Decision Tree Classification with *sklearn*

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Goal: Define a decision tree to classify according to Gender from body measurements. The motivation is to forecast Gender body type when that information is missing from online clothing orders.

Preliminaries

Misc

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
    Analysis on 2021-07-30 at 21:36
```

```
import os
os.getcwd()
```

```
    '/content'
```

Import Standard Data Analysis Libraries

Only need `pandas` and `matlib` in this analysis, though does not hurt to do the standard import that includes `numpy` and `seaborn`.

```
import pandas as pd
import matplotlib.pyplot as plt
```

Get and Structure Data

```
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/BodyMeas.csv')
#d = pd.read_csv('data/BodyMeas.csv')
d.shape
```

```
    (340, 8)
```

```
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	F	200	71	43	46	45	8.5	7.5
1	F	155	66	31	43	37	8.0	8.0
2	F	145	64	35	40	40	7.5	7.5
3	F	140	66	31	40	36	8.0	9.0
4	M	230	76	40	43	44	9.0	12.0

Create the features and target data structures. The target variable, *Gender*, in this data set has two levels, F and M. These values need to be scored 0 and 1. Could use `get_dummies()` to obtain this scoring, but here manually create our dummy variable with the `pandas` function `replace()`. Arbitrarily score 1 for Male. The remaining seven variables are potential features.

```
classes = ['Female', 'Male']  # for graph
features = ['Weight', 'Height', 'Waist', 'Hips', 'Chest', 'Hand', 'Shoe']
X = d[features]
y = d['Gender'].replace({'F':0, 'M':1})
```

Access Solution Algorithm

Activate the decision tree analysis with the `sklearn` module `DecisionTreeClassifier`. In this example, instantiate the module as *dt_model*, referred to throughout the analysis. Set a maximum tree depth of five with the `max_depth` parameter.

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(max_depth=5)
```

Evaluate Model with *Multiple* Hold-Out Samples

k-fold cross-validation for *one* model

In this analysis we do not do a single train/test data split. Instead we go right to a 5-fold cross-validation to build and estimate the same model five different times according to the `n_splits` parameter.

The `Kfold` module specifies the re-ordering of the data to create the folds, partitions of the data set into five different training sets of data, each with a corresponding testing data set. Save the result in the variable here called *kf*.

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=5, shuffle=True, random_state=1)
```

The `cross_validate()` function performs the model analyses, performing a cross-validation on five different decision tree models on five different training sets. Each fold is constructed as specified by the *kf* variable output from the `Kfold` procedure, then the model is tested on the corresponding testing data set.

The maximum depth of the tree has been set at 5. Use all 7 features as defined in the X data frame.

Both training data set fit indices are requested in addition to those from the test data set. To do so, set parameter `return_train_score` to `True`. Assess the fit of each model with with the `scoring` parameter, requesting accuracy, recall, and precision.

```
from sklearn.model_selection import cross_validate
scores = cross_validate(dt_model, X, y, cv=kf,
                        scoring=('accuracy', 'recall', 'precision'),
                        return_train_score=True)
```

Convert the *scores* output from *cross_validate()* to a data frame for the appearance of the display. Transpose the data frame to view all the variable names with the data.

```
ds = pd.DataFrame(scores).round(3).transpose()
print(ds)
```

```
                       0      1      2      3      4
    fit_time         0.004  0.003  0.002  0.002  0.002
    score_time       0.005  0.003  0.003  0.003  0.003
    test_accuracy    0.882  0.926  0.941  0.882  0.882
    train_accuracy   0.974  0.971  0.971  0.978  0.971
    test_recall      0.917  0.921  0.933  0.861  0.867
    train_recall     0.978  0.985  0.964  0.985  0.993
    test_precision   0.868  0.946  0.933  0.912  0.867
    train_precision  0.970  0.956  0.978  0.971  0.952
```

We have the fit indices for each of the five cross-validations. Compute the mean across the five data sets of fit indices to obtain the best estimate of fit for each index.

```
print('Mean of test accuracy: %.3f' % ds.loc['test_accuracy'].mean())
print('Mean of test recall: %.3f' % ds.loc['test_recall'].mean())
print('Mean of test precision: %.3f' % ds.loc['test_precision'].mean())
```

```
    Mean of test accuracy: 0.903
    Mean of test recall: 0.900
    Mean of test precision: 0.905
```

The decision tree model with a depth of 5 and all 7 features fits well. Accuracy, recall (sensitivity), and precision for the testing data are around 90%.

As shown at the end of this notebook, we can apply the model, but we have not yet demonstrated the set of hierarchical decisions imposed by this decision tree.

Grid Search: Hyperparameter Tuning with Cross-Validation

Define any one decision tree model by a specific depth and number of features. The previous cross-validation was just for one model. But what is the best depth for the decision tree? The optimal number of features? To answer those questions, systematically explore a range of different models. Define many models, each with a different combination of depth and features, examples of what are called hyper-parameters. Explore the fit of a complete set of related models by systematically varying the depth and number of features with a hyper-parameter grid search.

Here define 4 levels of depth, 4 different numbers of features for 16 different models.

```
params = {'max_depth': [2, 3, 4, 5],
          'max_features': [1, 2, 3, 4]}
```

For each model, do cross-validations on 3 different folds. So 16x3=48 different analyses in all, conveniently and automatically accomplished with the module `GridSearchCV`, for grid-search cross-validation.

```
from sklearn.model_selection import GridSearchCV
kf3 = KFold(n_splits=3, shuffle=True, random_state=1)
```

`GridSearchCV` sets up the grid of the 16 models. Here, instantiate the module as `grid_search` and then use `fit()` to fit all the models, each 3 times. The `param_grid` parameter specifies the hyper-parameters to systematically adjust in all possible combinations. Transpose the results to fit on the page.

Much work is accomplished with little code, illustrating the power and convenience of `sklearn`.

```
grid_search = GridSearchCV(dt_model, param_grid=params, cv=kf3,
                           scoring=('accuracy', 'recall', 'precision'), refit=False,
                           return_train_score=True)
grid_search.fit(X,y)
```

```
    GridSearchCV(cv=KFold(n_splits=3, random_state=1, shuffle=True),
                 error_score=nan,
                 estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                                  criterion='gini', max_depth=5,
                                                  max_features=None,
                                                  max_leaf_nodes=None,
                                                  min_impurity_decrease=0.0,
                                                  min_impurity_split=None,
                                                  min_samples_leaf=1,
                                                  min_samples_split=2,
                                                  min_weight_fraction_leaf=0.0,
                                                  presort='deprecated',
                                                  random_state=None,
                                                  splitter='best'),
                 iid='deprecated', n_jobs=None,
                 param_grid={'max_depth': [2, 3, 4, 5],
                             'max_features': [1, 2, 3, 4]},
                 pre_dispatch='2*n_jobs', refit=False, return_train_score=True,
                 scoring=('accuracy', 'recall', 'precision'), verbose=0)
```

The default output of `GridSearchCV` is a list of all the parameter values implemented in the analyses. To access specific results, access the output data structure `cv_results`.

Here are all the results, for each individual fold and their summaries. The model that corresponds to each of the 16 columns is defined by the variable computed variable *params*.

```
d_results = pd.DataFrame(grid_search.cv_results_).round(2)
d_results.transpose()
```

	0	1	2	3	
mean_fit_time	0	0	0	0	
std_fit_time	0	0	0	0	
mean_score_time	0.01	0	0	0	
std_score_time	0	0	0	0	
param_max_depth	2	2	2	2	
param_max_features	1	2	3	4	
params	{'max_depth': 2, 'max_features': 1}	{'max_depth': 2, 'max_features': 2}	{'max_depth': 2, 'max_features': 3}	{'max_depth': 2, 'max_features': 4}	{'max_d 'max_feat
split0_test_accuracy	0.84	0.86	0.85	0.82	
split1_test_accuracy	0.83	0.88	0.91	0.92	
split2_test_accuracy	0.82	0.88	0.81	0.88	
mean_test_accuracy	0.83	0.87	0.86	0.88	
std_test_accuracy	0.01	0.01	0.04	0.04	
rank_test_accuracy	16	11	12	10	
split0_train_accuracy	0.82	0.87	0.93	0.88	
split1_train_accuracy	0.83	0.89	0.9	0.89	
split2_train_accuracy	0.79	0.9	0.89	0.91	
mean_train_accuracy	0.81	0.89	0.91	0.89	
std_train_accuracy	0.01	0.01	0.02	0.01	
split0_test_recall	0.83	0.79	0.9	0.81	
split1_test_recall	0.78	0.88	0.96	0.9	
split2_test_recall	0.86	0.82	0.7	0.93	
mean_test_recall	0.82	0.83	0.86	0.88	
std_test_recall	0.03	0.04	0.11	0.05	
rank_test_recall	15	14	11	6	
split0_train_recall	0.83	0.79	0.93	0.85	
split1_train_recall	0.8	0.86	0.94	0.85	
split2_train_recall	0.94	0.89	0.85	0.99	
mean_train_recall	0.86	0.85	0.91	0.9	
std_train_recall	0.06	0.04	0.04	0.07	
split0_test_precision	0.88	0.94	0.84	0.86	
split1_test_precision	0.83	0.86	0.86	0.92	
split2_test_precision	0.8	0.92	0.89	0.85	
mean_test_precision	0.84	0.91	0.86	0.88	
std_test_precision	0.03	0.03	0.02	0.03	
rank_test_precision	16	3	13	11	
split0_train_precision	0.79	0.92	0.93	0.89	
split1_train_precision	0.86	0.93	0.88	0.93	
split2_train_precision	0.73	0.9	0.92	0.85	
mean_train_precision	0.8	0.92	0.91	0.89	
std_train_precision	0.06	0.01	0.02	0.03	

In this output our primary concern is the `mean` of each fit index over the five folds. For example, `mean_test_accuracy` provides the mean of the five accuracy scores from the cross-validation. Subset the results data frame, and rename variables to obtain a more compact display.

```
d_summary = d_results[['param_max_depth', 'param_max_features', 'mean_test_accuracy',
                       'mean_test_recall', 'mean_test_precision', 'mean_train_accuracy',
                       'mean_train_recall', 'mean_train_precision']]
d_summary = d_summary.rename(columns= {
                       'param_max_depth': 'depth',
                       'param_max_features': 'features',
                       'mean_test_accuracy': 'test_accuracy',
                       'mean_test_recall': 'test_recall',
                       'mean_test_precision': 'test_precision',
                       'mean_train_accuracy': 'train_accuracy',
                       'mean_train_recall': 'train_recall',
                       'mean_train_precision': 'train_precision'})
d_summary
```

	depth	features	test_accuracy	test_recall	test_precision	train_accuracy
0	2	1	0.83	0.82	0.84	0.81
1	2	2	0.87	0.83	0.91	0.89
2	2	3	0.86	0.86	0.86	0.91
3	2	4	0.88	0.88	0.88	0.89
4	3	1	0.84	0.81	0.86	0.86
5	3	2	0.88	0.87	0.90	0.92
6	3	3	0.89	0.86	0.91	0.93
7	3	4	0.89	0.89	0.89	0.94
8	4	1	0.85	0.85	0.85	0.89
9	4	2	0.88	0.86	0.90	0.95
10	4	3	0.90	0.89	0.91	0.95
11	4	4	0.90	0.90	0.89	0.95
12	5	1	0.84	0.85	0.84	0.92
13	5	2	0.90	0.89	0.91	0.96
14	5	3	0.88	0.87	0.89	0.97
15	5	4	0.89	0.88	0.90	0.97

All the decision trees provide a similar fit, with very little overfitting except as expected in the more complex models.

Parsimony: Choose the simplest model that provides the best or almost the best fit to the testing data.

As always, strive for the most parsimonious model. A model with a depth of two and four features yields an accuracy of 0.88. Moving to a depth of four with two features increases accuracy only by 0.02, so perhaps not worth the additional model complexity for such a small increase in fit.

Choose Model and Estimate on All Data

Generally obtain the best estimates with the most data. Given sufficient fit from the validation phase, choose and then fit (estimate) the final model on the full data set. Sacrifice a little accuracy for parsimony, which results in a model with 3 features at a depth of 2.

```
dt_model = DecisionTreeClassifier(max_features=3, max_depth=2)
mf = dt_model.fit(X,y)
```

Use the created output variable `feature_importances_` to identify the most important features to choose the best 3, the three with non-zero entries. Instead of writing code individually for each feature, do a `for` loop to process each feature one-by-one.

```
for name, importance in zip(X.columns, dt_model.feature_importances_):
    print(name, '%.3f' % importance)
```

```
    Weight 0.000
    Height 0.149
    Waist 0.000
    Hips 0.000
    Chest 0.000
    Hand 0.851
    Shoe 0.000
```

The model accuracy, recall, and precision have previously been evaluated for forecasting efficiency, each as the mean of the corresponding k-fold values. Given the best form of the model, now compute the same values for the full data set.

Begin with the $\hat{y}_i$'s, computed with the `predict()` function.

```
y_fit = dt_model.predict(X)
```

```
from sklearn.metrics import confusion_matrix
pd.DataFrame(confusion_matrix(y, y_fit))
```

	0	1
0	158	12
1	16	154

From the obtained confusion matrix, calculate the basic test indices.

```
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
print ('Accuracy:  %.3f' % accuracy_score(y, y_fit))
print ('Recall:  %.3f' % recall_score(y, y_fit))
print ('Precision:  %.3f' % precision_score(y, y_fit))
print ('F1:  %.3f' % f1_score(y, y_fit))
```

```
    Accuracy:  0.918
    Recall:  0.906
    Precision:  0.928
    F1:  0.917
```

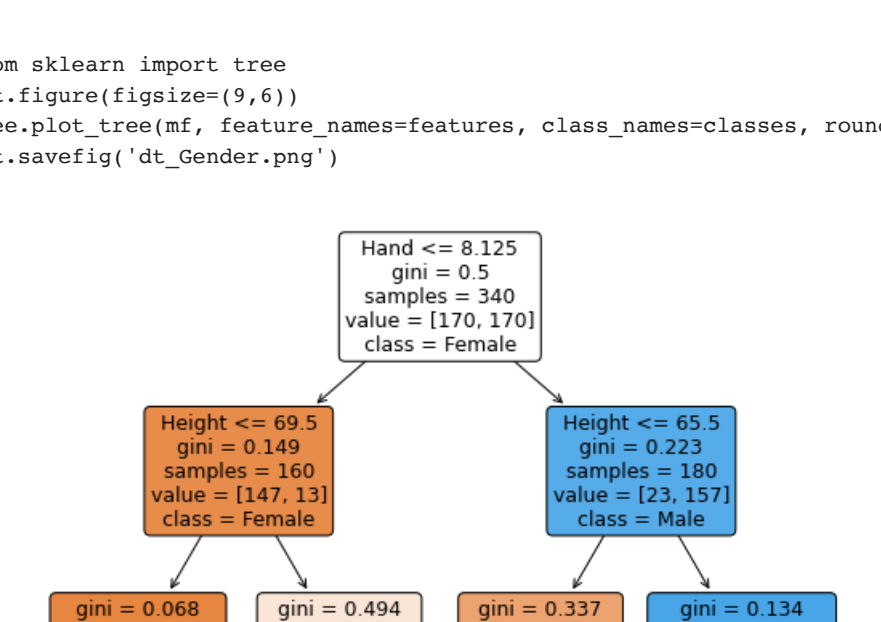
Illustrate the Model

The `sklearn` module `tree` provides a visualization of the obtained decision tree. The visualization shows the stakeholders who have sponsored the analysis (including your salary) how to classify customers with the model. The visualization facilitates model understanding, illustrating the sequence of decisions that arrive at the final classification.

The visualization also shows those classifications with which we have a high degree of confidence according to low Gini values and more intense coloring than those classifications for which we have little confidence, with a Gini coefficient closest to 0.5.

Obtain the visualization with the `plot_tree()` function from the `sklearn tree` module. Specify the features with the `feature_names` parameter, and the groups for which to classify into with the `class_names` parameter. Their respective values for this analysis, *features* and *classes*, have been previously defined much earlier in this notebook.

```
from sklearn import tree
plt.figure(figsize=(9,6))
tree.plot_tree(mf, feature_names=features, class_names=classes, rounded=True, filled=True)
plt.savefig('dt_Gender.png')
```



The `value` output indicates the number of samples classified as Female and Male, in that order. The more extreme the differences in the two numbers, the better the classification accuracy, which yields a more desirable lower *Gini coefficient*. For example, customers with a Hand size less than 8.125 inches, and a Weight less than 186 lbs, correctly classify 135 customers as Female and mis-classify only five Male customers as Female.

The total number of False Negatives, Males misclassified as Females, is 5 + 8 + 3 = 16, a number directly available from the confusion table. The tree diagram specifies exactly where those misclassifications occurred.

There are 12 False Positives, as indicated from the tree diagram or the confusion matrix.

Apply the Model

The purpose of building machine learning predictive models is to predict a target data value, including classifications, from new data with an unknown target value.

Jupyter notebooks are not an optimal production environment where new data is entered into the previously developed model to obtain a forecast. However, the following code does generate a model forecast from new data, such as in an actual forecasting application. The double brackets for the array of new data from which to make a prediction indicate the creation of a two-dimensional array, which is the form of input for `sklearn` functions. To apply the model more efficiently, we could add a read statement that reads X_new values from an external data file and then calculate a sequence of probabilities and predictions.

This model only uses three features, so it could refit on just a feature data set X that contains only those three features: Weight, Height, and Hand size. Then we would only enter just those three values to obtain a forecast of Gender.

```
X_new = [[142,66,31,40,36,8,9]]
y_new = dt_model.predict(X_new)
print("Predicted group membership:", y_new)
y_prob = dt_model.predict_proba(X_new)
print(round(y_prob[0,1], 3))
```

```
    Predicted group membership: [0]
    0.035
```