**Decision Tree**

*Reference: my understanding of decision trees has been supplemented and enriched by this set of videos and quizzes supplied by the excellent Analytics Vidhya resource for data science*. Link: <https://courses.analyticsvidhya.com/courses/getting-started-with-decision-trees>

The subsequent algorithm which will be applied to the song data and evaluated will be a Decision Tree classifier. As a Decision Tree is a relatively complex algorithm which accepts multiple hyperparameters (in contrast to the two classifiers above, especially Naive-Bayes which does not take any hyperparameters), the pre-made scikit-learn Decision Tree Classifier will be imported here. Although it would be an interesting challenge to implement a Decision Tree classifier from scratch, unfortunately due to the time constraints for the midterm assignment this is infeasible for the current project. Understanding which optimal value to select for splitting the group of samples based on a specific feature is something of a daunting task compared to the implementation of simpler algorithms like K-Nearest Neighbour or the Naive Bayes Classifier.

In order to select the optimal hyperparameters, the scikit-learn Grid Search facility will be used. The advantage of a Decision Tree classifier is that compared to the other classification algorithms here is that it lends itself to visualization of which features were the most important when classifying the samples into album categories.

**The Basic Concept of Decision Tree Classifiers**

A decision tree is a machine learning algorithm which can be used for both regression and classification tasks. As the objective of this particular project is to try to predict the album a song belongs to, the decision tree classifier rather than the regressor (used to predict continuous target variables) will be applied here. On the most basic level, a decision tree can be conceptualized as a series of yes-or-no questions, or if-else statements, each of which is represented as a decision ‘node’ in a binary tree. In the context of album-categorization, one of these ‘questions’ could be something like ‘is the value of the danceability feature of the song above 0.5?’ This ‘splits’ the original group of samples into two subsets of samples/songs, based on whether they meet this criterion or not. The ‘aim’ of each split can be thought of as trying to output two ‘pure’ or ‘homogenous’ groups, where each resulting subset contains only samples belonging to one class (album), thus perfectly separating the songs by album. A decision tree can have multiple splits – however, selecting the optimal stopping point at which there will be no more branching depends on various hyperparameters such as maximum tree depth which will be explored in more detail below.

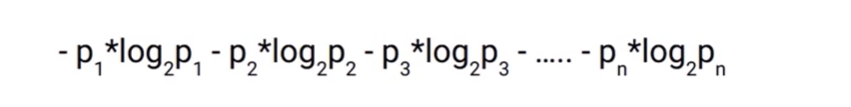
There are many options for which algorithm should be chosen for determining which of the splits should be the next one which is accepted and implemented in the tree. The most common of these is the ‘Gini’ split selection algorithm, which is the default criterion that is set in the scikit-learn Decision Tree Classifier class. The ‘Gini’ algorithm is based on the concept of ‘Gini impurity’, which is calculated with the formula:

* *Gini Impurity = 1 – Gini*, with *Gini* being the summation of *the squares of the probabilities* of each class in each of the sub-groups resulting from that particular ‘split’.
* The probability of each class in that sub-group is just the number of times a sample with that class appears in the sub-group, divided by the total number of samples in the subgroup.
* Once this *Gini* *Impurity* measure has been calculated for each of the two subgroups, it is multiplied by the sub-group’s weight, which is defined as the number of samples in that group as a proportion of the total number of the samples in the sub-group’s parent node.
* Gini Impurity is a measure of how ‘pure’ or ‘homogenous’ the samples in a sub-group/node are. As the aim of a decision tree is to output nodes/sub-groups which are as homogenous as possible, a lower Gini impurity is favored over a higher Gini impurity.
* Therefore, Gini Impurity is calculated for all possible splits, and the split with the lowest Gini Impurity is selected.

An alternative algorithm for selecting the optimal split is called *Chi-Squared*.

* The aim of chi-squared is to quantify the difference in homogeneity of classes between the parent node (original group of samples) and its child nodes (sub-groups based on the ‘split’).
* Overall, in this algorithm, the expected count of each class for the sub-groups is subtracted from the *actual* count of classes in that sub-group.
* This ‘expected’ frequency of samples belonging to a specific class is calculated as the proportion of those samples in the parent node (the one ‘making the decision’) multiplied by the total number of samples resulting from the split in the child node.
* The actual frequency is just that: the real count of the number of samples belonging to that specific class in the child node/sub-group.
* For each class existing in the group which is to be split, the difference between this actual count and expected count is calculated, then divided by the ‘expected’ count and squared.
* Then, the chi-squared value for that sub-group is defined as the sum of these squares for each class.
* As with Gini impurity, each child node’s chi-squared score is multiplied by its weight and these results are added together to form the chi-square value for that split.
* A high chi-squared value indicates that the difference in class distribution between the parent node’s group and the subgroups in the child nodes is high, and therefore that with this split, the tree has succeeded into taking a step towards creating purer and more homogenous groupings of samples with more samples in the groups having the same target variable.
* Therefore, the split resulting in the highest chi-squared value is the one that is selected.

There is also a third algorithm based on the concept of ‘information gain’ or ‘entropy’. ‘Entropy’ is another measure of how ‘pure’ a node is, with low entropy representing more homogenous nodes or sub-groups. The entropy of a resultant node is calculated with the following formula:



,where *n* is the number of possible classes, while *p* represents the proportion of that class in the node. This entropy value is calculated for both the parent node and as the weighted entropy of the two child nodes. If the entropy of the parent node is lower than the resulting weighted entropy of the two child nodes, this means that the split has not succeeded in making the sub-groups more homogenous, and that the split should be discarded. As such, the split resulting in the greatest reduction in entropy between the parent and weighted child nodes is selected.

Due to these ‘decisions’ splitting the group associated with the parent node into filtered sub-groups, a decision tree has the advantage of being easy to visualize, as one can see which features led to the selected splits and were therefore most useful for classifying the samples into target labels. I will make a brief aside that decision trees are my favourite supervised Machine Learning algorithm, as they are so intuitive and easy to imagine (although more difficult to implement than some of the other algorithms). Nevertheless, decision trees have important disadvantages that ust be discussed in order to justify the use of Grid Search to find the optimal configuration of parameters.

Technically, a decision tree could keep splitting and splitting until each ‘leaf’ or ‘terminal node’ (defined as a node without any further splits or children) would consist of completely ‘pure’ subsets of samples. Theoretically, one could imagine how for a set of training data, this splitting would continue until each leaf consisted of only one data point, which would trivially be ‘perfectly pure’. It is easy to imagine how this would lead to overfitting, where the model would basically have memorized the labels for every sample in the training data, but failed to grasp generalizable patterns, thus leading to poor performance on the test dataset. This is because the tree would simply have recorded the exact branching path down to the leaf nodes based on the specific, detailed characteristics of the training samples, which might not be applicable to new data. As such, various hyperparameters have to be tuned to stop this overfitting from occurring. Some of these include:

* Only splitting a parent node further if the parent node includes a subset of samples which reaches a certain (integer) threshold.
* Constraining the maximum depth (longest path from root node to a leaf node) of the decision tree, so that no more splits occur once this depth has been attained.
* Setting a minimum number of samples that a sub-group has to have in order to qualify to become a leaf node, and truncating the tree to the parent node if the potential leaf node does not meet this threshold.

Of course, while setting the minimum threshold value for the number of samples required for further splits to a very high number can prevent overfitting, but care has to be taken so that this value is not too low, in which case the decision tree will not learn sufficient patterns in the data and underfit. Similarly, with the maximum tree depth constrain, setting this to a number that is too low can result in an insufficiently complex model, and therefore to underfitting. These techniques used to prevent the risk of overfitting can be classed as ‘pre-pruning’ techniques (ref: <https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/#What_is_a_Decision_Tree>?) which refers to deleting nodes *while* growing the decision tree. However, sci-kit learn also enables input arguments determining the aggressiveness for ‘post-pruning’ of the tree, with parameters such as *ccp\_alpha*, or ‘cost-complexity pruning’ (ref: <https://scikit-learn.org/stable/modules/tree.html>), which identifies the ‘weakest’ leaf node (the ones with the greatest impurity) and removes them after the tree has been learnt. Using the PCA dimensionality reduction technique to reduce the number of features while retaining the most varied characteristics of the sample prior to constructing the Decision Tree can also prevent overfitting by giving the tree ‘a better chance of finding features which are discriminative’.

Consequently, we will use the scikit-learn Grid Search facility to search for and evaluate the optimal hyperparameters for these constraints before fitting the Decision Tree Classifier on the dataset and comparing its performance to that of the other supervised classifiers (ref: <https://plainenglish.io/blog/hyperparameter-tuning-of-decision-tree-classifier-using-gridsearchcv-2a6ebcaffeda#how-does-it-work>) .

**Random Forest Classifier**

Random Forests are a supervised learning algorithm that are a natural extension to Decision Trees. It is known as an ‘ensemble technique’ (ref: <https://www.analyticsvidhya.com/blog/2021/10/an-introduction-to-random-forest-algorithm-for-beginners/>), as it combines many individual Decision Tree models to improve the algorithm’s performance. The Random Forest classifier basically combines lots of decision trees and makes predictions based on a ‘majority vote’ of the predictions made by the individual decision trees. The class, in this case, the album name, which the majority of the individual decision trees output will thus constitute the final prediction. Random Forest algorithms have been renowned in the literature for being excellent at predicting musical genres (insert ref!). They are known for having several advantages over decision trees such as greatly reducing the risk of overfitting, reduce the need for otherwise cross-validating the dataset, and return accurate results even without having to tune too many hyperparameters. However, the theory behind Random Forest algorithms can initially be daunting and difficult to understand. I will try to give an overview of it here.

The reason that Random Forests reduce the problem of overfitting without relying on cross-validation is that they use a technique called *bootstrapping* during which subsets of the original dataset are formed, each of the same size as the original set, but with different combinations of samples (some samples might be duplicated, while others are left out completely from a subset). Then, each decision tree model inside the Random Forest is trained on a different one of these subsets. This reduces the risk of overfitting, as the training set is different for each separate decision tree, so new patterns can be recognized. Secondly, the performance of the models are actually sequentially improved after the completion of training each individual decision tree using a technique called *boosting*. Each tree tries to learn from the errors of the previous tree. Which features should be ‘split upon’ for the first ‘root’ node of each tree is decided by comparing the weighted Gini Impurity (see above in the explanation for decision trees) for splitting the first node on all the features in the dataset. Basically, a Random Forest just combines different decision trees and trains them on different same-sized combinations of the samples in the dataset, and then votes for the majority album name/class that most decision trees outputted. As well as evaluating the Random Forest classifier on a separate test set like the other algorithms, scikit-learn also provides a property of RandomForestClassifier called ‘oob\_score’, or ‘out-of-bag’ score which evaluates the model on the samples that were not included in the decision tree subsets to train the model. Although Random Forests are more robust than decision trees through this maximum utilization of data to reduce overfitting, they are slower and not as computationally efficient due to the requirement of training all the sub-trees.

As such, the final algorithm applied to the song data in this comparative analysis of the performance of different supervised learning techniques on album classification will be a Random Forest Classifier. This will hopefully result in optimized performance compared to the individual decision tree classifier. Similarly to the parameter optimization process used to obtain the optimal hyperparameters for the Decision Tree, the GridSearch class will be used again to tune some of the parameters, including the number of ‘estimators’ (individual decision trees) combined into this ensemble classifier.