decision-trees

August 14, 2020

1 Decision Trees

Welcome to your first Jupyter notebook for this course. Here, you will use scikit-learn to build and predict using decision trees. You will also see some other useful things to do that come up in the process of building and using supervised learning QuAMs. So, let's get started!

1.1 Before you start

- In order for the notebooks to function as intended, modify only between lines marked "### begin your code here (__ lines)." and "### end your code here.".
- The line count is a suggestion of how many lines of code you need to accomplish what is asked.
- You should execute the cells (the boxes that a notebook is composed of) in order.
- You can execute a cell by pressing Shift and Enter (or Return) simultaneously.

1.2 Loading the appropriate packages

The first thing we have to do is to load up some packages, including NumPy which is used for matrix and vector manipulation when you are doing scientific computing with Python and also several functions from scikit-learn! We will also load the pandas package, a data analysis library for Python as well as plotly, which is a cool visualization library for Python. We will also some other packages to show visualizations of decision trees.

[1]:
```
import numpy as np
from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.externals.six import StringIO
import pandas as pd
import plotly.express as px
from pydotplus import graph_from_dot_data
from IPython.display import Image
```

NumPy will show floating point (real) numbers in the scientific notation. Let's turn that off.

[2]:
```
np.set_printoptions(suppress=True)
```

1.3 Loading and examining the data

Let's load our data into a pandas data frame (`DataFrame`), which are good tools for manipulating, an in our case for now, displaying the data. We will load our data from a CSV (Comma-Separated Values) file.

[3]:
```
df = pd.read_csv('data_decision_trees.csv')
```

It's very good practice to check your data before trying to build a QuAM. We are talking about the methods themselves now. This seems optional because we have clean and completely useful data and we know exactly what we want to do. However, this will become a crucial thing to do once you get into real-world applications since you have to understand your data and convert it into a useful form. Oftentimes, that is one of the parts of the machine learning process that is going to take the most time. That process will be the focus of later courses in the specialization. However, let's do the good practice anyway. Some of our best tools for checking our data is to see them in tables and figures.

So, let's take a peek at our data (as a table) to see if everything is alright. We will ask the Jupyter notebook to show the data frame object `df` that we loaded with the data from the CSV file. Calling an object will let us inspect that object and in the case of pandas data frames, this shows us the data frame as a table:

[4]:
```
df
```

[4]:

	Feature 01	Feature 02	Feature 03	Feature 04	Feature 05	Feature 06	\
0	0.269186	2.890367	1.893983	-0.420737	-0.828978	-3.815491	
1	-1.327319	-1.762721	1.031584	2.766727	-3.174990	-1.395544	
2	0.107028	3.964838	1.073601	0.758348	1.570392	-0.401011	
3	-2.294776	-3.894267	-0.571486	3.067353	2.892005	-0.235421	
4	-1.635072	4.305268	0.101267	-5.335998	1.072216	1.286368	
5	1.982399	4.372578	-0.459589	-4.381318	1.866153	-0.962641	
6	1.375390	-3.291875	-1.077968	-1.254535	0.670842	-1.593755	
7	1.508717	2.987613	1.105563	-1.212302	3.149322	2.884053	
8	-2.818527	-0.619860	-0.474251	-5.073817	4.037249	2.124911	
9	-0.395904	2.303231	0.002418	-0.946588	2.619193	-0.417250	
10	1.282122	2.493585	1.869116	-1.527131	-2.943975	-2.640694	
11	-2.067966	2.408809	-1.248135	4.308685	-2.547477	-9.237300	
12	4.445966	-1.811825	-0.331432	2.095428	0.977219	0.350585	
13	-2.375471	2.419000	0.694735	-2.673371	-0.156419	-1.064870	
14	2.203714	-1.576822	-0.729784	-2.492998	2.209286	-1.413050	
15	0.439665	3.806928	-1.161985	1.930908	2.661265	-1.458178	
16	0.812320	3.269327	-0.725434	1.694893	2.189416	-2.559073	
17	-1.782249	-1.712106	-0.918737	0.629255	-1.536843	1.610173	
18	1.701166	-2.806427	0.414624	-1.022575	3.485875	-1.248432	
19	4.054705	2.594996	-1.933536	-0.138183	0.590014	-2.431382	
20	3.930392	-2.308039	-1.195974	0.858936	-2.178091	-5.088246	
21	2.152585	3.259644	-0.505315	-1.779510	2.987457	1.921015	
22	0.059572	2.125324	0.242972	-1.019444	2.424114	2.039635	
23	2.313558	-1.385546	1.481299	-3.281206	2.149474	-2.655257	
24	1.976358	-2.234122	-1.217360	-1.934456	1.854428	-2.625161	
25	-1.813869	2.825985	-0.150377	-0.285184	1.301155	2.206264	

26	-3.250344	0.567684	-1.174083	3.752707	-4.054206	-0.319237
27	0.598065	2.797548	-1.485709	-3.120271	-0.288775	-0.541793
28	-0.824521	0.921646	1.049984	-6.998551	1.967715	-0.548932
29	-0.554972	-2.243142	-1.457775	3.468102	0.711039	1.101553
...	...	...	...	...	...	...
1840	2.923271	1.758975	0.546965	-2.050033	0.213961	-2.601036
1841	3.578274	-0.478658	-1.487229	-1.437461	-0.835066	1.545648
1842	0.201246	4.384678	0.226535	1.063453	1.615229	-3.152694
1843	1.446957	3.550228	-0.506157	-4.272294	3.887621	1.931260
1844	1.040928	-4.090928	-0.486581	0.712609	2.121421	-0.650419
1845	-1.473689	-3.306025	0.246507	2.337558	-1.644894	3.068260
1846	-1.441851	-2.193373	-1.376124	0.737402	-1.533085	1.478987
1847	2.739506	2.152111	-1.225977	-0.117749	1.303835	-3.870379
1848	2.763264	-0.685172	1.180424	-3.120198	1.897915	-4.249598
1849	3.750473	-1.481624	-2.082075	2.278408	1.016840	0.918538
1850	-2.621575	4.855816	-0.111189	0.217455	1.570979	3.323710
1851	5.682529	2.282038	2.380964	-2.721518	-1.878965	0.314114
1852	2.172015	2.507307	-0.653161	-3.983377	2.963255	1.989137
1853	-3.795066	-0.804171	-0.496937	2.310091	0.765284	1.437152
1854	-1.566871	-1.257267	-1.633712	0.115845	-0.091387	0.538202
1855	1.230954	-0.421321	-0.624810	3.564808	-1.121385	1.011293
1856	0.655745	-2.411488	0.137698	0.713949	-2.649675	-3.419993
1857	4.407023	-1.073717	-0.725289	0.327215	5.239059	1.713507
1858	-4.587562	-2.387530	-0.272040	0.945135	1.389775	2.652410
1859	-1.502677	-2.660907	-1.872737	1.313576	-1.695232	-3.055933
1860	1.497673	-2.550316	-1.177584	2.220196	2.949008	2.135538
1861	-0.949489	1.552551	-0.812163	-2.840369	2.867859	-2.003784
1862	-2.248376	-2.014347	-0.200545	2.942187	-2.809844	1.789085
1863	3.042143	-2.277677	1.070992	0.258320	-3.493290	-1.438300
1864	1.571651	-2.833445	1.623697	-0.738137	3.644512	-0.810149
1865	-0.478231	2.201073	-1.444069	-2.515373	1.695445	4.027665
1866	0.888738	-2.017106	2.039652	2.001606	1.426542	1.720450
1867	-3.077087	1.400107	0.663180	3.309129	0.672774	2.550213
1868	0.803133	1.580357	0.054739	2.958573	3.340849	-1.811113
1869	-0.480225	3.952867	0.886747	-0.960139	2.574765	0.233662

	Feature 07	Label
0	-1.219257	B
1	-3.284436	D
2	3.332441	C
3	-1.292745	A
4	4.741389	A
5	1.386613	A
6	-3.047652	D
7	2.176984	D
8	3.021518	B
9	3.102944	A

```
10     -3.606983     B
11      3.833692     B
12      3.143418     B
13     -0.498929     B
14     -1.726400     D
15     -1.626649     C
16     -1.259300     C
17      1.330622     C
18     -2.546788     D
19     -3.235028     C
20      0.933449     A
21      2.869276     D
22      2.117995     D
23     -2.570389     D
24     -2.344806     D
25      1.450437     B
26     -3.603252     C
27     -4.744657     B
28      3.236193     B
29     -2.526874     A
...          ...   ...
1840    1.433784     A
1841    1.058680     D
1842   -1.956705     C
1843    3.594661     D
1844   -1.534266     D
1845    0.474445     C
1846    3.357453     C
1847    1.493750     A
1848   -1.409041     D
1849    3.547225     B
1850    1.873439     B
1851   -4.506998     B
1852    3.207010     D
1853   -3.309665     A
1854    2.859516     C
1855    3.570067     B
1856   -1.068708     D
1857    1.461659     B
1858    1.765212     C
1859   -0.771316     D
1860    1.086767     B
1861    0.816581     A
1862    2.460505     C
1863   -1.866993     A
1864   -2.378056     D
1865    1.092670     B
```

```
1866    0.886203     B
1867    0.038810     C
1868    0.176597     C
1869    3.803253     D

[1870 rows x 8 columns]
```

Cool. Everything seems fine. One of the other tools for checking our data is visualizing the data. However, our data has more than 3 features, so we cannot directly visualize it (We can directly see only up to 3D). One good way to check for statistical shape of the data is to use a matrix of scatter plots. We can get a feel for the distribution of our data using that.

We are using data visualization tools from the plotly package. We are going to ask plotly to show us a scatter matrix. However, we care only to see the dimensions of the data that are features and the label part should be used to show different classes in different colours. Also, the plotly will try to fit the entire scatter matrix to the width of our screen and with data with high number features, we don't want that because it may give small pairwise scatter plots, so let's specify a size of 256 pixels for the size of each scatter plot. We ask plotly to create a figure for us and show it:

[5]:
```
data_dimensions = df.columns[:-1].to_list()
figure_size = df.shape[1] * 256

fig = px.scatter_matrix(df, dimensions=data_dimensions, color='Label',␣
 ↪width=figure_size, height=figure_size)
fig.show()
```

Now that everything seems good, let's get our data as NumPy arrays that will be used by scikit-learn algorithms. Our data `X` will be a matrix which has different datapoints in different rows and different features in different columns and since the `'Label'` column of the datafram `df` is not a feature of the data (but rather the label), let's exclude that. Then, our labels (or *targets*) `y` will be a vector consisting of simply that `'Label'` column of data frame `df`.

[6]:
```
X = df.drop('Label', axis=1).to_numpy()
y = df['Label'].to_numpy()
```

Let's see if everything is fine. Let's see our data `X`:

[7]:
```
X
```

[7]:
```
array([[ 0.26918561,  2.89036704,  1.89398331, ..., -0.82897781,
        -3.81549087, -1.21925709],
       [-1.32731927, -1.76272079,  1.03158355, ..., -3.17499021,
        -1.39554398, -3.28443589],
       [ 0.10702806,  3.96483754,  1.07360074, ...,  1.57039201,
        -0.40101058,  3.33244142],
       ...,
       [-3.07708661,  1.40010747,  0.66318026, ...,  0.67277394,
         2.55021337,  0.03881049],
       [ 0.80313316,  1.58035675,  0.05473856, ...,  3.34084891,
        -1.81111266,  0.17659685],
       [-0.48022462,  3.95286668,  0.88674666, ...,  2.57476504,
         0.23366233,  3.80325314]])
```

Good. Now let's check the shape of `X`. We should have 1870 rows since we have 1870 datapoints and we should have 7 columns since there are 7 fetaures on our data:

[8]:
```
X.shape
```

[8]:
```
(1870, 7)
```

Let's do the same check with the targets vector `y`:

[9]:
```
y
```

[9]:
```
array(['B', 'D', 'C', ..., 'C', 'C', 'D'], dtype=object)
```

...and the shape of `y` should be the same number of rows and singular in column (so there is only a row dimension and no column dimension):

[10]:
```
y.shape
```

[10]:
```
(1870,)
```

Perfect! Everything looks fine.

1.4 Splitting the data

Now, remeber we emphasized the importance of splitting your data into train, validation and held-out test sets? Let's put that into use. We will talk about this splitting later on in this course and you don't need to understand how this function works, however, we will provide a short explanation for those of you who are interested.

Unfortunately, scikit-learn does not have a function to divide the data into three parts, only a function that splits the data into two. So, we are goign to call that function twice, once to split data into two sets: *a.* training; and *b.* validation and test combined. Then, we call the function once more to split that second set into distinct validation and test sets. We chose to reserve 0.4 (or 40%) of our data for validation and test and 1 - 0.4 = 0.6 or 60% of our data for training. From the 40% left for validation and test, we are going to use 0.5 or 50% of it (which make sit 20% of the total amount of data) for validation and the other 50% for test.

[11]:
```
(X_train, X_vt, y_train, y_vt) = train_test_split(X, y, test_size=0.4,␣
 ↪random_state=0)
(X_validation, X_test, y_validation, y_test) = train_test_split(X_vt, y_vt,␣
 ↪test_size=0.5, random_state=0)
```

1.5 Building and fitting a decision tree

Finally, it's time to build your decision tree. Remember, we create an object of the class `DecisionTreeClassifier` first. Let's set the name `dtree` for that object. In Python, you create an object of class `Clss` with its parameter `par` set to value `val` and assign it a name of `obj` by saying:

`obj = Clss(par=val)`

You can see the documentation for `DecisionTreeClassifier` here:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

So, go ahead and create that decision tree now (also remember we are not specifying any parametes for our decision tree this time):

[13]:
```
### begin your code here (1 line).
dt_clf = DecisionTreeClassifier()
print(dt_clf)
### end your code here.
```

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            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=False, random_state=None,
            splitter='best')
```

Next step is to train our decision tree model. Again, remember we use the function (or method) `fit` of our decision tree object to do that. In Python, you call a method `mthd_a` of an object `obj`, which takes arguments `arg_a1` and `arg_a2` by saying:

`obj.mthd_a(arg_a1, arg_a2)`

You can see the documentation for `fit` method here:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.De

Now, go ahead and fit your `dtree` to the training data and don't forget to pass the `X_train` training data and `y_train` training targets required for fitting:

[14]:
```
### begin your code here (1 line).
dt_clf.fit(X_train, y_train)
### end your code here.
```

[14]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            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=False, random_state=None,
            splitter='best')
```

If everything went on fine, you should be seeing a summary of the model you trained:

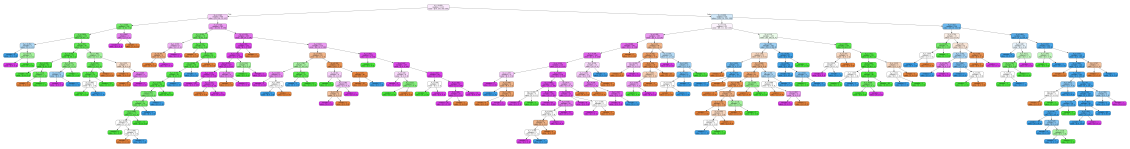
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, min_samples_leaf=1, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')

1.6 Visualizing the decision tree

Now, let's visualize our decision tree. Let's export our model as a special kind of data, create a visual representation form that, generate a graph from that representation and show that graph as an image (it may be a big image, so you may have to scroll to see the whole thing):

[15]:
```
dot_data = StringIO()
export_graphviz(dt_clf, out_file=dot_data, filled=True, rounded=True,␣
 ↪impurity=False, special_characters=True)
graph = graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png(), unconfined=True)
```

[15]:



1.7 Model assessment and selection

So let's evaluate our QuAM. First, let's see what happened on training data. You are going to put in the code that asks our decision tree `dtree` to predict the label for training data `X_train` and assign it to a vector variable `yhat_train`. Remember that the `predict` is the name of the method that asks a model object to predict labels. Again, you call a method `mthd_b` of an object `obj` that takes argument `arg_b1` and assign it to a variable `var` by doing:

```
var = obj.mthd_b(arg_b1)
```

You can see the documentation for `predict` method here:
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.De

So predict the labels of the training data and assign it to `yhat_train`:

[16]:
```
### begin your code here (1 line).
yhat_train = dt_clf.predict(X_train)
### end your code here.
```

Now we can assess the accuracy of our classification model between the `yhat_train` you computed and the actual alebls that came with the training data, `y_train`:

[17]:
```
accuracy_score(yhat_train, y_train)
```

[17]:
```
1.0
```

A perfect 1.0 or 100%! However, that was what was expected given the minimum leaf size was 1, we let the decision tree split leafs with even 2 points in them and we had no tree depth limittation among other things.

Let's use our validation data then. Calculate `yhat_validation` by asking `dtree` to predict labels for `X_validation`. Then, we can calculate the score for validation data.

[18]:
```
### begin your code here (1 line).
yhat_validation = dt_clf.predict(X_validation)
### end your code here.
accuracy_score(yhat_validation, y_validation)
```

[18]:
```
0.8048128342245989
```

Right. The accuracy on validation data is much lower (should be ~80%). Maybe we have overfit to our data. Unrestricted decision trees do that.

Let's create a new decision tree object `dtree2`, but this time let's set a minimum number of samples per leaf. Let's do 15. Go ahead and create `dtree2`, however, this time specify that you want the `min_samples_leaf` to be set to 15. Then, just fit your model to the training data and targets, `X_train` and `y_train`:

[19]:
```
### begin your code here (2 lines).
dtree2 = DecisionTreeClassifier(min_samples_leaf=15)
dtree2.fit(X_train, y_train)
```

```
### end your code here.
```

[19]:
```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=15, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Again, you should be getting a summary of your model parameters if everything went right. In the summary you should be seeing that `min_samples_leaf=15`:

> DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None, max_features=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, **min_samples_leaf=15**, min_samples_split=2, min_weight_fraction_leaf=0.0, presort=False, random_state=None, splitter='best')

Now, predict `yhat_train2`, so we can the accuracy on training data:

[20]:
```
### begin your code here (1 line).
yhat_train2 = dtree2.predict(X_train)
### end your code here.
accuracy_score(yhat_train2, y_train)
```

[20]:
```
0.8440285204991087
```

Training accuracy is predictably lower (~85%) as we restrcited our decision tree, so it does not fit perfectly to training data because of its constraints.

Let's predict `yhat_validation2` and we can the accuracy on validation data:

[21]:
```
### begin your code here (1 line).
yhat_validation2 = dtree2.predict(X_validation)
### end your code here.
accuracy_score(yhat_validation2, y_validation)
```

[21]:
```
0.7887700534759359
```

Yes! We have a better score (~79%) on validation points. We may be ovefitting less this time!

1.8 Evaluating the decision tree

Finally, let's use the test data to get a final accuracy performance number for our model. Predict `yhat_test2` using `dtree2`. We can then calculate the accuracy on test data:

[22]:
```
### begin your code here (1 line).
yhat_test2 = dtree2.predict(X_test)
### end your code here.
accuracy_score(yhat_test2, y_test)
```

[22]:
```
0.7647058823529411
```

Voila! The accuracy on test data (~77%), on validation data and on training data are close to each other, which is a good sign.

Well done!