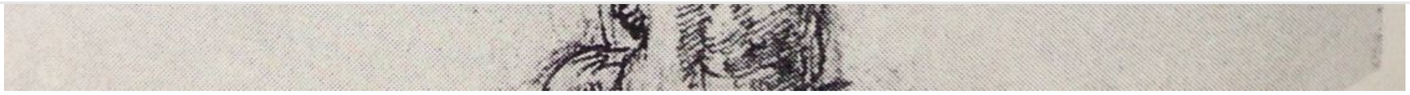
Joann

3 Followers About Follow

Classification Tree Growing and Pruning with Python Code (Grid Search & Cost-Complexity Function)

Joann Mar 25 · 7 min read

182


Cesare da Sesto (1480–1521), Study for a Tree

Topics in this article:

1. ***Recursive Partitioning***

2. *Impurity Measurements:* ***Gini Index*** *and* ***Entropy***

3. *Tree evaluation using* ***Grid Search*** *and* ***Cost-Complexity*** *with Cross Validation (CV)*

4. ***Pros and Cons*** *about Decision Tree*

Why Decision Tree?

Among the numerous data mining methods, decision tree is a flexible algorithm that could fit both regression and classification problems. Moreover, as a prediction-oriented algorithm, decision tree is also easy to interpret under transparent rules based on the tree splits, making the predictive results readily understandable.

Basic Concepts & Property in Tree

1. Decision nodes: nodes that have following branches and nodes downward the tree.

2. Terminal nodes/leaves: nodes that have no successors.

3. For binary trees: the number of terminal nodes is exactly one more than the number of decision nodes.

Response Variable: binary (0/1 or No/Yes), meaning a classification problem.

First, a variable is selected, say X1, then a value of it, s1, is chosen to split the input variables (a 4-dimensional space) into two non-overlapping parts, points with X1 < s1, and points with X1 => s1. One of these two parts is then selected and a variable and a value is used to split it in a similar manner. This process repeats again and again until there are many small partitions. Each partition is operated on the result of its previous partition, therefore it's called recursive partitioning. Importantly, the goal of the partitions is to **have each partition as pure as possible**. By pure, it means that the partition contains records that belong to only one class (YES/NO). The measurements of impurity are **Gini Index and Entropy**.

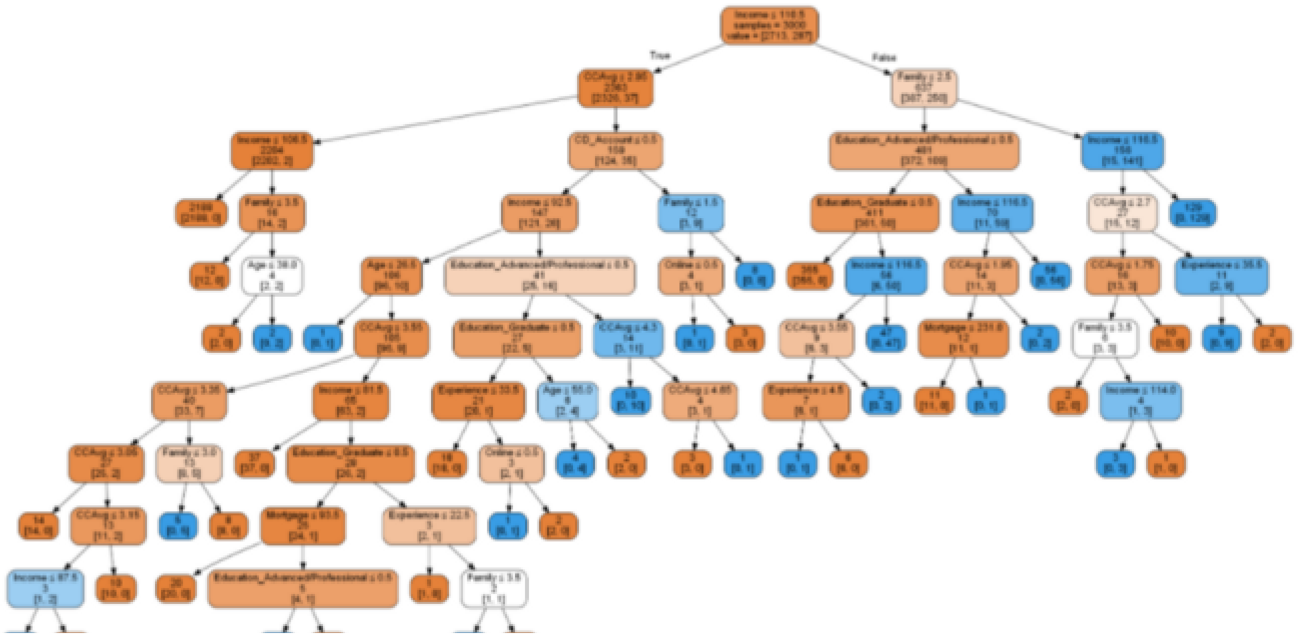
How s1 (the splitting point) is selected? For each variable, we sort the values and take the midpoints of each pair of consecutive values. Among the possible splitting points, we rank them according to how much impurity they have reduced.

How to define Reduction of impurity? It's the difference between the overall impurity before the split and the sum of impurities of the two partition after the split.

```
# load the data
bank_df = pd.read_csv('UniversalBank.csv')

# data preprocess
bank_df.columns = [c.replace(' ', '_') for c in bank_df.columns]
```

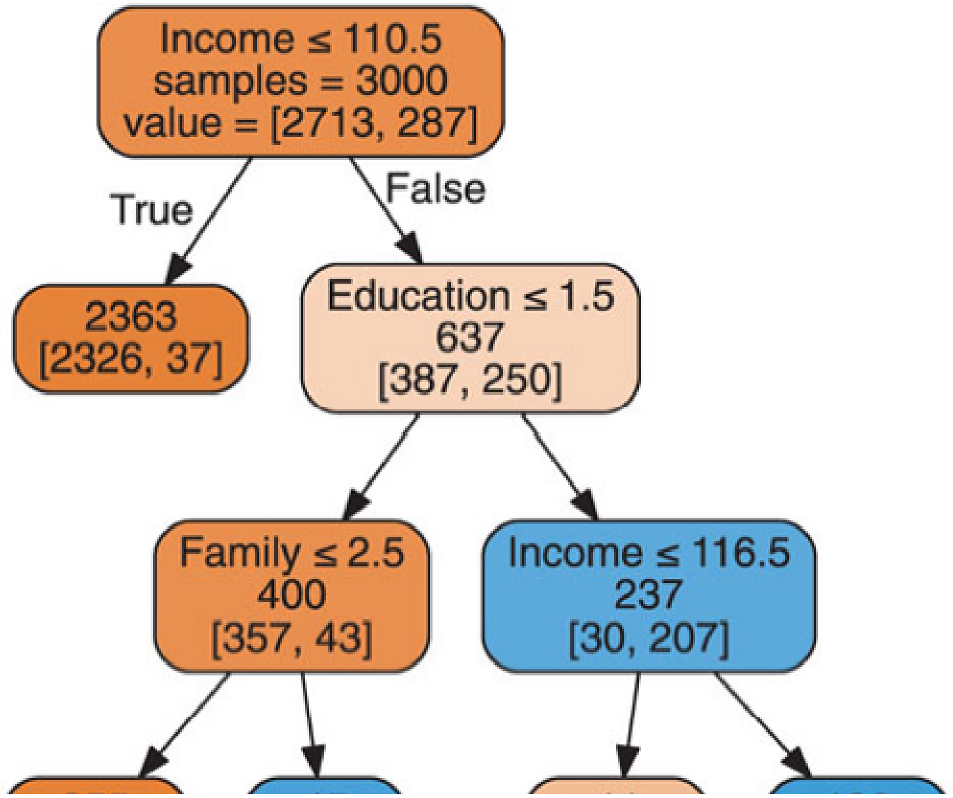
```
 'Advanced/Professional'}
bank_df.Education.cat.rename_categories(new_categories, inplace=True)
bank_df = pd.get_dummies(bank_df, prefix_sep='_', drop_first=True)

# split the train and validation data
x = bank_df.drop(['Personal_Loan', 'ZIP_Code', 'ID'], axis='columns')
y = bank_df['Personal_Loan']
train_x, val_x, train_y, val_y = train_test_split(x, y, test_size =
.4, random_state = 1)

# fit the tree
fulltree = DecisionTreeClassifier(random_state=2)
fulltree.fit(train_x, train_y)

# plot the tree
plotDecisionTree(fulltree, feature_names=train_x.columns)
```

Out[5]:



Tree Evaluation: Grid Search and Cost Complexity Function with out-of-sample data

Why evaluate a tree? The first reason is that **tree structure is unstable**, this is further discussed in the pro and cons later. Moreover, **a tree can be easily OVERFITTING**, which means a tree (probably a very large tree or even a fully grown tree) focus too much on the data and capture relationships and noise that are invalid when predict new data, leading to poor predictive power we won't desire.



Sample: A smaller tree

There are two major ways to achieve a smaller gap in performance between training set and testing set (new dataset). Firstly, we can stop the tree growth before it's overfitting, ending up with a tree with less terminal nodes and splits. The other way is to prune the fully grown tree back to a smaller one with a cost-complexity function in the algorithm. Details and Python code are as follows.

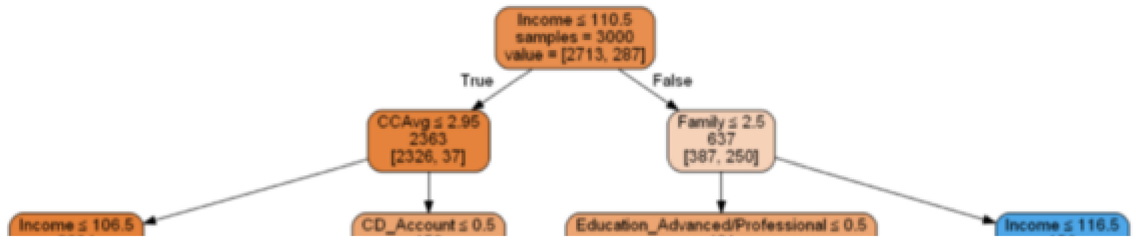
Grid Search: fine-tuning tree parameters

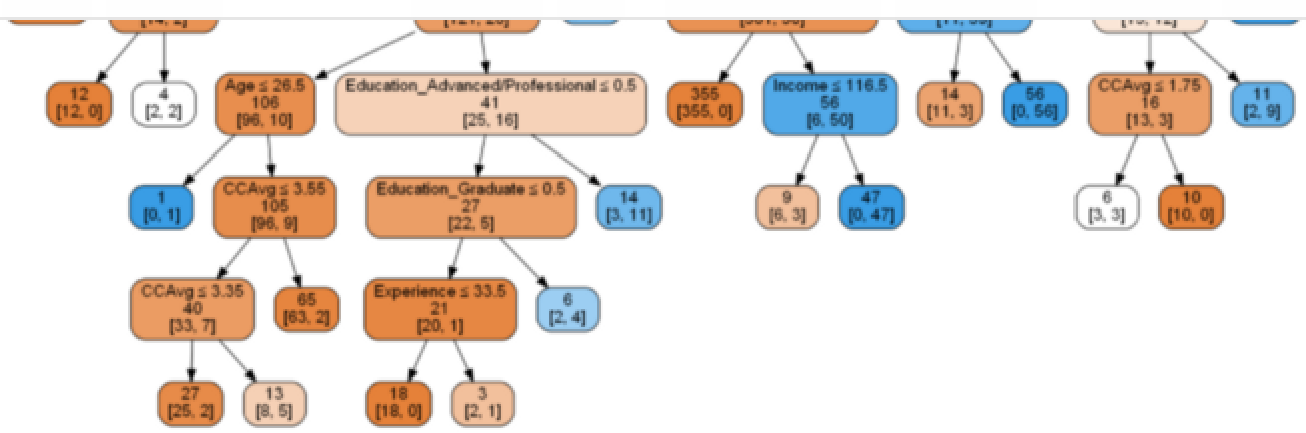
It's not easy to determine to what depth or number of splits we can develop the tree so that the predictive power can boost. One way to solve this is to try multiple different combinations of them, which is called **grid search**. There are a list of parameters in the DecisionTreeClassifier() from sklearn. The frequently used ones are **max_depth, min_samples_split, and min_impurity_decrease** (click here to check out more explanations).

```
# using GridSearchCV to fine tune method parameters
# Start with an initial guess for parameters
param_grid = {
 'max_depth':[5,10,20,],
 'min_samples_split':[20,40,60,80],
 'min_impurity_decrease': [0.0001, 0.0005, 0.001, 0.005, 0.01]
}

gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1),
param_grid, cv=5, n_jobs=1)
```

```
print('Initial parameters: ', gridSearch.best_params_)

#-----------------------------------------------------------------

# Adapt grid based on result from initial grid search
param_grid = {
 'max_depth': list(range(3,12)),
 'min_samples_split': list(range(15, 35)),
 'min_impurity_decrease': [x/10000 for x in range(1, 900, 5)],
}
gridSearch = GridSearchCV(DecisionTreeClassifier(random_state=1),
param_grid, cv=5, n_jobs=-1)
gridSearch.fit(train_x, train_y)

print('Improved score: ', gridSearch.best_score_)
print('Improved parameters: ', gridSearch.best_params_)
bestClassTree = gridSearch.best_estimator_

# fit the final model with best parameters: {'max_depth': 7,
'min_impurity_decrease': 0.0001, 'min_samples_split': 15}

prunedtree = DecisionTreeClassifier(random_state = 1, max_depth=7,
min_impurity_decrease=0.0001, min_samples_split=15)
prunedtree.fit(train_x, train_y)

plotDecisionTree(prunedtree, feature_names=train_x.columns)
```

Out[8]:





The smaller tree after parameter-tuning with grid search

Is there a way to determine the range of parameters so that the optimal choice could be covered with the best probability? (Answered by Professor Noah Giansiracusa)

> *Practically, decision tree is one of the algorithms that can be trained quickly, therefore it's fine to start with a broad parameter range and a fairly large step size and conduct grid search. Then we can zoom in to a sub-range where we think the better values are located and perform another grid search with a smaller step size.*
>
> *Having said that, there are a few technical methods commonly used in the context of deep learning, such as random walk type approaches and some Bayesian ones, for better search of hyperparameter values (a useful article here).*

Cost-Complexity Function with Cross Validation

> *More specifically, the CART algorithm uses a cost-complexity function that balances tree size (complexity) and misclassification error (cost) in order to choose tree size.*

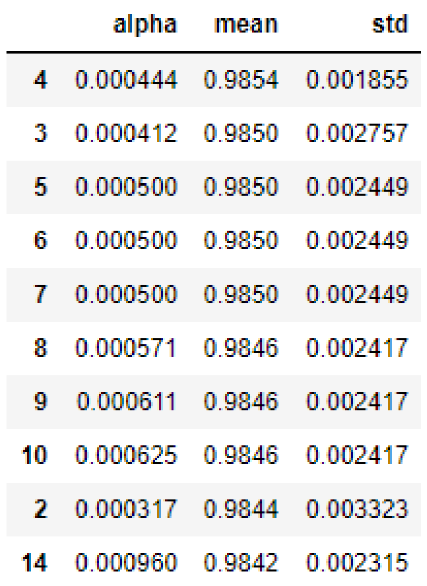
CC(T) = err(T) + αL(T)

α is the cost-complexity parameter in the penalty term (tree size). When α = 0, the tree grows fully and overfit the data, when α =1, the tree is just a single node and hence underfit the data. The process is that starting from α = 0, we slowly increase it to 1 to fit trees at each stage. For each tree and its corresponding α, we use cross-validation to evaluate the tree, then compute the average errors across all different splits. We choose the α complexity parameter for which the CV error is minimal. And use the α and all the data we have grow a new tree.

There is no such a shortcut function currently in Python to implement the above process. So I applied a loop function to incorporate the necessary steps.

```
# prune the tree with cost complexity pruning — Alpha
path = fulltree.cost_complexity_pruning_path(train_x, train_y)
alphas, impurities = path.ccp_alphas, path.impurities

mean, std = [], []
for i in alphas:
 tree = DecisionTreeClassifier(ccp_alpha=i, random_state=0)
# 5 fold cross validation for each alpha value
 scores = cross_val_score(tree, x, y, cv=5)
 mean.append(scores.mean())
```

```
eva_df = pd.DataFrame({'alpha': alphas, 'mean': mean, 'std': std})
eva_df = eva_df.sort_values(['mean'], ascending = False)
eva_df.head(10)
```



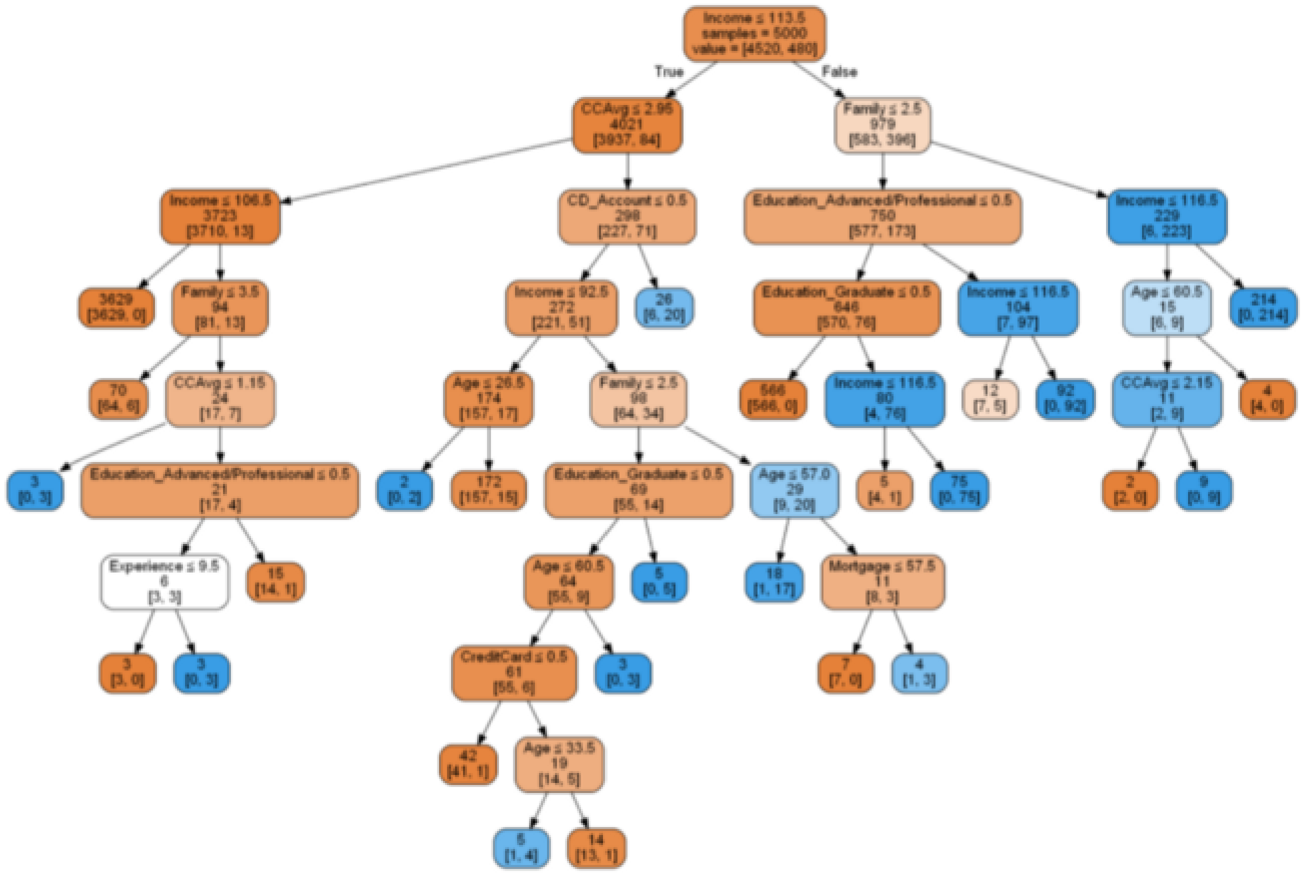
	alpha	mean	std
4	0.000444	0.9854	0.001855
3	0.000412	0.9850	0.002757
5	0.000500	0.9850	0.002449
6	0.000500	0.9850	0.002449
7	0.000500	0.9850	0.002449
8	0.000571	0.9846	0.002417
9	0.000611	0.9846	0.002417
10	0.000625	0.9846	0.002417
2	0.000317	0.9844	0.003323
14	0.000960	0.9842	0.002315

Partial evaluation scores after CV

It looks like the model with the best accuracy performance has α = 0.000444. It also has a lower standard deviation, meaning the accuracy scores are less variant, or more stable than that from models under other α values.

```
# fit the pruned tree - cv
prunedtree_cv = DecisionTreeClassifier(ccp_alpha=eva_df.iloc[4, 0],
classificationSummary(train_y, prunedtree_cv.predict(train_x))
classificationSummary(val_y, prunedtree_cv.predict(val_x))
```

Out[32]:



Income ≤ 113.5 samples = 5000 value = [4520, 480]
True
False
CCAvg ≤ 2.95 4021 [3937, 84]
Family ≤ 2.5 979 [583, 396]
Income ≤ 106.5 3723 [3710, 13]
CD_Account ≤ 0.5 298 [227, 71]
Education_Advanced/Professional ≤ 0.5 750 [577, 173]
Income ≤ 116.5 229 [6, 223]
3629 [3629, 0]
Family ≤ 3.5 94 [81, 13]
Income ≤ 92.5 272 [221, 51]
26 [6, 20]
Education_Graduate ≤ 0.5 646 [570, 76]
Income ≤ 116.5 104 [7, 97]
Age ≤ 60.5 15 [6, 9]
214 [0, 214]
70 [64, 6]
CCAvg ≤ 1.15 24 [17, 7]
Age ≤ 26.5 174 [157, 17]
Family ≤ 2.5 98 [64, 34]
566 [566, 0]
Income ≤ 116.5 80 [4, 76]
12 [7, 5]
92 [0, 92]
CCAvg ≤ 2.15 11 [2, 9]
4 [4, 0]
3 [0, 3]
Education_Advanced/Professional ≤ 0.5 21 [17, 4]
2 [0, 2]
172 [157, 15]
Education_Graduate ≤ 0.5 69 [55, 14]
Age ≤ 57.0 29 [9, 20]
5 [4, 1]
75 [0, 75]
2 [2, 0]
9 [0, 9]
Experience ≤ 9.5 6 [3, 3]
15 [14, 1]
Age ≤ 60.5 64 [55, 9]
5 [0, 5]
18 [1, 17]
Mortgage ≤ 57.5 11 [8, 3]
3 [3, 0]
3 [0, 3]
CreditCard ≤ 0.5 61 [55, 6]
3 [0, 3]
7 [7, 0]
4 [1, 3]
42 [41, 1]
Age ≤ 33.5 19 [14, 5]
5 [1, 4]
14 [13, 1]

The final tree after pruning with cost-complexity function

Pros and Cons about Decision Tree

Pros:

> *Variable subset selection is automatic since it is part of the split selection.*
>
> *Trees are also intrinsically robust to outliers, since the choice of a split depends on the ordering of values and not on the absolute magnitudes of these values.*
>
> *Without having to impute values or delete records with missing values.*
>
> *Transparent rules that they generate.*

Cons:

> *Sensitive to changes in the data, and even a slight change can cause very different splits!*
>
> *Computational expensive, since it requires a large dataset;*
>
> *In "favor" predictors with many potential split points. This includes categorical predictors with many categories and numerical predictors with many different values. Such predictors have a higher chance of appearing in a tree.*
>
> *Nonlinear and nonparametric, not assume any relationship between predictors and the outcome. Classification trees are useful classifiers in cases where horizontal and vertical splitting of the predictor space adequately divides the classes. But consider, for instance, a dataset with two predictors and two classes, where separation between the two classes is most obviously achieved by using a diagonal line.*

To find everything about the code in this article, refer to my Github here.

learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html
https://www.analyticsvidhya.com/blog/2020/10/cost-complexity-pruning-decision-trees/
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
https://scikit-learn.org/stable/modules/cross_validation.html
https://www.dataminingbook.com/book/python-edition

My LinkedIn: https://www.linkedin.com/in/joannzhang1818/

Thanks for reading!

Classification Tree Data Mining Grid Search K Fold Cross Validation Decision Tree