Homework 2: Decision Tree

Nam Ho Phan

May 05, 2021

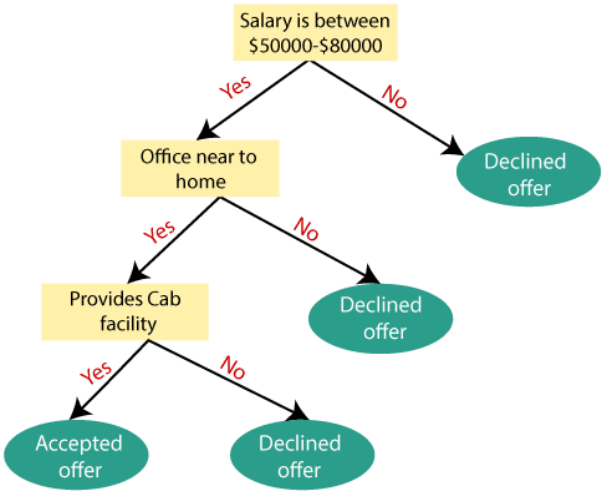
Class: ALY6020 - Predictive Analytics

Professor: Marco Montes de Oca

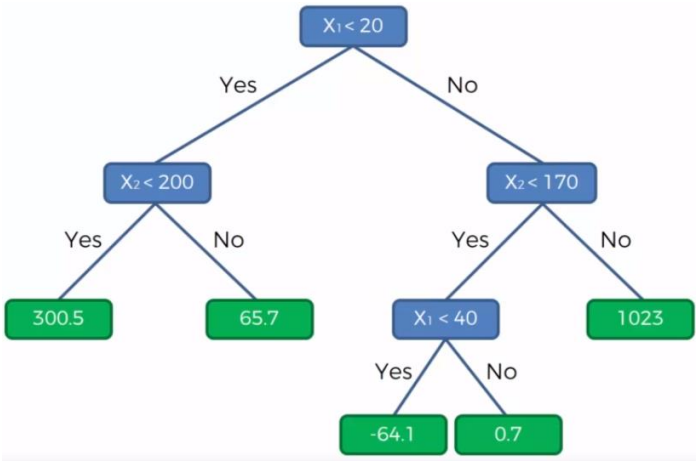
- **A brief introduction to the topic.**

Decision tree is the predictive model based on tree structure, which is suitable for fitting complex datasets. Decision tree can be used to predict the outcomes for both classification and regression; its algorithm is known as CART (Classification and Regression Tree).

For classification, decision tree consists of 3 components for prediction such as Nodes, Edges/Branch and Leaf Nodes. Nodes will prompt the main questions for certain features, Edges/Branch will lead the results of each certain feature to another Nodes for further insights until there is no more branch of the feature, and Leaf Nodes will have the function to decide the outcome of each Nodes (Y/N). To fit the decision tree classifier model, the data set should be formatted as categorical or discreate type. To predict the outcome, the model will go through the iterative process of splitting so the discreate type of data will help them to answer Yes/No properly.

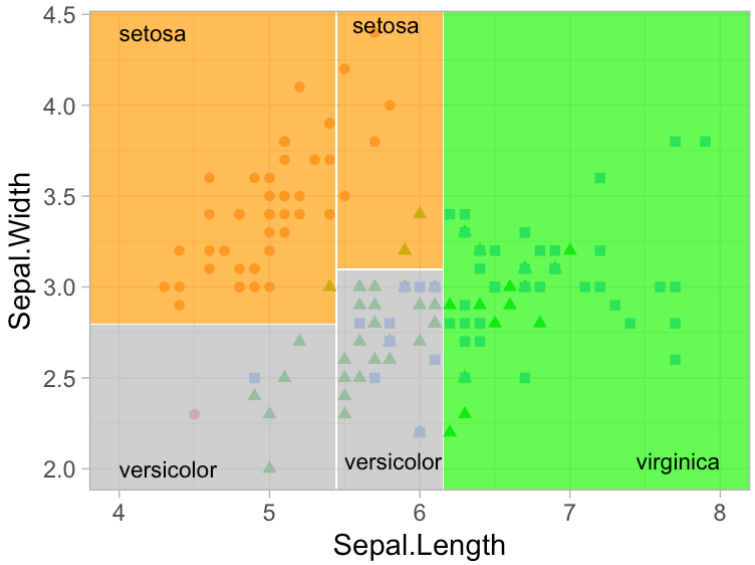


For regression, decision tree also consists of 3 components like classifier, but it will predict the outcome by breaking down the certain features into smaller subsets until it has enough attributes to conclude the decision. Therefore, the regression tree is appropriate for data set as continuous values.



One of the advantages for using decision tree is that we can avoid noise and many steps of data preparation due to the model's flexibility, ignore missing values, interpret the algorithm easily, reduce runtime and so save computational costs. Meanwhile, the overfitting is considered as the disadvantage of decision tree because this algorithm is quite greedy in splitting nodes.

In addition, decision tree is also determined based on clustering method, because it would separate the values with the borderline to decide the outcomes according to the common feature.



- **Code and output (or screenshots) with an explanation of what the code does.**

Install packages and load data set

Firstly, I will install all relevant packages for our necessity and load the data set from csv file. The data set should be adjusted to perform the records after July 2008 according to the requirement.

```
In [1]: import pandas as pd
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.preprocessing import OrdinalEncoder
        import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        from sklearn import preprocessing
        from sklearn import metrics
        from sklearn.metrics import confusion_matrix, f1_score, accuracy_score, classification_report
        from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
```

```
In [8]: #import data
        eshop = pd.read_csv('C://Users//nickh//OneDrive - Northeastern University//Notebooks//PredictiveAnalytics//e-shop data and descri
        eshop = eshop[eshop['month'] >=7]
        eshop
```

Out[8]:

	year	month	day	order	country	session ID	page 1 (main category)	page 2 (clothing model)	colour	location	model photography	price	price 2	page
116095	2008	7	1	1	9	17032	2	B4	10	2	1	52	1	1
116096	2008	7	1	2	9	17032	2	B7	6	3	1	38	2	1
116097	2008	7	1	1	29	17033	1	A11	3	4	1	62	1	1
116098	2008	7	1	2	29	17033	1	A11	3	4	1	62	1	1
116099	2008	7	1	1	29	17034	2	B30	9	4	1	57	1	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
165469	2008	8	13	1	29	24024	2	B10	2	4	1	67	1	1
165470	2008	8	13	1	9	24025	1	A11	3	4	1	62	1	1
165471	2008	8	13	1	34	24026	1	A2	3	1	1	43	2	1
165472	2008	8	13	2	34	24026	3	C2	12	1	1	43	1	1
165473	2008	8	13	3	34	24026	2	B2	3	1	2	57	1	1

49379 rows × 14 columns

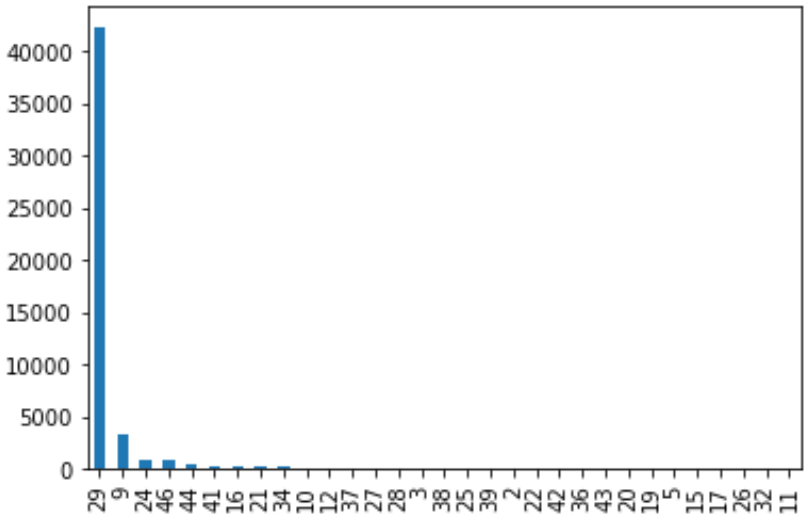
Preprocessing data

Before building predictive model for data set, we should convert categorical strings into numerical type so that it will be able to fit the classification model. For this step, I prefer to use ‘LabelEncoder’ function as it is quite convenient to convert data. After that, I will split the data into train set and test set, this will help me to validate the reliability of the model,

```
In [9]: #creating labelEncoder
        le = preprocessing.LabelEncoder()
        # Converting string labels into numbers.
        eshop['page 2 (clothing model)'] = le.fit_transform(eshop['page 2 (clothing model)'])

In [10]: y = eshop.country
         x = eshop.drop('country',axis=1)

In [11]: x_train,x_test,y_train,y_test =train_test_split(x,y,test_size=0.2)
```

Target counts

As the targets for model are classification, I will check the frequency of it to know how good the data set is for prediction. The graph below shows there is the imbalance in data set since the country 29 occupies most of the data set. Due to the imbalance data, the model will be less accurate because it just learns on the majority when skipping the minority.



Build decision tree models with random parameters

To understand the mechanism of decision tree classifier, I would try to run it with random parameters. Some of important hyperparameters would have an impact on the model performance:

Criterion: This parameter will allow us to select whether we want to use Gini Impurity or Entropy. Since it will not make much change to the trees, Gini impurity is often preferable because of its computational cost.

$$Gini: Gini(E) = 1 - \sum_{j=1}^{c} p_j^2$$
$$Entropy: H(E) = -\sum_{j=1}^{c} p_j \log p_j$$

Class_weight: This parameter will automatically add weights to the features if the data set is not balanced.

Max_depth: The max_depth represents for the nodes expanded based on the calculation of the impurity. The 'pure' (gini=0) means all classes are correctly labeled. Therefore, the depth of tree would depend on our purpose, the too low depth or too high depth will lead to underfitting or overfitting issue.

Min_samples_split: This is the minimum number of samples I require for each node. A high number of samples will help model learn the similar pattern effectively, but the too high number could lead to overfitting or lower the runtime.

Min_samples_leaf: This is the minimum number of samples I require for each leaf. The function of it is quite like min_samples_split as it helps model to control the overfitting.

Max_features: This will allow the maximum number of features for node splitting. The lower of features will help model be stable as it decreases the complexity of model.

Random_state: This is just a seed for random number generator. Any number would have same function.

Max_leaf_nodes: This is the maximum number of leaf nodes.

```
In [9]: tree_clf = DecisionTreeClassifier(criterion ='gini',class_weight='balanced',splitter = 'best',max_depth = 2,min_samples_split=5,
                                          min_samples_leaf=2,max_features='auto',random_state=10,max_leaf_nodes=2)
        tree_clf.fit(x_train,y_train)

Out[9]: DecisionTreeClassifier(class_weight='balanced', max_depth=2,
                               max_features='auto', max_leaf_nodes=2,
                               min_samples_leaf=2, min_samples_split=5,
                               random_state=10)

In [11]: y_predict = tree_clf.predict(x_test)

In [13]: print('accuracy:',accuracy_score(y_test,y_predict))
         print('f1_score:',f1_score(y_test,y_predict,average='macro'))
         print(classification_report(y_test, y_predict))

         accuracy: 0.00020251113811251962
         f1_score: 1.9661333529946666e-05
                       precision    recall  f1-score   support

                    2       0.00      0.00      0.00         4
                    3       0.00      0.00      0.00        11
                    9       0.00      0.00      0.00       651
                   10       0.00      0.00      0.00        14
                   11       0.00      0.00      0.00         0
                   12       0.00      0.00      0.00        17
                   15       0.00      0.00      0.00         1
                   16       0.00      0.00      0.00        50
                   19       0.00      0.00      0.00         1
                   21       0.00      0.00      0.00        43
                   22       0.00      0.00      0.00         6
                   24       0.00      0.00      0.00       187
                   25       0.00      0.00      0.00         7
                   27       0.00      0.00      0.00        11
                   28       0.00      0.00      0.00        11
                   29       0.00      0.00      0.00      8516
                   32       0.00      0.00      0.00         1
                   34       0.00      0.00      0.00        31
                   36       0.00      1.00      0.00         2
                   37       0.00      0.00      0.00        14
                   38       0.00      0.00      0.00         9
                   39       0.00      0.00      0.00         4
                   41       0.00      0.00      0.00        60
                   42       0.00      0.00      0.00         1
                   43       0.00      0.00      0.00         1
                   44       0.00      0.00      0.00        65
                   46       0.00      0.00      0.00       158

            accuracy                           0.00      9876
           macro avg       0.00      0.04      0.00      9876
        weighted avg       0.00      0.00      0.00      9876
```

After fitting and making prediction for test set, we will evaluate model based on the accuracy and f1_score. It is clearly that the imbalanced data set will not perform correctly if we equalize the weight of each target. To overcome this problem, I would prefer to oversample dataset before modelling it. I will show you how I tune hyperparameters with gridsearchCV.

Hyperparameter tuning with GridSearchCV

GridsearchCV is a useful tool for optimizing the model because it will try to combine different parameters to provide the most satisfied scores. Therefore, the tool is also beneficial for cross validation because it will be able to evaluate all the possible combination. With parameter ‘cv’ = 5, the model will train each model 5 times (five-fold cross validation). I will find the models having smallest RMSE to observe how the prediction fits the training data, so we can use this model to train on new instances with least errors.

In [198]:
```
#create a dictionary of all values we want to test
param_grid = {'max_depth': np.arange(2, 10),'min_samples_leaf': np.arange(2,10)
              ,'min_samples_split': np.arange(2,10)
             ,'min_samples_leaf':np.arange(2,10),'max_features':  ['auto','log2','None']}
# decision tree model
dtree_model=DecisionTreeClassifier(criterion= 'gini',class_weight='balanced', min_impurity_decrease = 0 , random_state = 1)
#use gridsearch to test all values
dtree_gs = GridSearchCV(dtree_model, param_grid, cv=5,scoring='neg_mean_squared_error',return_train_score=True)
#fit model to data
model_tree = dtree_gs.fit(x_train, y_train)
```

In [18]:
```
cvres = dtree_gs.cv_results_
score = []
for mean_score, params in zip(cvres['mean_test_score'],cvres['params']):
    if mean_score <0:
        print(round(np.sqrt(-(mean_score)),5),params)
        score.append(round(np.sqrt(-(mean_score)),5))
    else:
        next
```

```
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 2}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 3}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 4}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 5}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 6}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 7}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 8}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 2, 'min_samples_split': 9}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 2}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 3}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 4}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 5}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 6}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 7}
10.17558 {'max_depth': 2, 'max_features': 'auto', 'min_samples_leaf': 3, 'min_samples_split': 8}
```

Prediction with the best optimizer

The accuracy of model is increased after I tuned the hyperparameters although the accuracy is too small due to the imbalanced problem. However, it also tells us the importance of hyperparameter tuning, which helped me reduce computational cost and achieve good model quickly.

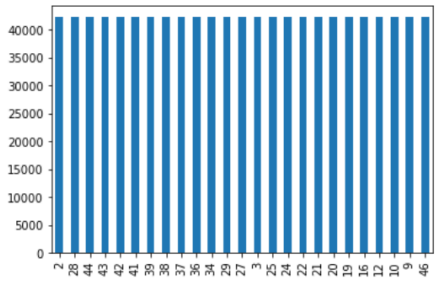
In [21]:
```
y_pred = model_tree.predict(x_test)
print('accuracy:',accuracy_score(y_test,y_pred))
print('f1_score:',f1_score(y_test,y_pred,average='weighted'))
print(classification_report(y_test, y_pred))
```

```
accuracy: 0.08404212231672742
f1_score: 0.1383109723415233
              precision    recall  f1-score   support

           2       0.02      1.00      0.04         4
           3       1.00      0.18      0.31        11
           9       0.29      0.05      0.08       651
          10       0.50      0.07      0.12        14
          11       0.00      0.00      0.00         0
          12       1.00      0.06      0.11        17
          15       0.00      0.00      0.00         1
          16       0.46      0.22      0.30        50
          19       0.00      0.00      0.00         1
          21       0.17      0.19      0.18        43
          22       0.00      1.00      0.00         6
          24       0.79      0.10      0.18       187
          25       0.02      0.86      0.04         7
          27       0.00      0.00      0.00        11
          28       0.02      0.64      0.03        11
          29       0.99      0.08      0.14      8516
          32       0.20      1.00      0.33         1
          34       0.23      0.74      0.35        31
          36       0.07      1.00      0.13         2
          37       0.00      1.00      0.01        14
          38       0.24      0.67      0.35         9
          39       0.17      1.00      0.30         4
          41       0.19      0.10      0.13        60
```

Oversample with multiple class

However, I have mentioned the imbalanced data will make the model become less reliable, so that model will not be able to apply into new instances. For example, the decision boundaries will just revolve around the majority targets, and this will cause bias on model because it cannot recognize the minority with just only 1 or 2 records. Obviously, with this limited information, model cannot learn well to perform in new data set in the future. Therefore, I will prefer to use SMOTE to oversample the classes. SMOTE is suitable for multiple classes and it uses KNN algorithm to predict classes. The SMOTE is just an option since I do not have good data set, because it requires further knowledge of use to avoid overfitting and the too low number of minorities (1,2) is not good idea for resampling.

```
In [47]: from imblearn.over_sampling import SMOTE
         oversample = SMOTE(sampling_strategy='auto',k_neighbors=3)
         X, y = oversample.fit_resample(x, y)

In [50]: y.value_counts().plot(kind='bar')
Out[50]: <AxesSubplot:>
```



Make prediction with the oversampled data set

The accuracy and f1_score are absolutely improved after I oversampled the dataset. The accuracy can be increased more if I use higher range of parameters, but my computer power is limited on the data set with more than 200 thousand records.

```
accuracy: 0.6296774193548387
f1_score: 0.6182638167340311
              precision    recall  f1-score   support

           2       0.97      0.91      0.94      8438
           3       1.00      0.84      0.91      8468
           9       0.19      0.29      0.23      8515
          10       0.85      0.48      0.62      8417
          12       0.30      0.90      0.45      8379
          16       0.57      0.55      0.56      8588
          19       1.00      0.93      0.97      8485
          20       1.00      1.00      1.00      8523
          21       0.92      0.22      0.35      8582
          22       0.47      0.81      0.59      8455
          24       0.54      0.46      0.49      8510
          25       0.45      1.00      0.62      8393
          27       0.36      0.63      0.46      8498
          28       0.89      0.80      0.84      8494
          29       0.91      0.07      0.14      8416
          34       1.00      0.68      0.81      8404
          36       1.00      0.82      0.90      8308
          37       0.60      0.96      0.74      8312
          38       0.57      0.67      0.62      8544
          39       0.99      1.00      0.99      8553
          41       0.88      0.12      0.22      8412
          42       1.00      0.57      0.72      8370
          43       1.00      0.73      0.84      8500
          44       0.39      0.19      0.25      8535
          46       0.45      0.12      0.19      8476

    accuracy                           0.63    211575
   macro avg       0.73      0.63      0.62    211575
weighted avg       0.73      0.63      0.62    211575
```

Feature Importance

The feature importance shows us how the decision tree is impacted by these features. The 81% of ‘session ID’ shows that the targets are more easily to be recognized mostly due to the difference of its ‘Session ID’.

In [71]:
```
## Calculating feature importance
feature_cols = list(x_train.columns)
feat_importance = model_tree.tree_.compute_feature_importances(normalize=True)
feat_imp_dict = dict(zip(feature_cols, model_tree.feature_importances_))
feat_imp = pd.DataFrame.from_dict(feat_imp_dict, orient='index')
feat_imp.rename(columns = {0:'FeatureImportance'}, inplace = True)
feat_imp.sort_values(by=['FeatureImportance'], ascending=False).head()
```

Out[71]:

	FeatureImportance
session ID	0.814438
day	0.125627
order	0.027711
page 2 (clothing model)	0.012932
page 1 (main category)	0.009773

Evaluate the model performance

‘Cross_val_score’ is very helpful to evaluate the performance of model on new data set. I think the gridsearchCV function can handle this but using ‘cross_val_score’ will ensure my

modeling quality. The Decision Tree has a score of around 12.65, with std +/- 0.009, which is quite good for implementing into the machine learning system.

```
In [108]: scores = cross_val_score(model_tree,x_train,y_train,scoring='neg_mean_squared_error',cv=5)
          tree_rmse_scores = np.sqrt(-scores)
```

```
In [109]: def display_score(scores):
              print('Scores:',scores)
              print('Mean:',scores.mean())
              print('Standard deviation:',scores.std())
```

```
In [110]: display_score(tree_rmse_scores)
```

```
Scores: [12.64590821 12.6383781  12.64658048 12.66733185 12.65345859]
Mean: 12.650331445368582
Standard deviation: 0.009749974834376481
```

- ***A critical analysis of the results***

Decision Tree is a very recommended algorithm for predicting model. Because the prediction targets are categorical type (different countries), I will decide to use decision tree classifier to predict correctly. The result of model accuracy is not reliable although it could provide 80 or 90%, because the data set is so imbalanced that the model is just be able to learn on classes having most data. Thus, this will lead to the failure if we implement into the new instances. Since the model learns on new data set balanced, the result will be more reliable, and we will know the minor targets to improve it for the next data set. After using SMOTE for balancing classes and GridSearchCV for tuning the hyperparameters, I can use the best hyperparameter for predicting on train set. With smallest RMSE as approximately 12.65, the model will perform better on new instances with 62% accuracy. Although this accuracy could be improved if I try higher range of parameters, the computational cost will be worser. With f1 as 61%, this is not too bad, and it is useful to check False negatives and False positives of model if necessary. Afterward, feature importance is needed after we already have the predictive model, sessionID is the most important feature with 81% of its impact on tree. The summary of feature importance targets will help us to reduce the complexity of features in the future. Additionally, the cross_val_score will be used to evaluate the quality of model on test set, the mean score and standard deviation will allow me to understand how good my predictive model. Thus, the mean score as 12.65 and standard deviation as 0.009, which is quite safe for model application.

Reference:

Mithrakumar. M (2019). How to tune a Decision Tree? Retrieved from https://towardsdatascience.com/how-to-tune-a-decision-tree-f03721801680

Chakure. A (2019). Decision Tree Classification. Retrieved from https://medium.com/swlh/decision-tree-classification-de64fc4d5aac

Gupta. P (2017). Decision Trees in Machine Learning. Retrieved from https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052