Predicting employee Attrition using decision tree method

By Jeffrey Mitini Nkhoma

September 25, 2022

Applying decision trees methods

A decision tree is a very flexible and popular method of classifying objects. It is a form of supervised machine learning where we continuously split data according to a certain parameter (Chakure, 2022). There are two main types of decision trees: Classification Trees and Regression Trees. In this project, I will be applying the classification tree to classify data in an employee attrition dataset.

Dataset

In this project I create a model that predicts employee attrition using the famous IBM HR Analytics Employee Attrition & Performance dataset obtained from Kaggle (Pavansubhash, 2017) It is a fictional dataset that was created by IBM data scientists. The dataset has personal, benefits and HR information for employees in thirty-five variables and 1470 records.

I apply decision tree to map out a path that leads to attrition by using sixteen variables out of twenty-six available in the dataset using python, specifically Scikit-Learn library. From scikit-Learn, I imported decisiontreeclassifier, train_test_split and metric modules. The resulting model should be able to predict attrition or classify employees according to those that are likely to leave and those likely to stay, based on historical data. **Figure 1** below shows descriptive analytics for the dataset.

Figure 1

Employee Attrition Descriptive statistics (First nine columns)

```
attrition.describe()
```

	Age	DailyRate	DistanceFromHome	Education	EmployeeCount	EmployeeNumber	EnvironmentSatisfaction	HourlyRate	JobInvolvement
count	1470.000000	1470.000000	1470.000000	1470.000000	1470.0	1470.000000	1470.000000	1470.000000	1470.000000
mean	36.923810	802.485714	9.192517	2.912925	1.0	1024.865306	2.721769	65.891156	2.729932
std	9.135373	403.509100	8.106864	1.024165	0.0	602.024335	1.093082	20.329428	0.711561
min	18.000000	102.000000	1.000000	1.000000	1.0	1.000000	1.000000	30.000000	1.000000
25%	30.000000	465.000000	2.000000	2.000000	1.0	491.250000	2.000000	48.000000	2.000000
50%	36.000000	802.000000	7.000000	3.000000	1.0	1020.500000	3.000000	66.000000	3.000000
75%	43.000000	1157.000000	14.000000	4.000000	1.0	1555.750000	4.000000	83.750000	3.000000
max	60.000000	1499.000000	29.000000	5.000000	1.0	2068.000000	4.000000	100.000000	4.000000

8 rows × 26 columns

Figure 2

The first five records and ten attributes of the dataset

```
attrition.head()
```

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EmployeeCount	EmployeeNumber
0	41	1	Travel_Rarely	1102	Sales	1	2	Life Sciences	1	1
1	49	2	Travel_Frequently	279	Research & Development	8	1	Life Sciences	1	2
2	37	1	Travel_Rarely	1373	Research & Development	2	2	Other	1	4
3	33	2	Travel_Frequently	1392	Research & Development	3	4	Life Sciences	1	5
4	27	2	Travel_Rarely	591	Research & Development	2	1	Medical	1	7

5 rows × 35 columns

The first task was to change the values in the 'Attrition' field, which is the target field, to have a zero in place of a 'No,' and one in place of a 'Yes,'

Secondly, the dataset was split to select sixteen variables as feature variables, and the 'Attrition' variable as the target variable. The dataset was also split into train and test data forming 80 and 20 percent, respectively. **Figure 3** below shows the two processes of selecting feature and target variables, and then test and train data.

Figure 3

Creating train and test data, feature, and target variables

```
#split dataset in features and target variable
feature_cols = ['DistanceFromHome', 'Education', 'EnvironmentSatisfaction', 'HourlyRate',  'RelationshipSatisfaction',
                'WorkLifeBalance', 'YearsAtCompany','Age', 'DailyRate', 'StandardHours','TotalWorkingYears',
                'TrainingTimesLastYear','YearsInCurrentRole', 'YearsSinceLastPromotion', 'YearsWithCurrManager',
                'StockOptionLevel']
X = attrition[feature_cols] # Features
y = attrition['Attrition'] # Target variable

# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1) # 80% training and 20% test
```

Training the model

Using the Decision Tree Classifier module, the decision tree object was created fitting in the X_train data (features) and Y_train (target). **Figure 4** shows the creation of the model, and code that predicts using the training and testing data.

Figure 4

Creating and testing model

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)


#Predict the response for train dataset
y_train_pred = clf.predict(X_train)


#Predict the response for test dataset
y_test_pred = clf.predict(X_test)
```

Obviously, using the training data should have accuracy of 100%, and testing data yielded 71%. See **figure 5** below:

Figure 5

Model Accuracy

```
# Model Accuracy, how often is the classifier correct?

print("Accuracy (Using Train Data):",metrics.accuracy_score(y_train, y_train_pred))
print("Accuracy (Using Test Data ):",metrics.accuracy_score(y_test, y_test_pred))
```

```
Accuracy (Using Train Data): 1.0
Accuracy (Using Test Data ): 0.7108843537414966
```

The resulting tree has been visualized in figure 6 (code) and figure 7 (diagram) below. Looking at the diagram, it is clear that the model has been overfitted and we can apply pruning to improve the accuracy from 71%, while reducing the levels of the tree.

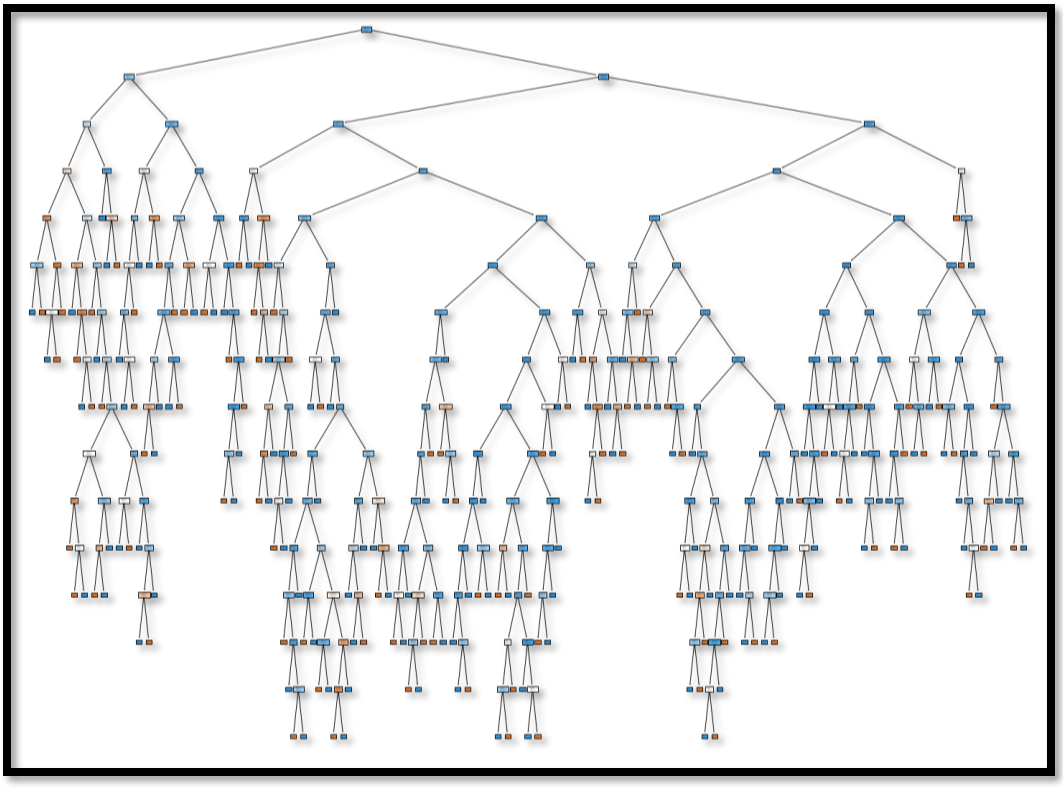
Figure 6

Visualizing decision tree

```
#visualizing the tree

plt.figure(figsize=(20,20))
features = feature_cols
classes = ['No Attrition','Attrition']
tree.plot_tree(clf,feature_names=features,class_names=classes,filled=True)
plt.show()
#plt.savefig("image1.png")
```

Figure 7

Decision Tree



Improving the model

There are several methods of preventing overfitting. In this assignment I am using Pruning, specifically, cost complexity pruning Alpha. Pruning is the process of removing some parts of the tree to make it more adaptable, which might slightly increase the training error but improve the model by decreasing testing error (Arora, 2022)

Minimal cost complexity pruning recursively improves the model by removing nodes that do not have a bi effect on the model, called “weakest link,” which is characterized by an effective alpha. It starts by removing the nodes with the smallest effective alpha (Arora, 2022). Below is the code that was used to prune the tree, starting with getting the alpha values in Figure 8, and using the alpha values in the model to find an optimal alpha to use.

Figure 8

Getting Alpha values

```
#Using ccp to prune
#Getting alpha values

path = clf.cost_complexity_pruning_path(X_train, y_train)
alphas=path['ccp_alphas']

alphas
#ccp_alphas, impurities = path.ccp_alphas, path.impurities
#print(ccp_alphas)
```

```
array([0.        , 0.00055775, 0.00067347, 0.00070482, 0.00074405,
       0.00074405, 0.00075586, 0.00076531, 0.00078493, 0.0007896 ,
       0.00079365, 0.00079719, 0.00080886, 0.00081048, 0.00082745,
       0.00083662, 0.00084022, 0.00085034, 0.00085034, 0.00097596,
       0.00106293, 0.00110408, 0.00113379, 0.00113379, 0.00113379,
       0.00113379, 0.00113379, 0.00113379, 0.00113379, 0.00113379,
       0.00114953, 0.00121882, 0.00123686, 0.00124922, 0.00125232,
       0.00127551, 0.00127551, 0.00127551, 0.00127551, 0.00127551,
       0.00128901, 0.00131856, 0.00132866, 0.0013552 , 0.00140633,
       0.00141717, 0.00141723, 0.00141723, 0.00144042, 0.00150878,
       0.00151172, 0.00151172, 0.00151172, 0.00152344, 0.00153061,
       0.0015361 , 0.00156986, 0.00158472, 0.00177966, 0.00182059,
       0.00183269, 0.0020061 , 0.00202429, 0.00208333, 0.00211271,
       0.00212585, 0.00219254, 0.00219886, 0.00238095, 0.00239023,
       0.00239668, 0.00257199, 0.00257228, 0.00258401, 0.00263863,
       0.00290967, 0.00291812, 0.00301558, 0.00331714, 0.00363692,
       0.00519017, 0.00553265, 0.01270344])
```

The next step substitutes the alpha values into the model to by looping over the alphas array to find the accuracy on both Train and Test parts of our dataset. The result has been plotted on Figure 10 below:

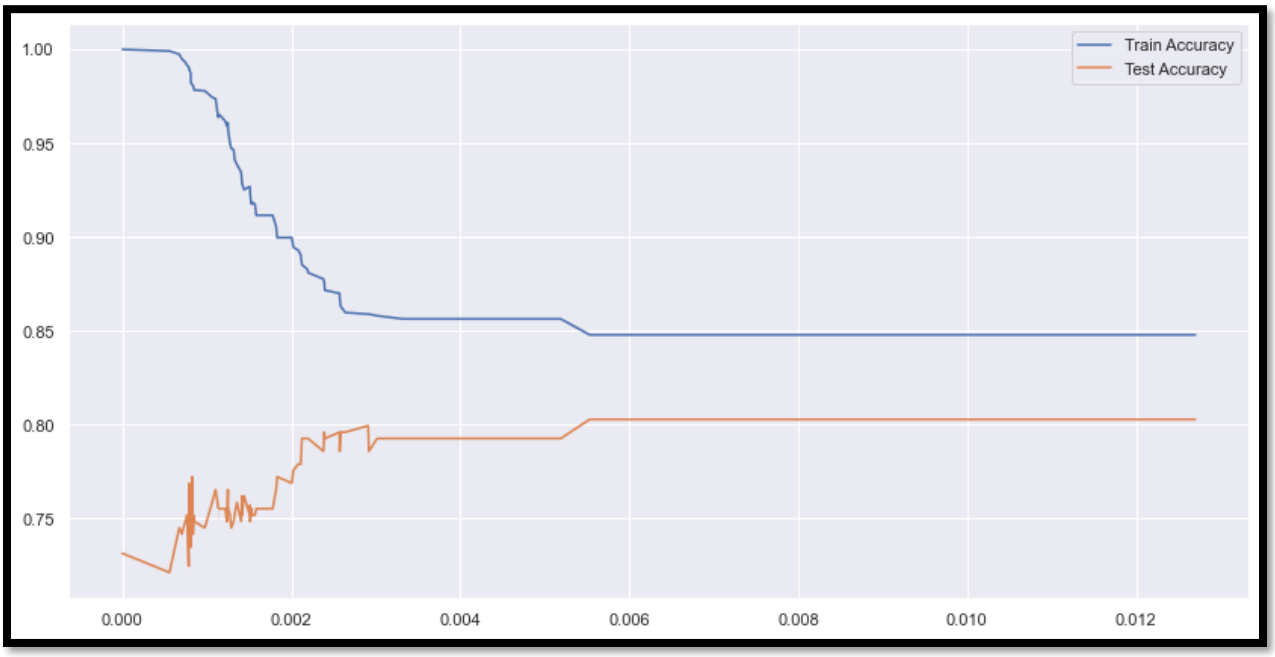
Figure 9

Appending Alpha values

```
# For each alpha we will append our model to a list

#import seaborn
import seaborn as sns


accuracy_train, accuracy_test=[], []

for i in alphas:
    clf=DecisionTreeClassifier(ccp_alpha=i)

    clf.fit(X_train, y_train)
    y_train_pred=clf.predict(X_train)
    y_test_pred=clf.predict(X_test)

    accuracy_train.append(accuracy_score(y_train, y_train_pred))
    accuracy_test.append(accuracy_score(y_test, y_test_pred))

sns.set()
plt.figure(figsize=(14,7))
sns.lineplot(y=accuracy_train, x=alphas, label="Train Accuracy")
sns.lineplot(y=accuracy_test, x=alphas, label="Test Accuracy")
plt.xticks(ticks=np.arrange(0.00, 0.25, 0.01))
plt.show

clfs = []
for ccp_alpha in ccp_alphas:
    clf = tree.DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
```

Figure 10

Accuracy Test



Running Decision Tree Classifier using 0.003 alpha

From the accuracy test plot, we see that there is some sort of convergence from alpha 0.002. I tried several values between 0.002 and 0.004, and finally settled for 0.003 which reduces the Train Accuracy from 100% to 86% but increases the test accuracy to 79% from 71%. More importantly, it generalizes the model by pruning the tree. **Figure 11** below shows the code for running the Decision Tree Classifier again, with alpha value of 0.003, and **Figure 12** shows the resulting decision tree after pruning.

Figure 11

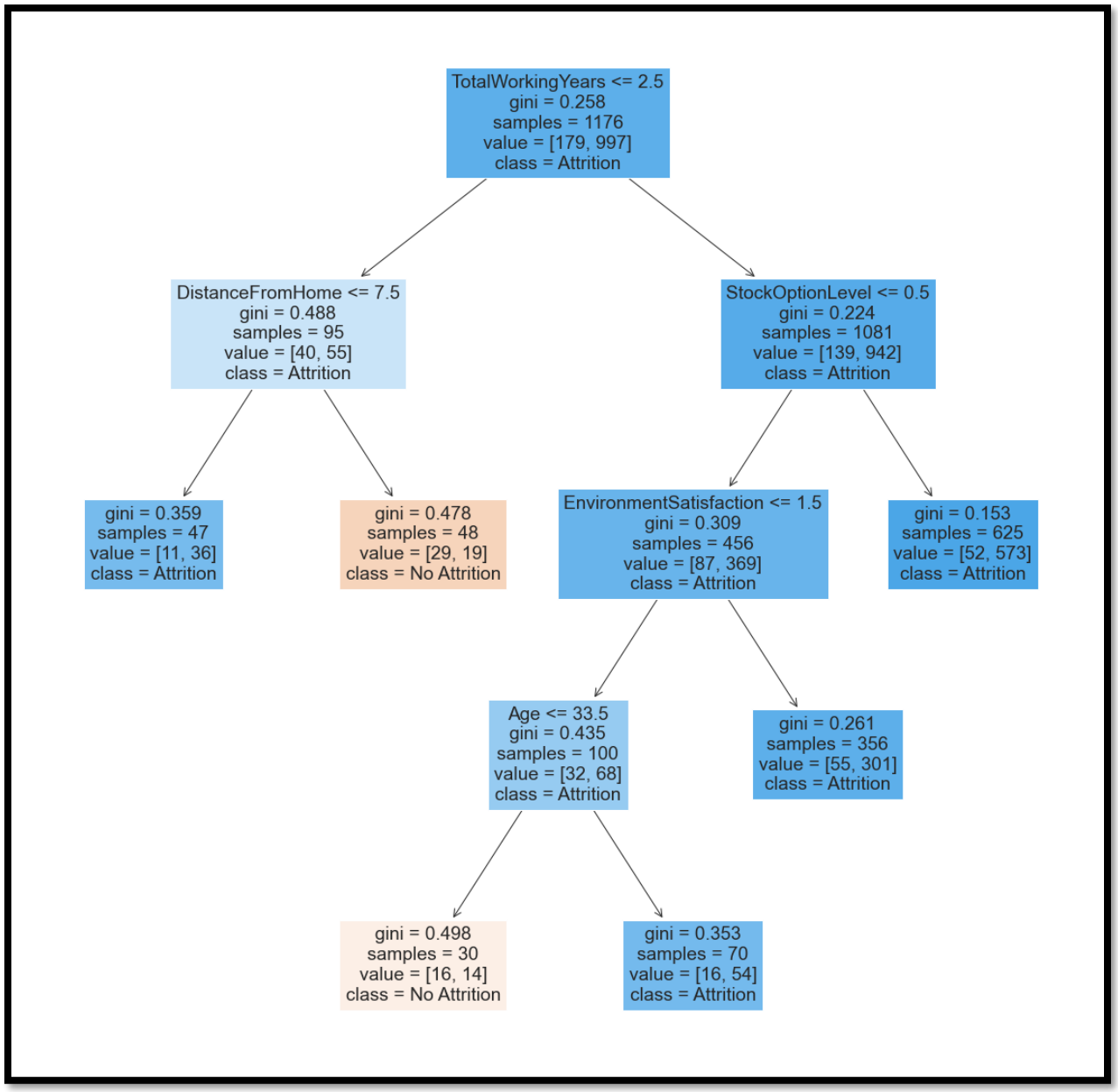
Running Decision Tree Classifier using 0.003 alpha value

```
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(ccp_alpha=0.00301558, random_state=40)

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)


#Predict the response for train dataset
y_train_pred = clf.predict(X_train)


#Predict the response for test dataset
y_test_pred = clf.predict(X_test)

#Accuracy score
print(accuracy_score(y_train, y_train_pred), round(accuracy_score(y_test, y_test_pred), 2))
```

```
0.8579931972789115 0.79
```

Figure 12

The resulting decision tree after pruning



Conclusion

In this project, I created a model for predicting employee attrition using the IBM HR Analytics Employee Attrition & Performance dataset. The model turned out to be overfitted, I improved the model by pruning it, making the tree shorter and improving the accuracy from 71% to 79%.

References

Illowsky, B., & Dean, S. L. (2022). *Introductory statistics*. OpenStax, Rice University.

Chakure, A. (2022, February 10). What is decision tree classification? Built In. Retrieved September 20, 2022, from https://builtin.com/data-science/classification-tree

Pavansubhash. (2017, March 31). IBM HR Analytics Employee Attrition & Performance, Version 1. Retrieved September 21, 2022, from https://www.kaggle.com/datasets/tombenny/foodhabbits

Arora, S. (2022, July 26). Cost complexity pruning in decision trees: Decision tree. Analytics Vidhya. Retrieved September 22, 2022, from https://www.analyticsvidhya.com/blog/2020/10/cost-complexity-pruning-decision-trees/