CART

Not only Classification and Regression Trees

Marc Garcia

Marc 12th, 2016

Overview

Introduction

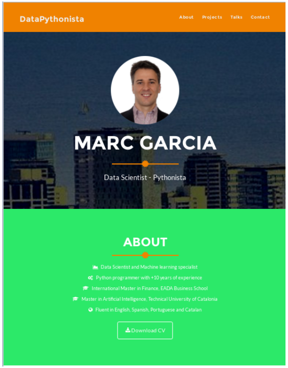
Talk goals

- How decision trees work
- Common problems and advantages
- What they can be used for

Focus on classification

About me

- @datapythonista - http://datapythonista.github.io/
- Python user since 2006
- Active Django developer, 2007 - 2012
  - GSoC 2009: Django localization
- Master in Artificial Intelligence, 2012
- Master in Finance 2014
- Currently working at Bank of America Merrill Lynch in London
- Owner of Quantitative Mining
  - Machine learning applied to digital marketing



Warm up example

Example: The geek party

- We want to **predict attendance** to the geek party
- We know for each possible attendee:
    - Their **age**
    - The **distance** from their home to the event location

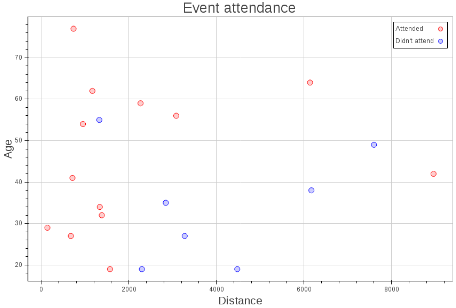
Let's get some data first...

```python
import pandas as pd

data = {'age': [38, 49, 27, 19, 54, 29, 19, 42, 34, 64,
                19, 62, 27, 77, 55, 41, 56, 32, 59, 35],
        'distance': [6169.98, 7598.87, 3276.07, 1570.43, 951.76,
                     139.97, 4476.89, 8958.77, 1336.44, 6138.85,
                     2298.68, 1167.92, 676.30, 736.85, 1326.52,
                     712.13, 3083.07, 1382.64, 2267.55, 2844.18],
        'attended': [False, False, False, True, True, True, False,
                     True, True, True, False, True, True, True,
                     False, True, True, True, True, False]}

df = pd.DataFrame(data)
```

	age	distance	attended
0	38	6169.98	False
1	49	7598.87	False
2	27	3276.07	False
3	19	1570.43	True
4	54	951.76	True
5	29	139.97	True
6	19	4476.89	False
7	42	8958.77	True
8	34	1336.44	True
9	64	6138.85	True
10	19	2298.68	False
11	62	1167.92	True

...and visualize it

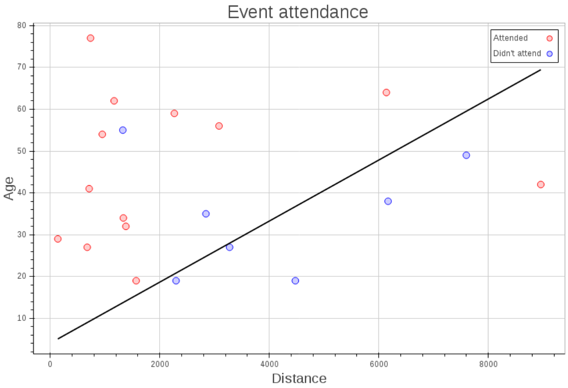
```
from bokeh.plotting import figure, show

p = figure(title = 'Event attendance')
p.xaxis.axis_label = 'Distance'
p.yaxis.axis_label = 'Age'
p.circle(df[df.attended]['distance'],
         df[df.attended]['age'],
         color='red',
         legend='Attended',
         fill_alpha=0.2,
         size=10)
p.circle(df[~df.attended]['distance'],
         df[~df.attended]['age'],
         color='blue',
         legend="Didn't attend",
         fill_alpha=0.2,
         size=10)
show(p)
```



Using a linear model

```
from sklearn.linear_model import LogisticRegression

logit = LogisticRegression()
logit.fit(df[['age', 'distance']], df['attended'])

def get_y(x): return -(logit.intercept_[0] + logit.coef_[0,1] * x) / logit.coef_[0,0]

plot = base_plot()
min_x, max_x = df['distance'].min(), df['distance'].max()
plot.line(x=[min_x, max_x],
          y=[get_y(min_x), get_y(max_x)],
          line_color='black',
          line_width=2)

_ = show(plot)
```

Linear model



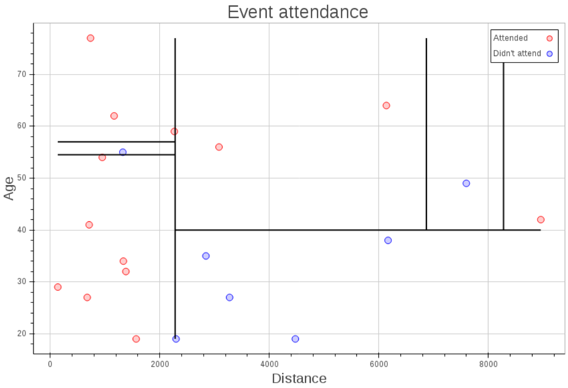
How is the model?

$$\theta_{intercept} + \theta_{age} \cdot age + \theta_{distance} \cdot distance >= 0 \quad (1)$$

Using a decision tree

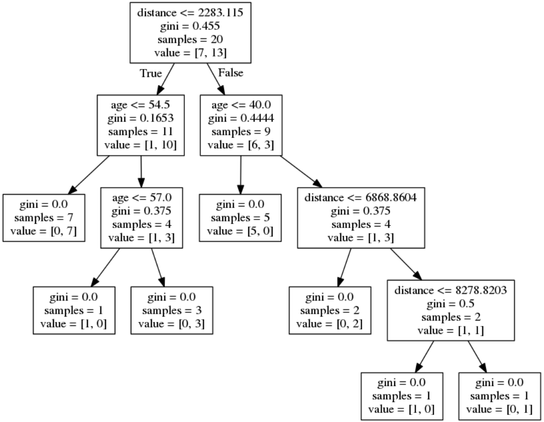
```
from sklearn.tree import DecisionTreeClassifier

dtree = DecisionTreeClassifier()
dtree.fit(df[['age', 'distance']], df['attended'])

cart_plot(dtree)
```

Decision tree



How is the model?

```python
def decision_tree_model(age, distance):
    if distance >= 2283.11:
        if age >= 40.00:
            if distance >= 6868.86:
                if distance >= 8278.82:
                    return True
                else:
                    return False
            else:
                return True
        else:
            return False
    else:
        if age >= 54.50:
            if age >= 57.00:
                return True
            else:
                return False
        else:
            return True
```



Properties

Overfitting

- Decision trees LOVE to overfit
- But it's easy to make them generalize with their parameters:
    - **min_samples_leaf**
    - min_samples_split
    - max_depth
    - max_leaf_nodes

Standardization / Normalization

- Not required
- Using original units we will be able to understand the tree better

Feature selection

- We get feature selection for free
- If a feature is not relevant, it is not used in the tree
- **sklearn**: Gives you the feature importances:

```
>>> list(zip(['age', 'distance'],
...          dtree.feature_importances_))

[('age', 0.5844155844155845), ('distance', 0.4155844155844156)]
```

Feature extraction and data cleaning

- We need to take care of it
- Most of the times, we can improve more the results by **better data**, than by better models

Categorical variables and missing values

- Decision trees can deal with them
- But this depends on the implementation. **sklearn uses float without missing values**

Binning

- Very useful for decision trees
- Two types:
    - Unify values. e.g. 20, 20, 30, 30, 30, 40, 50, 50
    - Binary variables. e.g. lives_in_the_city (yes/no)
- We can add domain knowledge to the classifier
- Avoid **Highly-branching attributes** bias
- Where to cut cyclic variables? e.g. hour of the day

Unbalanced data

- According to the literature, decision tree is biased towards the dominant class
- In my experience, it behaves much better than linear models without changes
- But classical techniques like weighting can be applied

Training

Basic algorithm

```
def train_decision_tree(x, y):
    feature, value = get_best_split(x, y)

    x_left, y_left = x[x[feature] < value], y[x[feature] < value]
    if len(y_left.unique()) > 1:
        left_node = train_decision_tree(x_left, y_left)
    else:
        left_node = None

    x_right, y_right = x[x[feature] >= value], y[x[feature] >= value]
    if len(y_right.unique()) > 1:
        right_node = train_decision_tree(x_right, y_right)
    else:
        right_node = None

    return Node(feature, value, left_node, right_node)
```

Splitting

Candidate split 1

age	18	19	21	27	29	34	38	42	49	54	62	64
attended	F	F	T	F	T	T	F	T	F	T	T	T

Split	True	False
Left	0	1
Right	7	4

Candidate split 2

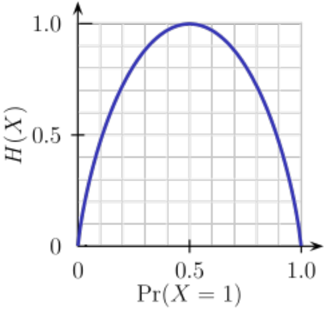
age	18	19	21	27	29	34	38	42	49	54	62	64
attended	F	F	T	F	T	T	F	T	F	T	T	T

Split	True	False
Left	0	2
Right	7	3

Entropy

For a given **subset**[1]:

$$entropy = -Pr_{attending} \cdot \log_2 Pr_{attending} - Pr_{\neg attending} \cdot \log_2 Pr_{\neg attending} \quad (2)$$



[1] Note that pure splits have an entropy of $0 \cdot \infty = 0$

Information gain

For a given **split**:

$$information\_gain = entropy_{parent} - \left(\frac{items_{left}}{items_{total}} \cdot entropy_{left} + \frac{items_{right}}{items_{total}} \cdot entropy_{right} \right) \quad (3)$$

Best split algorithm

```
def get_best_split(x, y):
    best_split = None
    best_entropy = 1.
    for feature in x.columns.values:
        column = x[feature]
        for value in column.iterrows():
            a = y[column < value] == class_a_value
            b = y[column < value] == class_b_value
            left_weight = (a + b) / len(y.index)
            left_entropy = entropy(a, b)

            a = y[column >= value] == class_a_value
            b = y[column >= value] == class_b_value
            right_items = (a + b) / len(y.index)
            right_entropy = entropy(a, b)

            split_entropy = left_weight * left_etropy +
      right_weight * right_entropy
            if split_entropy < best_entropy:
                best_split = (feature, value)
                best_entropy = split_entropy

    return best_split
```

```
import math

def entropy(a, b):
    total = a + b
    prob_a = a / total
    prob_b = b / total
    return - prob_a * math.log(prob_a, 2) \
           - prob_b * math.log(prob_b, 2)
```

Uses

Other uses

- Exploratory analysis
  - Visualize your model to see if it makes sense
  - Detect problems in your data
- Probability estimation
  - Uses frequencies
  - Linear models use distance to the decision boundary, which IMHO is a worse heuristic
- Regression
  - Simple method: Constant value for each split
  - Advanced methods: Linear regression (or other) for each split

Appendix

Tree to nodes

```
from collections import namedtuple
from itertools import starmap

def tree_to_nodes(dtree):
    nodes = starmap(namedtuple('Node', 'feature,threshold,left,right'),
                    zip(map(lambda x: {0: 'age', 1: 'distance'}.get(x),
                            dtree.tree_.feature),
                        dtree.tree_.threshold,
                        dtree.tree_.children_left,
                        dtree.tree_.children_right))
    return list(nodes)
```

Cart plot (I)

```
from collections import namedtuple, deque
from functools import partial

class NodeRanges(namedtuple('NodeRanges', 'node,max_x,min_x,max_y,min_y')):
    pass

def cart_plot(nodes):
    nodes = tree_to_nodes(dtree)

    plot = base_plot()
    add_line = partial(plot.line, line_color='black', line_width=2)

    stack = deque()
    stack.append(NodeRanges(node=nodes[0],
                            max_x=df['distance'].max(),
                            min_x=df['distance'].min(),
                            max_y=df['age'].max(),
                            min_y=df['age'].min()))

# (continues)
```

Cart plot (II)

```
while len(stack):
    node, max_x, min_x, max_y, min_y = stack.pop()
    feature, threshold, left, right = node

    if feature == 'distance':
        add_line(x=[threshold, threshold], y=[min_y, max_y])
    elif feature == 'age':
        add_line(x=[min_x, max_x], y=[threshold, threshold])
    else:
        continue

    stack.append(NodeRanges(node=nodes[left],
                            max_x=threshold if feature == 'distance' else max_x,
                            min_x=min_x,
                            max_y=threshold if feature == 'age' else max_y,
                            min_y=min_y))

    stack.append(NodeRanges(node=nodes[right],
                            max_x=max_x,
                            min_x=threshold if feature == 'distance' else min_x,
                            max_y=max_y,
                            min_y=threshold if feature == 'age' else min_y))

show(plot)
```