regression

In machine learning, classification is a supervised method of segmenting data points into various labels or classes. Unlike regression, the target variable in a classification problem is discrete. Each data point used in training classification models must have a corresponding label in order for the characteristics and patterns in the classes to be learnt appropriately. Classification can either be binary - identifying that a given email is spam or not or, multi-class - classifying a fruit as orange, mango or banana.

All codes used in this lesson can be found here.:- https://gist.github.com/HamoyeHQ/94d52ad113d1eac80d073a4affb0a490

Gentle Introduction

Every year people demand more from nature than it can regenerate. Individuals, communities and government leaders use ecological footprint data to better manage limited resources, reduce economic risk, and improve well-being. The Dataset provides Ecological Footprint per capita data for years 1961-2016 in global hectares (gha). Ecological Footprint is a measure of how much area of biologically productive land and water an individual, population, or activity requires to produce all the resources it consumes and to absorb the waste it generates, using prevailing technology and resource management practices. The Ecological Footprint is measured in global hectares. Since trade is global, an individual or country's Footprint tracks area from all over the world.

Apart from predicting numeric values, another important supervised machine learning method is classification and it involves predicting classes (either binary or multinomial classes). In this section, we will cover how to measure performances of class prediction, linear classification methods and non-linear/tree-based methods. We'll also focus on strategies for applying a successful classification model like interpretability-accuracy trade-off, class and imbalance.

The National Footprint and Biocapacity Accounts (NFAs) measure the ecological resource use and resource capacity of nations from 1961 to 2016. The calculations in the National Footprint and Biocapacity Accounts are primarily based on United Nations data sets, including those published by the Food and Agriculture Organization, United Nations Commodity Trade Statistics Database, and the UN Statistics Division, as well as the International Energy Agency. In this project, we will use this data to classify and predict the quality metrics (qascore) of the ecological footprint data for the different countries. This data includes total and per capita national biocapacity, the ecological footprint of consumption, the ecological footprint of production and total area in hectares.

Data Source: https://data.world/footprint/nfa-2019-edition

Linear Classification & Logistic Regression

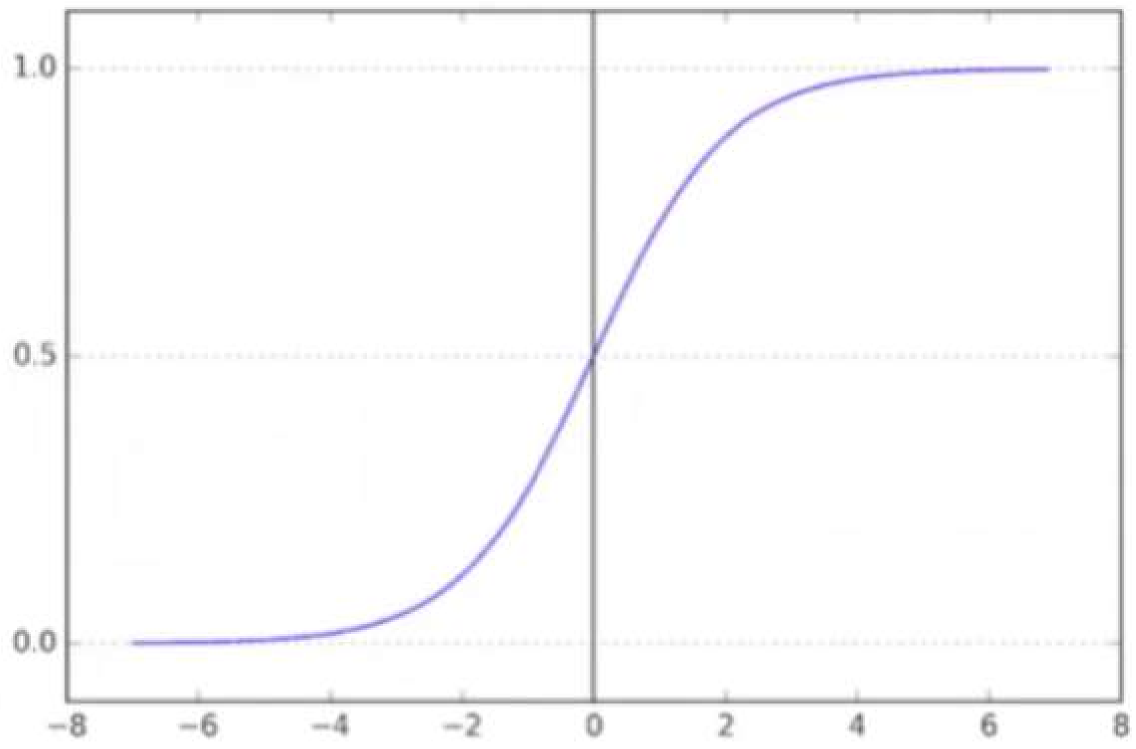
Linear classifiers and the importance of class probabilities

For simplicity, we define a linear classifier as a binary classifier that separates two classes (positive and negative class) using a linear separator by computing a linear combination of the features and comparing against a set threshold.

Logistic Regression: Sigmoid, logit and the log-likelihood

Logistic regression is a linear algorithm that can be used for binary or multiclass classification. It is a discriminative classifier that estimates the probability that an instance belongs to a class using an s-shape function curve called the sigmoid function. The predicted values obtained after using a linear equation on the predictors by applying logistic regression can fall in the range of negative infinity to positive infinity. The sigmoid maps these results by shrinking the value to fall between 0 and 1. We can say that we use the sigmoid function to transform linear regression into logistic regression.

$$sigmoid\ \sigma(x) = \frac{1}{1 + e^{-x}}$$



The sigmoid function can be applied to a linear equation

$$z = \beta_0 + \beta_1 x$$

to obtain values h between 0 and 1 such that:

$$h = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}}$$

For a binary classification task with classes A and B, if a threshold is set for 0.5 and the probability of an instance belonging to a class is p, we can say that if $p < 0.5$ the instance if of class A while it is of class B is $p > 0.5$.

Also known as the log of odds, logit is the logarithm of odds ratio where the odds ratio is the probability that an event occurs divided by the probability that the event does not occur. Logit is the inverse of the sigmoid such that it maps values from negative infinity to positive infinity.

Linear Classification and Logistic Regression

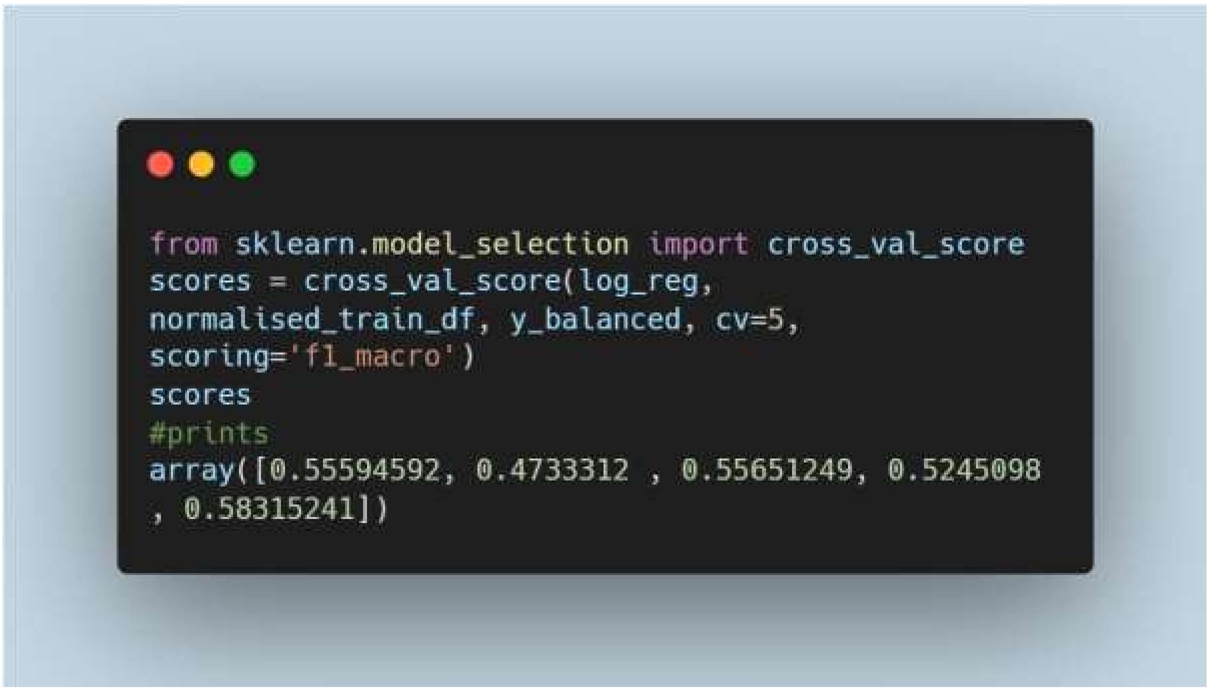
```
df = pd.read_csv('https://query.data.world/s/wh6j7rxy2hvrn4ml75ci62apk5hgae')
#check distribution of target variable
#prints
3A    51481
2A    10576
2B    10096
1B       16
1A       16
Name: QScore, dtype: int64

df['QScore'].value_counts()
df.isna().sum()
#for simplicity, we will drop the rows with missing values.
df = df.dropna()
df.isna().sum()
#An obvious change in our target variable after removing the missing values is that there
are only three classes left #and from the distribution of the 3 classes, we can see that
there is an obvious imbalance between the classes. #There are methods that can be applied to
handle this imbalance such as oversampling and undersampling.
#Oversampling involves increasing the number of instances in the class with fewer instances
while undersampling #involves reducing the data points in the class with more instances.
#For now, we will convert this to a binary classification problem by combining class '2A'
and '1A'.
df['QScore'] = df['QScore'].replace(['1A'], '2A')
df.QScore.value_counts()
#prints
3A    51473
2A      240
Name: QScore, dtype: int64

df_2A = df[df.QScore=='2A']
df_3A = df[df.QScore=='3A'].sample(350)
data_df = df_2A.append(df_3A)

import sklearn.utils
data_df = sklearn.utils.shuffle(data_df)
data_df = data_df.reset_index(drop=True)
data_df.shape
data_df.QScore.value_counts()
#prints
3A    350
2A    240
Name: QScore, dtype: int64

#more preprocessing
data_df = data_df.drop(columns=['country_code', 'country', 'year'])
```

```
X = data_df.drop(columns='QScore')
y = data_df['QScore']

#split the data into training and testing sets
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
y_train.value_counts()
#prints
3A    242
2A    171
Name: QScore, dtype: int64
#There is still an imbalance in the class distribution. For this, we use SMOTE only on the
training data to handle this.

#encode categorical variable
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
x_train.record = encoder.fit_transform(x_train.record)
x_test.record = encoder.transform(x_test.record)

import imblearn
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=1)
x_train_balanced, y_balanced = smote.fit_sample(x_train, y_train)

from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
normalised_train_df = scaler.fit_transform(x_train_balanced.drop(columns=['record']))
normalised_train_df = pd.DataFrame(normalised_train_df,
columns=x_train_balanced.drop(columns=['record']).columns)
normalised_train_df['record'] = x_train_balanced['record']

x_test = x_test.reset_index(drop=True)
normalised_test_df = scaler.transform(x_test.drop(columns=['record']))
normalised_test_df = pd.DataFrame(normalised_test_df,
columns=x_test.drop(columns=['record']).columns)
normalised_test_df['record'] = x_test['record']



#Logistic Regression
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(normalised_train_df, y_balanced)
#returns
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Measuring Classification Performance

Cross-validation and accuracy

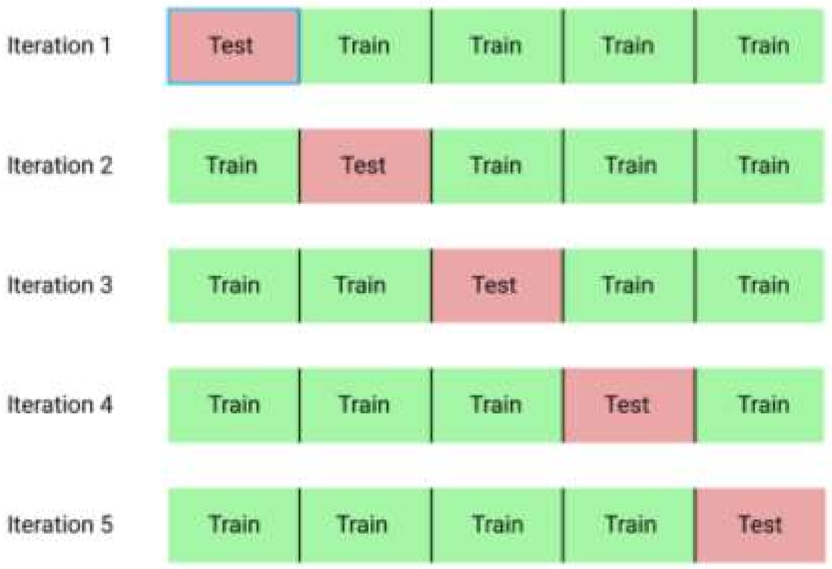
From the previous module, we now understand why data scientists and machine learning engineers avoid having models that overfit or underfit. Cross Validation (CV) is a well known and trusted method applied to avoid overfitting and enable generalization. Although there are different techniques used in performing cross validation, the fundamental concept involves partitioning the dataset into a number of subsets, holding out a set for evaluation then training the model on the other sets. This gives a more reliable estimate of how the model performs across different training sets because it provides an average score across different training samples used. The only drawback with cross validation is that it takes more time and computational resources however, the gain obtained in having a better model is very well worth this cost. K-Fold cross validation, Stratified K-Fold cross validation and Leave One Out Cross Validation (LOOCV) are some cross validation techniques.

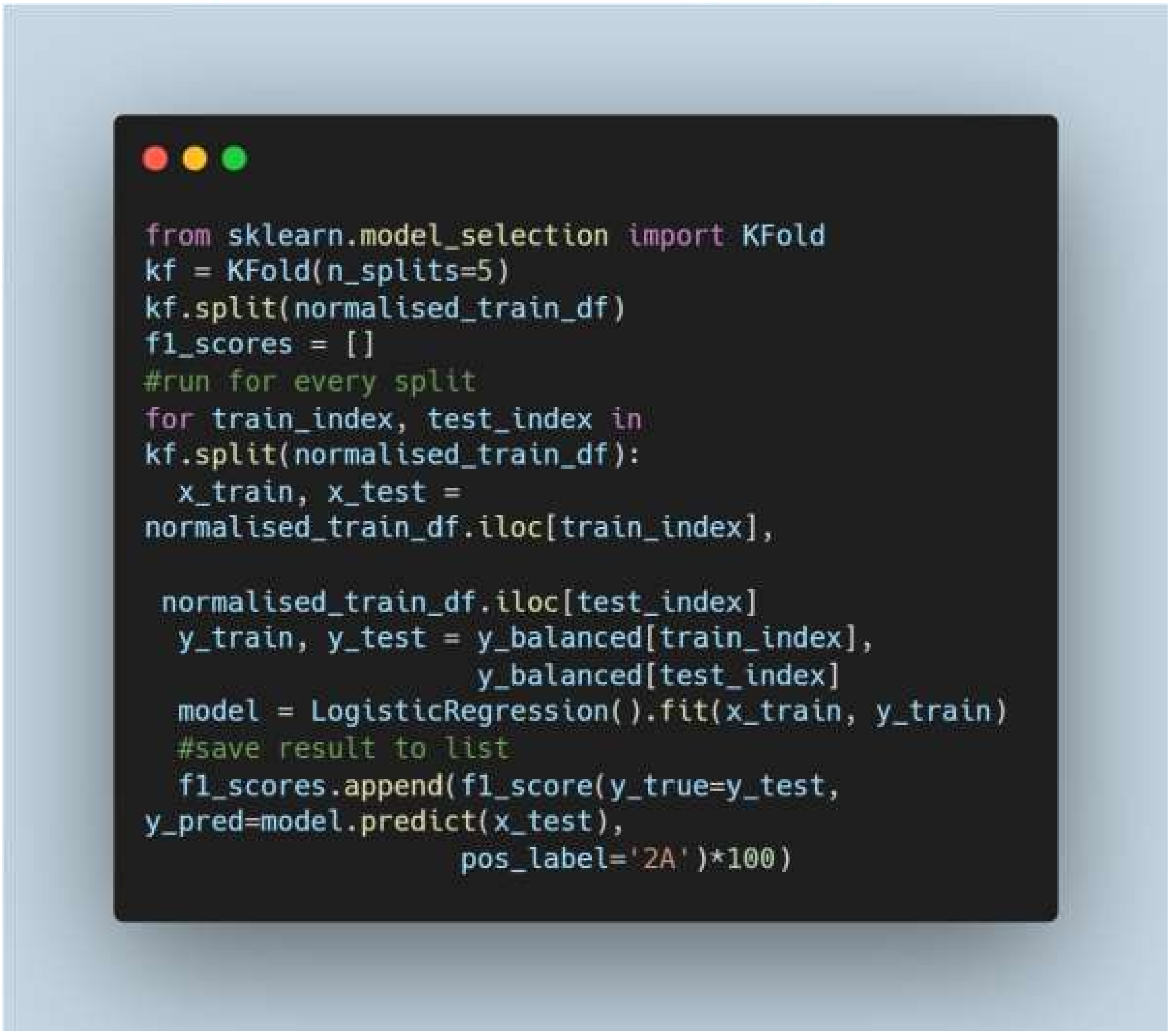


```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(log_reg,
normalised_train_df, y_balanced, cv=5,
scoring='f1_macro')
scores
#prints
array([0.55594592, 0.4733312 , 0.55651249, 0.5245098
, 0.58315241])
```

K-Fold Cross Validation

This technique is called K-Fold because the data is split into K equal groups. If k=5,a 5-fold cross validation can be performed such that the data is split into k1, k2, k3, k4 and k5. The model is trained on k2 - k5 and evaluated on k1 then repeated k times until every group is used to train and test the model.

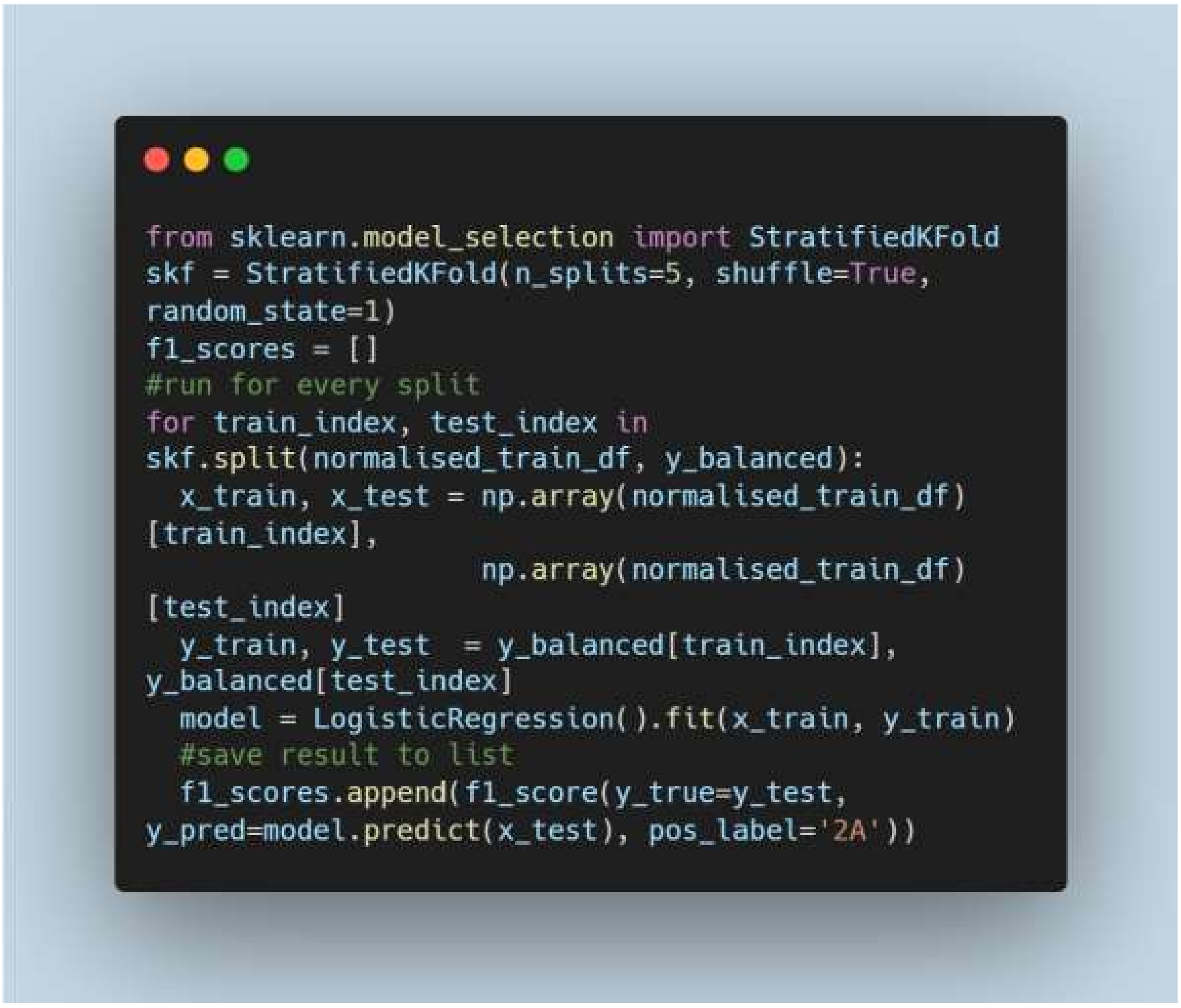




```
from sklearn.model_selection import KFold
kf = KFold(n_splits=5)
kf.split(normalised_train_df)
f1_scores = []
#run for every split
for train_index, test_index in
kf.split(normalised_train_df):
  x_train, x_test =
normalised_train_df.iloc[train_index],

 normalised_train_df.iloc[test_index]
  y_train, y_test = y_balanced[train_index],
                    y_balanced[test_index]
  model = LogisticRegression().fit(x_train, y_train)
  #save result to list
  f1_scores.append(f1_score(y_true=y_test,
y_pred=model.predict(x_test),
                   pos_label='2A')*100)
```

Stratified K-Fold Cross Validation

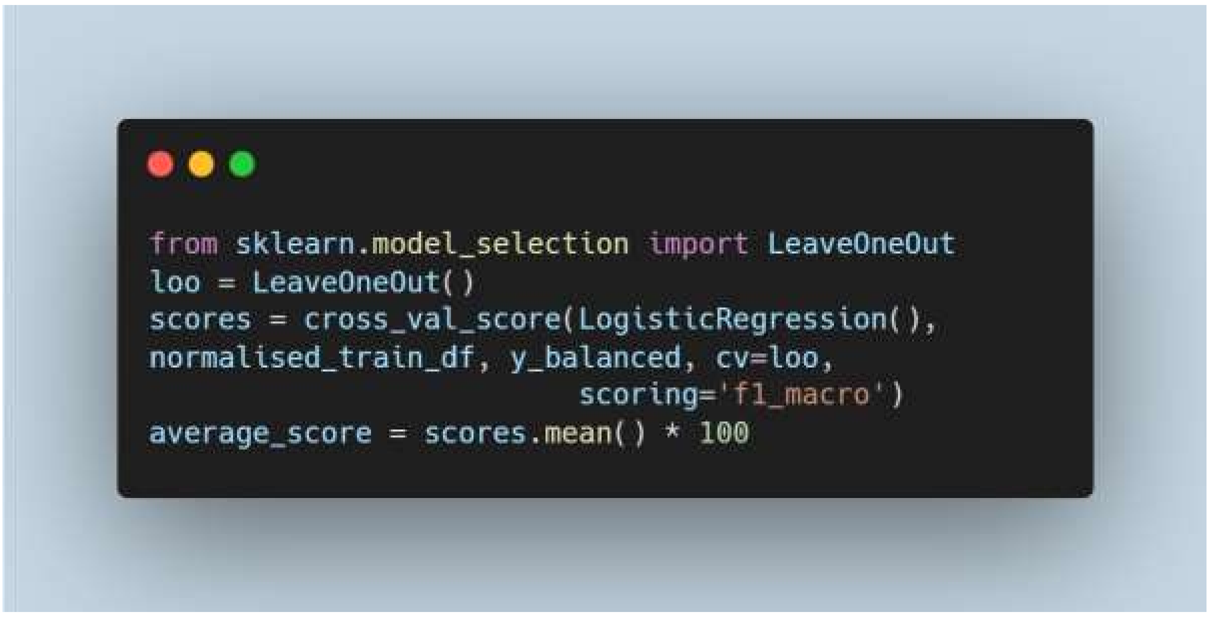
Although similar to the technique described above, Stratified K-Fold cross validation ensures that in every fold, there is an equal proportion of each target class to obtain a good representation of the data and avoid imbalance and biased results. For example, if there are two target classes t1 and t2 with equal distribution in the data, it is best to ensure that the folds also have the same distribution.



```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=1)
f1_scores = []
#run for every split
for train_index, test_index in
skf.split(normalised_train_df, y_balanced):
  x_train, x_test = np.array(normalised_train_df)
[train_index],
                    np.array(normalised_train_df)
[test_index]
  y_train, y_test  = y_balanced[train_index],
y_balanced[test_index]
  model = LogisticRegression().fit(x_train, y_train)
  #save result to list
  f1_scores.append(f1_score(y_true=y_test,
y_pred=model.predict(x_test), pos_label='2A'))
```

Leave One Out Cross Validation (LOOCV)

In this method, one instance is left out and used as the test set while the model is trained on N-1data points where N is the number of data points. This means that the number of instances and folds are equal.



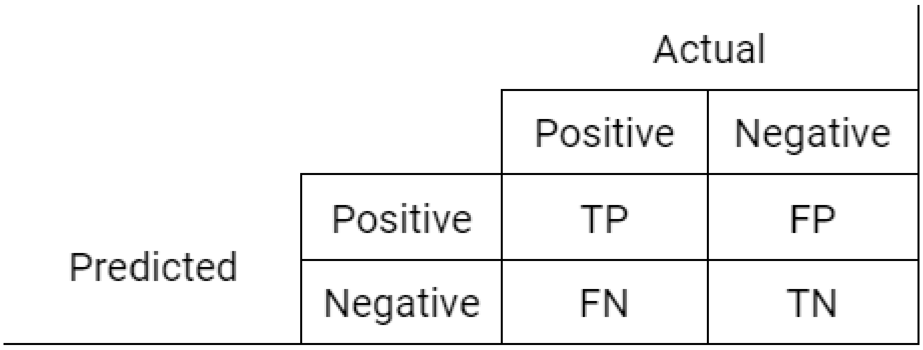
```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(LogisticRegression(),
normalised_train_df, y_balanced, cv=loo,
                            scoring='f1_macro')
average_score = scores.mean() * 100
```

Confusion Matrix, Precision-Recall, ROC curve and the F1-score

Accuracy, precision, recall, F1-score and many others are evaluation metrics used in measuring the performance of classification models. In this section, we discuss these metrics.

Confusion Matrix

It is an N x N matrix that gives a summary of the correct and incorrect predicted classification results for the *N*target classes. The values in the diagonal of the matrix represent the number of correctly predicted classes while every other cell in the matrix indicates the misclassified classes. This means that the more predicted values that fall in the diagonal, the better the model. True positive, false positive, true negative and false negative are terms used when interpreting a confusion matrix.



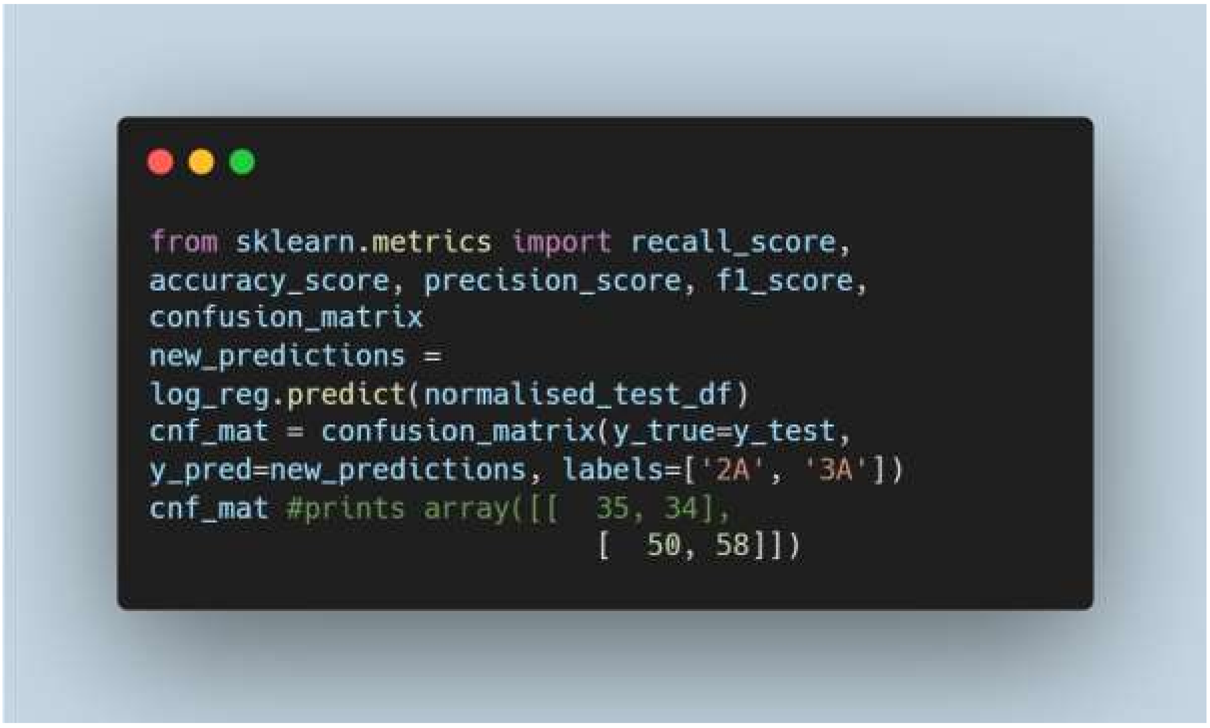
		Actual	
		Positive	Negative
Predicted	Positive	TP	FP
	Negative	FN	TN

- True Positive (TP): This is a correct classification where the predicted value is the same as the actual value. Using the table above, this means that actual value was positive and the predicted value was also positive.

- True Negative (TN): The predicted value also matches the actual value. In this case, it is for the negative class. The actual value is negative and the predicted value is negative.

- False Positive (FP): Also called a Type I error, this is a misclassification such that the model predicted a positive class while the actual class is negative. Telling a man that he is pregnant is definitely a false positive.

- False Negative (FN): Also another misclassification where the predicted value is negative and the actual value is positive. Another example will be telling a pregnant woman that she is not pregnant. FN is known as a Type II error.

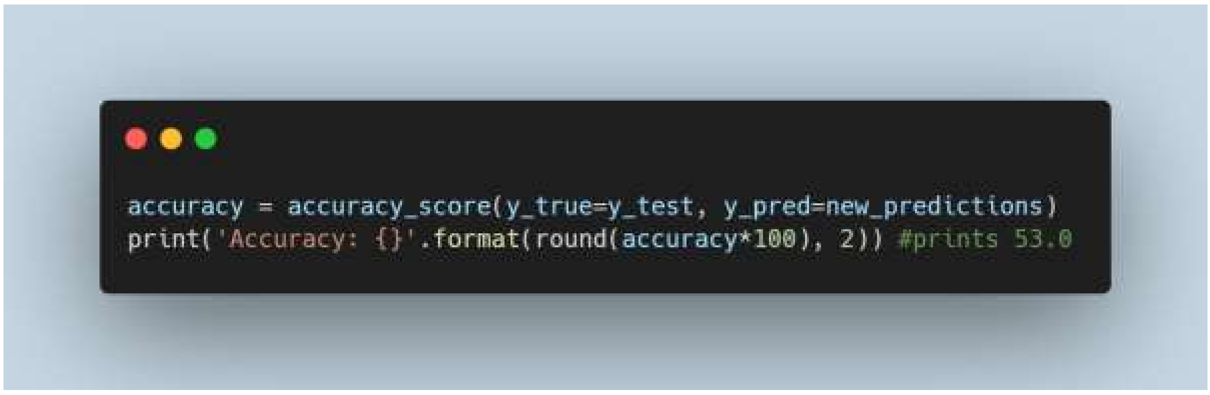


```
from sklearn.metrics import recall_score,
accuracy_score, precision_score, f1_score,
confusion_matrix
new_predictions =
log_reg.predict(normalised_test_df)
cnf_mat = confusion_matrix(y_true=y_test,
y_pred=new_predictions, labels=['2A', '3A'])
cnf_mat #prints array([[  35, 34],
                       [  50, 58]])
```

Accuracy

This is the ratio of the number of correctly predicted instances to the total number of instances. It is a commonly used metric suitable when the target classes are not imbalanced. A high accuracy does not necessarily mean that the model has high predicting power. Hence, depending on the task, it is important to not use only the accuracy metric because it does not provide enough information about the model.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

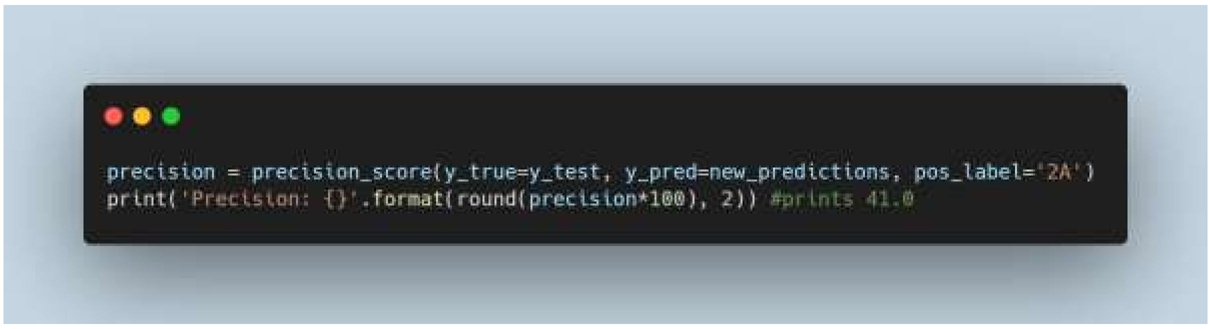


```
accuracy = accuracy_score(y_true=y_test, y_pred=new_predictions)
print('Accuracy: {}'.format(round(accuracy*100), 2)) #prints 53.0
```

Precision

The ratio of correctly predicted instances of a class to the total number of items predicted by the model to be in that class is referred to as precision (known as Positive Predicted Value - PPV). This translates to the total percentage of the results obtained that are relevant. For the positive class, it is the ratio of true positives to the sum of true positives and false positives

$$Precision = \frac{TP}{TP + FP}$$



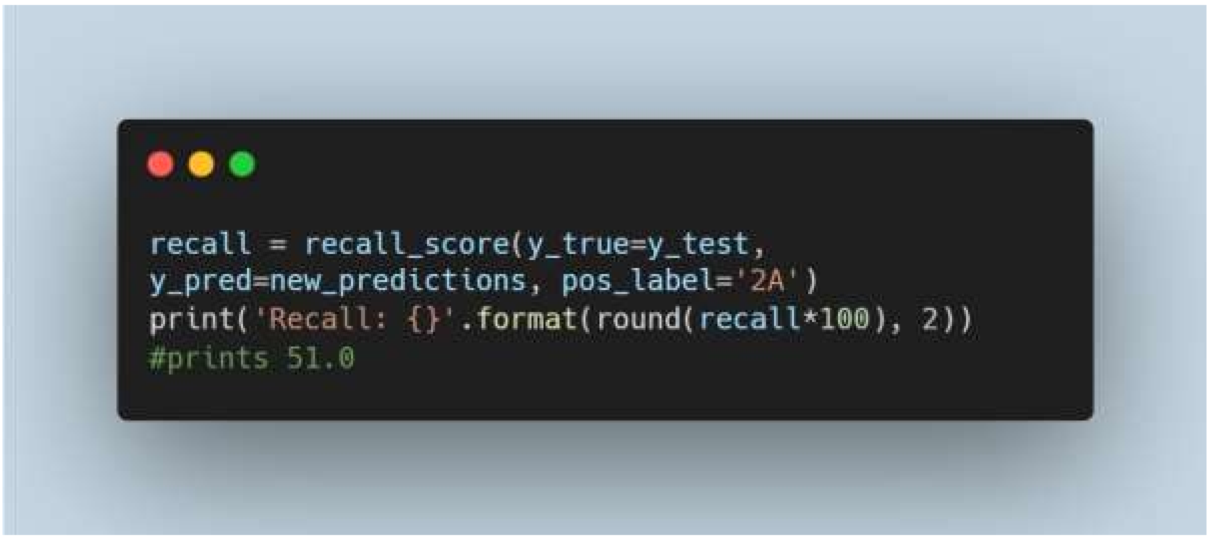
```
precision = precision_score(y_true=y_test, y_pred=new_predictions, pos_label='2A')
print('Precision: {}'.format(round(precision*100), 2)) #prints 41.0
```

Recall

Known as the sensitivity of the model, recall gives a percentage of total relevant results correctly predicted by the model. It is the ratio of the true positives to the actual number of positives (true positives and false negatives).

$$Recall = \frac{TP}{TP + FN}$$

Like in the previous module where we discussed the bias-variance trade-off, there is also a trade-off between precision and recall. It is impossible to maximise both metrics simultaneously because an increase in recall decreases precision. Identify which metric is important based on your task and optimise.

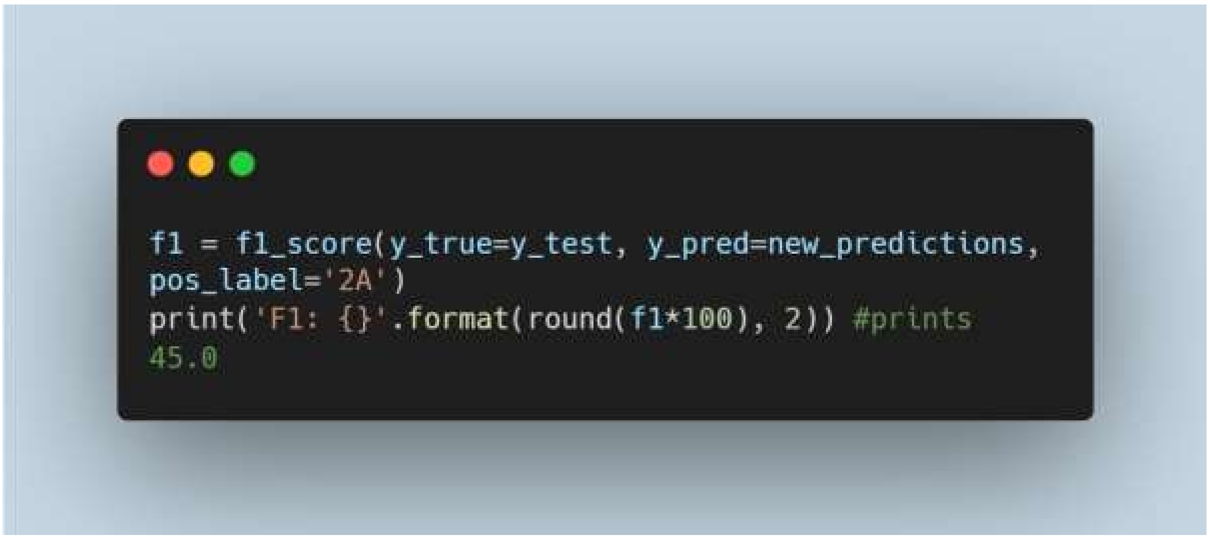


```
recall = recall_score(y_true=y_test,
y_pred=new_predictions, pos_label='2A')
print('Recall: {}'.format(round(recall*100), 2))
#prints 51.0
```

F1-Score

This metric is the harmonic mean of precision and recall that aims to have an optimal balance of both. The F1-Score is quite easy to use and can be focused on to maximize as opposed to maximizing precision and recall.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$



```
f1 = f1_score(y_true=y_test, y_pred=new_predictions,
pos_label='2A')
print('F1: {}'.format(round(f1*100), 2)) #prints
45.0
```

ROC Curve

The Receiver Operating Characteristics (ROC) curve is a probability curve that measures the performance of a classification model at different set thresholds. Recall also known as the True Positive Rate (TPR) is plotted on the y-axis against the False Positive Rate (FPR) on the x-axis.

The code examples above are not the optimal results that can be obtained with the model. Hyperparameter tuning can be performed to improve the model.

https://colab.research.google.com/notebooks/mlcc/logistic_regression.ipynb?hl=en

Measuring Classification Performance

Cross-validation and accuracy

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(log_reg, normalised_train_df, y_balanced, cv=5, scoring='f1_macro')
scores
#prints
array([0.55594592, 0.4733312 , 0.55651249, 0.5245098 , 0.58315241])
```

K-Fold Cross Validation

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=5)
kf.split(normalised_train_df)
f1_scores = []
#run for every split
for train_index, test_index in kf.split(normalised_train_df):
  x_train, x_test = normalised_train_df.iloc[train_index],
                    normalised_train_df.iloc[test_index]
  y_train, y_test = y_balanced[train_index],
                    y_balanced[test_index]
  model = LogisticRegression().fit(x_train, y_train)
  #save result to list
  f1_scores.append(f1_score(y_true=y_test, y_pred=model.predict(x_test),
                   pos_label='2A')*100)
```

Stratified K-Fold Cross Validation

```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
f1_scores = []
#run for every split
for train_index, test_index in skf.split(normalised_train_df, y_balanced):
  x_train, x_test = np.array(normalised_train_df)[train_index],
                    np.array(normalised_train_df)[test_index]
  y_train, y_test  = y_balanced[train_index], y_balanced[test_index]
  model = LogisticRegression().fit(x_train, y_train)
  #save result to list
  f1_scores.append(f1_score(y_true=y_test, y_pred=model.predict(x_test), pos_label='2A'))
```

Leave One Out Cross Validation (LOOCV)

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(LogisticRegression(), normalised_train_df, y_balanced, cv=loo,
                         scoring='f1_macro')
average_score = scores.mean() * 100
```

Confusion Matrix

```
from sklearn.metrics import recall_score, accuracy_score, precision_score, f1_score,
confusion_matrix
new_predictions = log_reg.predict(normalised_test_df)
cnf_mat = confusion_matrix(y_true=y_test, y_pred=new_predictions, labels=['2A', '3A'])
cnf_mat #prints       array([[  35, 34],
                             [  50, 58]])
```

Accuracy

```
accuracy = accuracy_score(y_true=y_test, y_pred=new_predictions)
print('Accuracy: {}'.format(round(accuracy*100), 2)) #prints 53.0
```

Precision

```
precision = precision_score(y_true=y_test, y_pred=new_predictions, pos_label='2A')
print('Precision: {}'.format(round(precision*100), 2)) #prints 41.0
```

Recall

```
recall = recall_score(y_true=y_test, y_pred=new_predictions, pos_label='2A')
print('Recall: {}'.format(round(recall*100), 2)) #prints 51.0
```

F1-Score

```
f1 = f1_score(y_true=y_test, y_pred=new_predictions, pos_label='2A')
print('F1: {}'.format(round(f1*100), 2)) #prints 45.0
```

Multiclass Classification

Multilabel and Multiclass classification

Multiclass classification deals with more than two classes where an instance is classified into a single class. For example, given a dataset with a set of features that describe the weather such that the classes are sunny, rainy and windy, a multiclass classification task will only give a single class as the result. In contrast, multilabel classification classifies an instance into a set of target labels. Articles and movies are examples where this can apply. An article can discuss a single topic but can also be about politics, religion, education and many more while movies are commonly tagged to multiple genres such as comedy, adventure, action.

The Sigmoid and the Softmax function

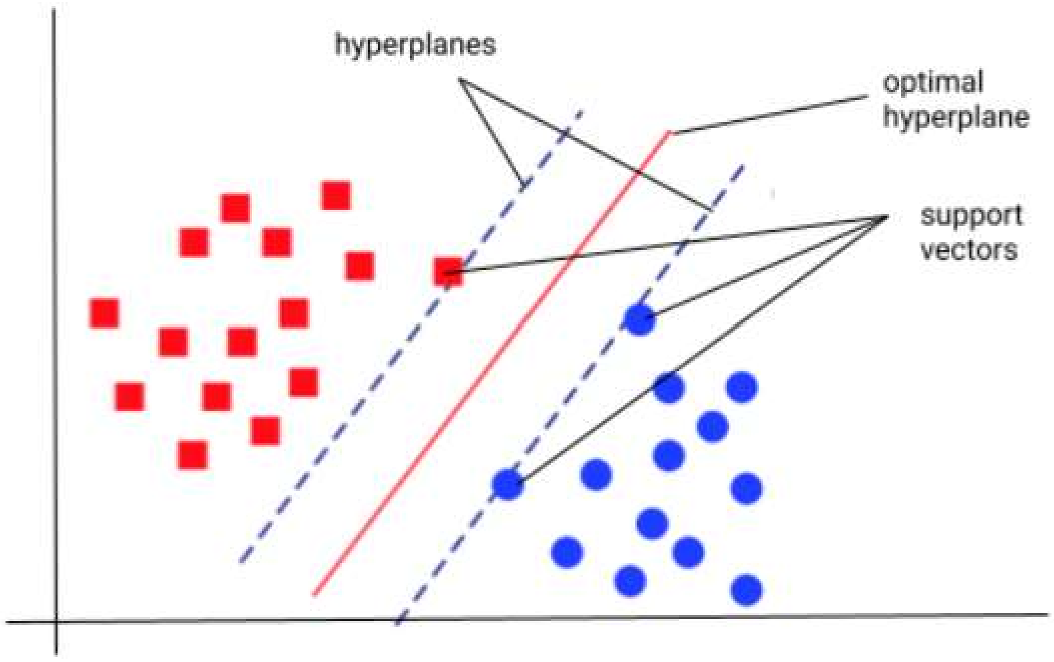
The softmax function is quite similar to the sigmoid explained earlier. It is used for multiclass classification because it can obtain the probabilities for various classes such that the probabilities of each class sum to 1. This means that an increase in the probability of a class causes a decrease in the probability of at least one of the other classes. It can also be referred to as a generalization of logistic regression or the sigmoid function and can be used for multi-class classification while the sigmoid function is used in multi-label classification. The softmax function is popularly used in the output layers of neural networks. Although the sum of the outputs of the softmax must be 1, this is not the same for the sigmoid function.

$$softmax(z_j) = \frac{e^{z_j}}{\sum_k e^{z_k}} \quad for\ j = 1, \ldots, k$$

Tree-Based Methods and The Support Vector Machine

Linear and non-linear Support Vector Machine

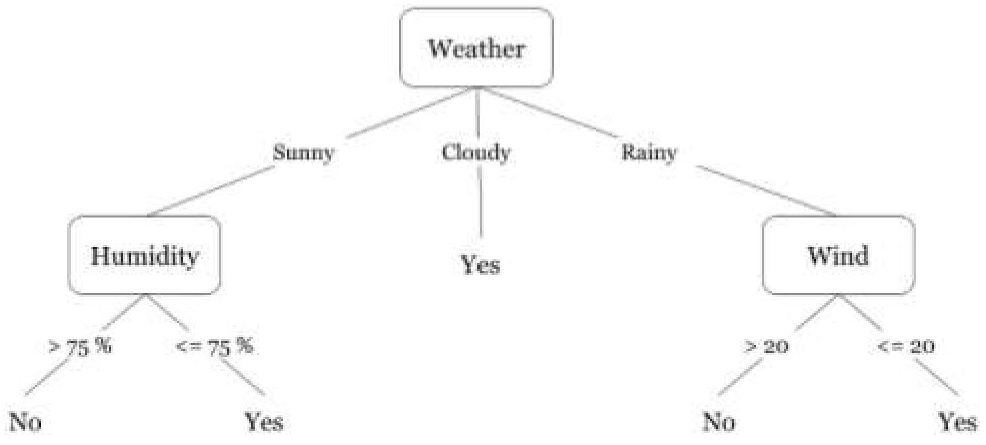
Support Vector Machine (SVM) is a supervised machine learning algorithm that is used to solve both classification and regression tasks. In classification, the algorithm uses a line or hyperplane to separate classes by using data points close to the boundary (support vector) for each class and a hyperplane that maximizes the distance between the classes. For clarity, a hyperplane is a line that linearly separates data points. Although there can be several hyperplanes between classes, the optimal hyperplane which has the maximum distance or margin between itself and the support vectors is chosen.



As we know, data is not always linearly separable such that a straight line might not be able to adequately segregate classes. Although SVM is a linear classifier, it can be used to classify a non-linear dataset by transforming the dataset to a higher dimensional feature space where it can be linearly separable. This is done using the kernel trick such that a kernel function is applied on each data point to map to a higher dimensional space.

Decision Trees and CART algorithm

The decision tree is a widely used non-parametric supervised machine learning approach that splits instances in a dataset based on different decision rules inferred from the features in the dataset. It is a tree-based algorithm with nodes that represent a specific attribute or decision rule such that for an instance, a question is asked at a node and possible answers to the question found on both edges. This is a sequential process that involves recursive partitioning of nodes for several features until the leaves for the tree provides the final output or class for that instance. Decision trees can also be used to solve regression problems.



ID3 - Iterative Dichotomiser 3, CART - Classification and Regression Trees and C4.5 are some examples of decision tree algorithms. In this section, we only discuss the CART algorithm. The CART predictive model generates decision rules that have a binary tree representation such that each non-terminal node has two child nodes as opposed to some other tree-based methods that have more child nodes. It supports numerical target variables. At every node, the best split is chosen such that the splitting criterion is maximised. Gini impurity index is used as the splitting criterion in CART.

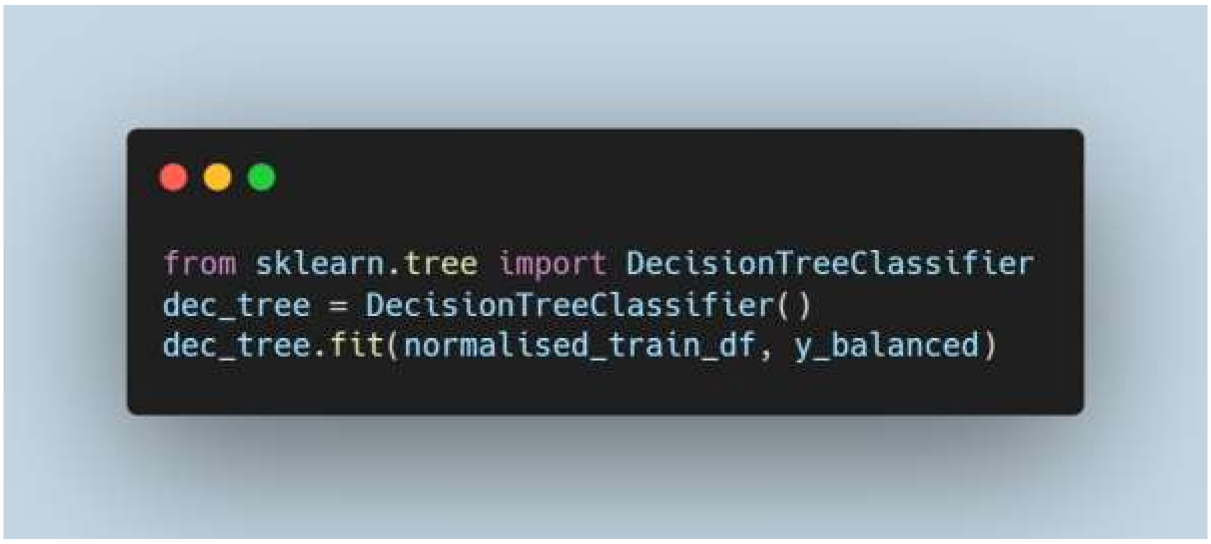
Gini Impurity: this is a measure of the chance that a randomly selected instance will be wrongly classified when selected. For different classes in a dataset, with p(i) as the probability that the chosen instance belongs to class i, the gini impurity index for all classes G, can be calculated such that:

$$G = 1 - \sum_{i} p(i)^2$$

Gini impurity index values range between 0 and 1 such that 0 translates to a pure classification where all instances belong to the same class while 1 means that there is a random distribution of the instances across different classes. To select the best split, the gini gain is calculated by taking a weighted sum of the gini impurity index then subtracting from the original impurity. Higher gini gain leads to better splits simply put, the lower the gini impurity, the better the split.

Overfitting in Decision Trees, Early Stopping and Pruning

The recursive partitioning of nodes until the final subsets are obtained in decision trees makes it prone to overfitting. The deeper the tree, the higher the chances of the overfitting. This can be prevented using a stopping criterion such as early stopping and pruning. Early stopping or pre-pruning involves stopping the tree-building process before the tree becomes too complex and the training data is perfectly classified. An early stopping condition like the maximum depth can be set to avoid deep trees such that the tree stops growing after reaching the set maximum depth for the tree. Another early stopping criterion that can be used is the classification error. At every splitting stage, the error is checked. If there is no significant decrease in the error, there is no need to make the tree more complex. When there are fewer data points than a set threshold value, early stopping can also take place. Early stopping may also produce underfit models if it stops too early. Post-pruning, on the other hand, allows the tree to be fully built before simplifying by removing sections of the tree at different levels by calculating the error rate.



```
from sklearn.tree import DecisionTreeClassifier
dec_tree = DecisionTreeClassifier()
dec_tree.fit(normalised_train_df, y_balanced)
```

Ensemble Methods

Beyond decision trees and ensemble classifiers

Ensembling in machine learning involves the combination of several classifiers to obtain an optimal model with better performance as opposed to just a single classifier. These classifiers can be of different algorithms and hyperparameters. Bagging, boosting, stacking and blending are methods classifiers can be combined.

Bagging

Bootstrap Aggregation or Bagging is a parallel ensembling technique that randomly bootstraps or samples the dataset with replacement to create subsets from the original. Multiple models are then trained using these subsets and the predicted results from these models aggregated to return final predictions. Bagging results in a final model that has less variance than its base classifiers.

- Bagging: Random Forests

When bagging is applied to decision trees, it results in random forests which is a supervised learning algorithm that has a large number of decision trees. For an instance in the dataset, each tree returns a prediction for the class the instance belongs to then, the class with the most votes becomes the final class for that instance. In random forests, it is assumed that a group of uncorrelated trees will do better than an individual tree. While some of the trees might be wrong in their predictions, many others will be correct.

Boosting: AdaBoost, Gradient Boosting and XGBoost

- Boosting

Boosting is a sequential process where every phase attempts to correct the errors made by the previous model. The main principle is to fit multiple weak learners which are slightly better than just random guessing. In contrast to bagging, boosting attempts to reduce both variance and bias. AdaBoost, Gradient Boosting and XGBoost are examples of boosting algorithms.

- AdaBoost

Adaptive Boosting is the first boosting algorithm. It is a very popular method for boosting that can be used on any classifier to present a more accurate model and improve its performance. It can be described with the following steps: create a subset from the entire dataset, assign equal weights to the data points, create a base model using this subset, predict using this model, calculate errors from the predicted results, assign higher weights to misclassified instances to increase their chances of being selected, create another model that tries to correct these mistakes and make new predictions then repeat until the maximum number of models specified are created. The final model is the weighted average of all the weak learners created. AdaBoost is very sensitive to noisy data and outliers so it is important to remove these when using AdaBoost.

- Gradient Boosting

This is another boosting algorithm that improves model performance where each model in the ensemble minimizes a loss function using gradient descent. The loss function which is used to obtain an estimate of how the model is performing, a weak learner - a model only slightly better than random guessing typically decision stumps (a decision tree with a single split - one level) and an additive model that combines the weak learners to make the final model are three important components in gradient boosting.

- XGBoost

Extreme Gradient Boosting is a supervised learning algorithm that implements gradient boosting by building trees parallely while applying regularization. It is well known for its scalability and fast execution. XGBoost can automatically identify missing values in data and it builds very deep trees before pruning for optimisation.

Additional Reading Resources

Additional Reading List & Their Respective Links:

Ensemble Learning: Bagging and Boosting:-

https://becominghuman.ai/ensemble-learning-bagging-and-boosting-d20f38be9b1e

Feature Engineering by Wale Akinfaderin.:-

https://www.youtube.com/watch?v=ZQ5wF7z01I0

Scikit-Learn Classification.:

https://stackabuse.com/overview-of-classification-methods-in-python-with-scikit-learn/

Gentle Introduction to XGBoost.:

https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

Learning from Imbalanced Class.:

https://www.jeremyjordan.me/imbalanced-data/

Hands-on Machine Learning - NUMBER ONE GUIDE:
https://www.lpsm.paris/pageperso/has/source/Hand-on-ML.pdf

Please read the following instructions carefully:

For the Graded Assessment, you are expected to make use of this dataset with link below:

http://bit.ly/HDSC-ML-Classiification-Dataset

You are to attempt all questions in the Graded Quiz within the stipulated time.

After completion of the assessment, you are required to submit the link to your code using this Google Form.

NB: Learn how to submit your Tag-Along project here: https://hamoyehq.medium.com/how-to-submit-your-tag-along-codes-8f210e53a83

Please note that plagiarism is highly prohibited and you will be disqualified if found guilty of it.

Instructions for Tag-Along Project

Stability of the Grid System

Electrical grids require a balance between electricity supply and demand in order to be stable. Conventional systems achieve this balance through demand-driven electricity production. For future grids with a high share of inflexible (i.e., renewable) energy sources, the concept of demand response is a promising solution. This implies changes in electricity consumption in relation to electricity price changes. In this work, we'll build a binary classification model to predict if a grid is stable or unstable using the UCI Electrical Grid Stability Simulated dataset.

Dataset: https://archive.ics.uci.edu/ml/datasets/Electrical+Grid+Stability+Simulated+Data+

It has 12 primary predictive features and two dependent variables.

Predictive features:

1. 'tau1' to 'tau4': the reaction time of each network participant, a real value within the range 0.5 to 10 ('tau1' corresponds to the supplier node, 'tau2' to 'tau4' to the consumer nodes);
2. 'p1' to 'p4': nominal power produced (positive) or consumed (negative) by each network participant, a real value within the range -2.0 to -0.5 for consumers ('p2' to 'p4'). As the total power consumed equals the total power generated, p1 (supplier node) = - (p2 + p3 + p4);
3. 'g1' to 'g4': price elasticity coefficient for each network participant, a real value within the range 0.05 to 1.00 ('g1' corresponds to the supplier node, 'g2' to 'g4' to the consumer nodes; 'g' stands for 'gamma');

Dependent variables:

1. 'stab': the maximum real part of the characteristic differential equation root (if positive, the system is linearly unstable; if negative, linearly stable);
2. 'stabf': a categorical (binary) label ('stable' or 'unstable').

Because of the direct relationship between 'stab' and 'stabf' ('stabf' = 'stable' if 'stab' <= 0, 'unstable' otherwise), 'stab' should be dropped and 'stabf' will remain as the sole dependent variable (binary classification).

Split the data into an 80-20 train-test split with a random state of “1”. Use the standard scaler to transform the train set (x_train, y_train) and the test set (x_test). Use scikit learn to train a random forest and extra trees classifier. And use xgboost and lightgbm to train an extreme boosting model and a light gradient boosting model. Use random_state = 1 for training all models and evaluate on the test set. Answer the following questions: