Decision Management Community Challenge Jan 2019 Identify Unfriendly Robots

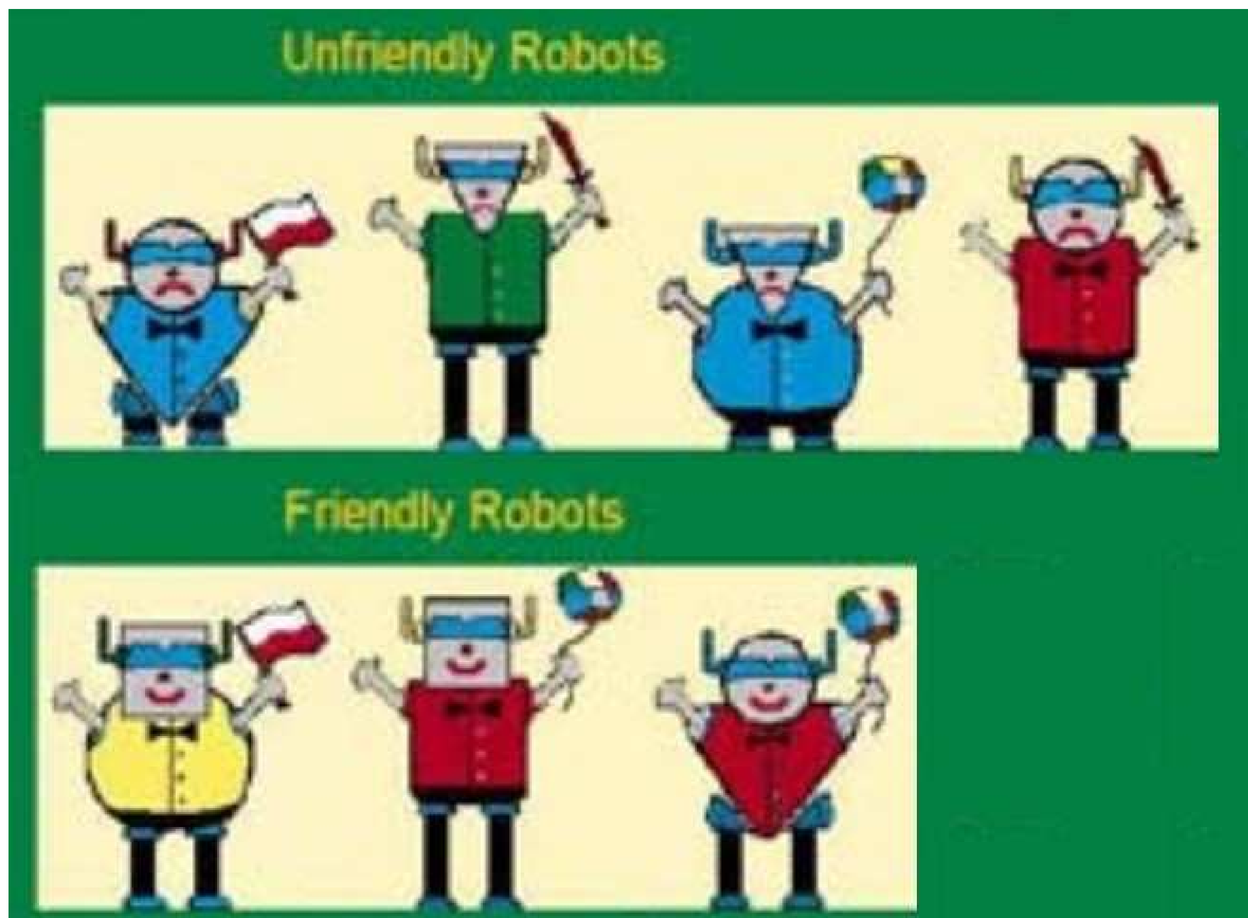
A solution using Python Machine Learning Packages

(Bob Moore, JETset Business Consulting, 31st Jan 2019)

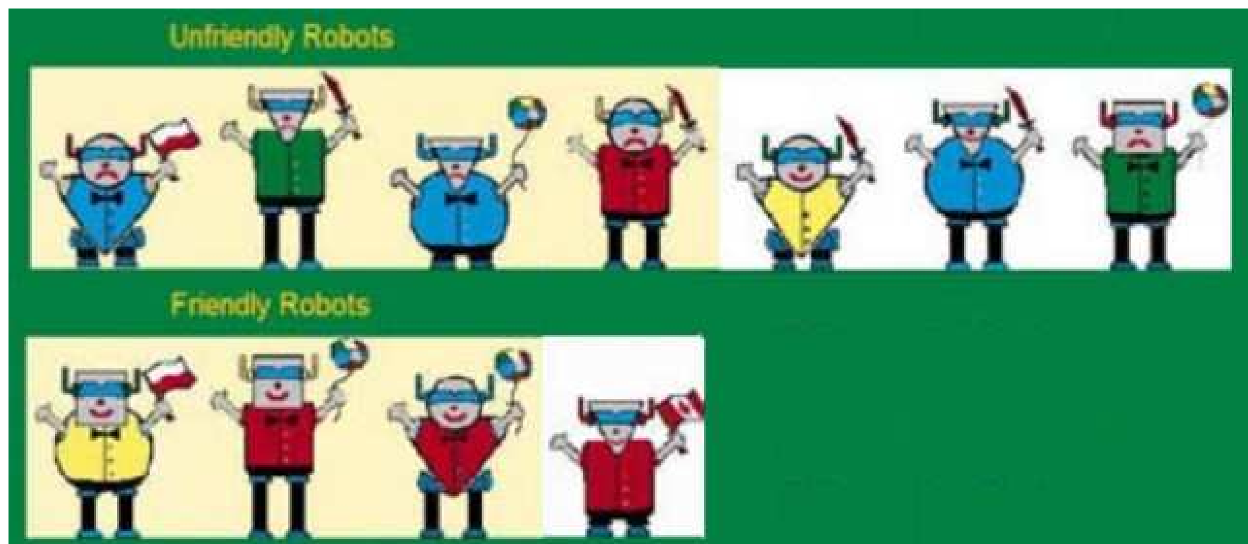
1 Problem Statement (cut and pasted from the web site)

Cyber police received the following information about Unfriendly and Friendly Robots:

The police asked its analysts to specify rules to identify if a robot is friendly or unfriendly, The rules were expressed in terms of any features you can see in robots, such as the shape of the head, the colour of the jacket, the height, the colour of their antennas, what they are holding in their hands, whether they are smiling or not, etc. The analysts quickly identified simple rules that succeeded for the robots from the above lists. Can you guess which rules were used?



However, when the police received more information about robots



the manually determined rules failed miserably. Please help the police to determine rules that can identify if any robot (from these lists or a new one) is friendly or unfriendly. You may use any combination of machine learning and business rules tools to come up with reliable rules.

2 Approaching the Challenge

Due to some pressing concerns of my own I held off having a look at this until a couple of days ago. However, I decided I'd take a quick bash at it to try some practical work on Machine Learning.

The challenge can be viewed as comprising several parts:

1. How do we recognise the attributes of a robot from seeing it?
2. Given a set of examples, generate a decision strategy which matches the exemplars we have been given
3. Provide an execution mechanism

The first part is way and above the most challenging part since it requires us to do feature recognition from images. To avoid writing some image recognition software, I've cheated and done a manual feature extraction. There are a number of features I could have used, but I've gone with body colour, length of legs, the body shape, if the robot is smiling and what it is carrying (as we will see most of these turn out to be irrelevant). The focus of the solution is on step 2, with step 3 largely left to the reader.

The first set of data can be summarised as follows:

Attributes					Classification
Colour	**Legs**	**Body**	**Smile**	**Carrying**	**Friendly**
Blue	Short	Triangle	No	Flag	False
Green	Long	Rectangle	No	Sword	False
Blue	Short	Circle	No	Balloon	False
Red	Medium	Triangle	No	Sword	False
Yellow	Medium	Circle	Yes	Flag	True
Red	Long	Rectangle	Yes	Balloon	True
Red	Medium	Triangle	Yes	Balloon	True

and the second set like this:

Attributes					Classification
Colour	**Legs**	**Body**	**Smile**	**Carrying**	**Friendly**
Blue	Short	Triangle	No	Flag	False
Green	Long	Rectangle	No	Sword	False
Blue	Short	Circle	No	Balloon	False
Red	Medium	Triangle	No	Sword	False
Yellow	Short	Triangle	Yes	Sword	False
Blue	Medium	Circle	Yes	Sword	False
Green	Medium	Rectangle	No	Balloon	False
Yellow	Medium	Circle	Yes	Flag	True
Red	Long	Rectangle	Yes	Balloon	True
Red	Medium	Triangle	Yes	Balloon	True
Red	Short	Rectangle	Yes	Flag	True

For the purposes of the solutions, these have been put into a couple of CSV (comma separated value) files (with a header row)

3 Solution One

There are many techniques for inferring rules from data. Python seems to be the programming language of choice for data analysis currently, so my first thought was to find a Python implementation of Quinlan's ID3 algorithm[1]. This is one of the original methods for creating a fairly 'optimal' decision tree from examples (i.e. by coming up with a classification which minimises the number of tests needed to come up with an answer). When I googled for an implementation though, I discovered that the 'standard' algorithm in Python's learning module (sklearn) is CART[2]. This is a sophisticated evolution from ID3 but the way it is used in Python has some drawbacks as I found.

To build a decision tree using the Python implementation of CART only takes a couple of lines of code, one which creates a classifier, and the second which builds the tree:

```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(attributes, classification)
```

Unfortunately, there are a few hurdles to overcome to generate the attributes and classification and then a bit of a question as to what to do next.

The first complication is that currently the sklearn implementation of CART only supports numerical values and the data we have uses categorical values, so if you use the data directly then you get a run time error when Python finds you have used a string value like 'Flag' instead of a number. One way around this is simply to replace the categorical values with numbers with a mapping like:

Blue = 1, Green = 2, Red = 3, Yellow = 4

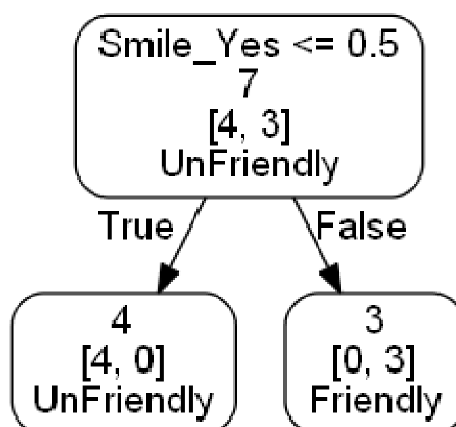
[1] See https://en.wikipedia.org/wiki/ID3_algorithm
[2] See https://en.wikipedia.org/wiki/Predictive_analytics#Classification_and_regression_trees_.28CART.29

This approach is not recommended, however. Doing this means the CART algorithm effectively assumes that 'Yellow' is bigger than 'Blue' which can lead to some slightly odd behaviour in the tree[3].

The 'approved' approach is to create indicator variables, one for each categorical value (these are known as 'dummy variables'). So, for example instead of having one column for 'Colour', one has four (binary) columns, one which says if the robot's body is Blue, one if it is Red etc. In practise one of the columns will be redundant (if the robot's body is not Red or Green or Yellow it must be Blue). While in principle this is tedious, there are utility methods in the Python Pandas package which make it a fairly painless process. Specifically, we can import the data set using Pandas `read_csv` method, then generate the dummy variables (dropping the redundant one) with a call to `get_dummies`, giving the names of the categorical columns (i.e. all of them except 'Friendly' which is recognised as a Boolean attribute), so the whole business is again only a couple of lines of code.

```
robotsRaw = pd.read_csv(path + "\\" + datafile + ".csv")
robotsFixed = pd.get_dummies(
    robotsRaw[list(robotsRaw.columns.values)],drop_first=True)
```

The second issue is once the `DecisionTreeClassifier` instance has calculated a decision tree what do you do with it? One simple thing you can do is to generate a picture of the decision tree using the Python graphviz package[4]. Somewhat more convoluted, you can walk the decision tree and from it generate a Python function which implements the tree[5]. The appendix gives the full Python code, so let's just look at the outputs. The decision tree which comes out of the first set of data looks like this:



What this means in general terms is fairly obvious. The details need a bit more work. 'Smile_Yes' is the dummy variable generated from the initial attribute 'Smile'. If 'Smile_Yes' is not equal to 1, the robot is not smiling (0.5 appears simply because this is halfway between the possible values of 0 and 1. The various numbers in the box

---

[3] I skipped trying to do this, but since the derived 'rules' are so simple, it would probably have not been an issue for this data set

[4] See https://scikit-learn.org/stable/modules/tree.html for the code for generating a decision tree diagram

[5] The code for generating a functional implementation is derived from the code samples at https://stackoverflow.com/questions/20224526/how-to-extract-the-decision-rules-from-scikit-learn-decision-tree

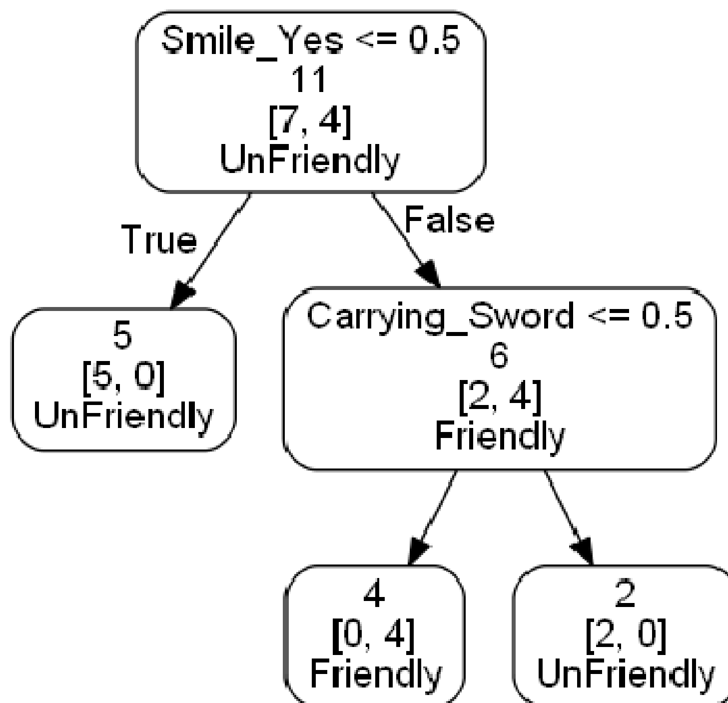
indicate how the data set is being segmented, so the initial data set comprises seven examples, 4 of which are unfriendly and 3 which are friendly and so on.

The generated implementation code in python looks like this:

```
def tree(Colour_Green, Colour_Red, Colour_Yellow,
     Legs_Medium, Legs_Short, Body_Rectangle,
     Body_Triangle, Smile_Yes, Carrying_Flag, Carrying_Sword):
# decision tree implementation
# -----------------------------
  if Smile_Yes <= 0.5:
    return False
  else:  # if Smile_Yes > 0.5
    return True
# -----------------------------
```

I'd judge that even if you don't know Python, it's pretty easy to follow what the code is doing.

For the more extensive set of data, the tree looks like this:



And the implementation code in python for this second tree is:

```
def tree(Colour_Green, Colour_Red, Colour_Yellow,
     Legs_Medium, Legs_Short, Body_Rectangle,
     Body_Triangle, Smile_Yes, Carrying_Flag, Carrying_Sword):
# decision tree implementation
# -----------------------------
  if Smile_Yes <= 0.5:
    return False
```

```
    else:  # if Smile_Yes > 0.5
      if Carrying_Sword <= 0.5:
        return True
      else:  # if Carrying_Sword > 0.5
        return False
# ----------------------------
```

Again it shouldn't be a challenge to make sense of the code

4 Solution Two

Getting to a decision tree solution using the CART implementation in sklearn only uses a half dozen lines of actual code. There is a lot more code involved in extracting the result into a useful form, and the use of dummy variables make the solution difficult to map to the original problem. In particular the generated functions are rather distant from the inputs. They take ten parameters, but there were only five input attributes to start with. With more work one could improve the generation process, but things seemed to be getting a bit laboured.

So, I decided to search for a solution which supported categorical data and found one which is based on ID3[6]. This turned out to be delightfully easy to use! All you need to do is set up a small configuration file giving the location of the CSV file (we use the existing format), the names of the columns and the 'target' attribute. The configuration file is in JSON format for example:

```
{
   'data_file' : './robotsSmall.csv',
   'data_project_columns' :
     ['Colour','Legs','Body','Smile','Carrying','Friendly'],
   'target_attribute' : 'Friendly'
}
```

And once you have this, you need to do is type the following line in a console window:

```
python id3.py configuration.cfg
```

to get a set of simple rules which could then be simply migrated into a rule engine. The results from running this against the small data set are as follows:

```
IF Smile EQUALS Yes THEN True
IF Smile EQUALS No THEN False
```

Which corresponds to the rules from CART (True means the robot is friendly), but is rather easier to read since we don't have the slightly confusing dummy variable naming conventions. Applying ID3 to the larger data set output gave me a bit of a surprise:

```
IF Colour EQUALS Yellow AND Carrying EQUALS Balloon THEN False
IF Colour EQUALS Red AND Carrying EQUALS Balloon THEN True
IF Colour EQUALS Blue THEN False
IF Colour EQUALS Green THEN False
```

[6] See https://github.com/tofti/python-id3-trees

```
IF Colour EQUALS Yellow AND Carrying EQUALS Flag THEN True
IF Colour EQUALS Yellow AND Carrying EQUALS Sword THEN False
IF Colour EQUALS Red AND Carrying EQUALS Sword THEN False
IF Colour EQUALS Red AND Carrying EQUALS Flag THEN True
```

We get a lot more rules and the first rule in particular is counter-intuitive since we have no examples of yellow robots carrying balloons, so why have we got a rule about them[7]? This output to some degree reflects that CART has had a lot of work put into it to improve on ID3 in terms of trimming down the number of conditions and the number of rules required.

5 Discussion

This is a trivial problem to address, but it has been fun playing around with some of the Python tools for machine learning.

Let's take a little step back. Once we are past the feature extraction step, the data set we end up with is in essence a decision table, but one where we usually have more rows that we need for the decisions we do know about and usually a large set of scenarios where the decision table has no corresponding rows. Algorithms like ID3 and CART are aimed at trying to deduce a decision tree or set of rules which correctly handle the known cases and give plausible guesses at how to address unknown cases.

However, we can also be sure that whatever algorithm we use and whatever rules we come up with, we cannot meet the challenge's demand "to determine rules that can identify if any robot (from these lists *or a new one*) is friendly or unfriendly". The rules can only be sure of identifying robots from the current (small) set of examples to work with. Whatever rules we come up with the next robot we see could violate them.

The big plus point of using a machine learning approach to these kinds of decisions is that as new examples appear it's pretty easy to rerun the algorithm to adjust/improve the rules. However, the downside is that simply using statistics-based processing algorithms to generate rules ignores the fact that for most decision-making problems we have a considerable amount of domain knowledge to guide how we do things.

The ID3 solution suggests that (on the basis of the examples) we should avoid Green and Blue robots. The rules are certainly valid given our current state of information, but these colours are not something we would normally associate with being unfriendly. On the other hand, domain knowledge tells us that if we meet someone waving a sword above their head, we should be wary. Also, if they are smiling, they are (usually) friendly, so if they are not smiling, they are probably unfriendly. So, our knowledge of the domain taken with the examples naturally leads us to the rules:

```
IF Carrying EQUALS Sword THEN Unfriendly Robot
IF Smile EQUALS No THEN Unfriendly Robot
OTHERWISE Friendly Robot
```

These comprehensively cover the example set and is very close to what the CART algorithm tells us, but we can come up with these rules by ourselves in only a few

[7] It is possible the code is not fully following the ID3 algorithm, I've not gone through it line by line to check, but the generated rules are 'correct' in that they give the correct answer for each of the examples.

seconds using our ‘common sense’[8] instead of spending a few hours writing and running machine learning algorithms.

However, statistics does give us some extra information over heuristics. We usually work from limited information, so we may know some attributes not all. From the decision tree diagrams, we see 7 out of 11 robots are unfriendly (64%). So, if we just see a robot, with no other information we know that nearly 2 times out of 3 it’s bad news. However, once we know the robot is smiling, then 2 times out of 3 the robot is going to be friendly so we can let them approach with more confidence[9].

6 Appendix – code for CART solution

```
# import various packages required
import pandas as pd
import graphviz
from sklearn import tree
from sklearn.tree import _tree

# create a Python function to implement the decision tree
# Don't worry if you can't make sense of it, but basically it
# recursively walks the decision tree, generating if statements
# at each branch and then a return statement at each leaf node.
def tree_to_code(tree, feature_names):
    tree_ = tree.tree_

    feature_name = [
        feature_names[i] if i != _tree.TREE_UNDEFINED else "undefined!"
        for i in tree_.feature
    ]
    print("def tree({}):".format(", ".join(feature_names)))

    def recurse(node, depth):
        indent = "  " * depth
        if tree_.feature[node] != _tree.TREE_UNDEFINED:
            name = feature_name[node]
            threshold = tree_.threshold[node]
            print("{}if {} <= {}:".format(indent, name, threshold))
            recurse(tree_.children_left[node], depth + 1)
            print("{}else:  # if {} > {}".format(indent, name, threshold))
            recurse(tree_.children_right[node], depth + 1)
        else:
            result = tree_.value[node][0][1] > 0
            print("{}return {}".format(indent, result))

    print("# decision tree implementation")
    print("# ---------------------------")
    recurse(0, 1)
    print("# ---------------------------")


# create a decision tree from a CSV file
# path specifies the folder where the CSV is found, datafile the name
```

[8] If you remove the ‘Colour’ column from consideration the ID3 algorithm output reduces to this set of rules also

[9] Again. thinking heuristics vs statistics, determining if a robot is smiling or not first gives a smaller decision tree, but from a practical perspective you can usually see if a robot is waving a sword or not around before you see if it’s smiling.

```
# of the file less the extension
def classifyRobots(path, datafile):
    # read in the data as a csv file
    robotsRaw = pd.read_csv(path + "\\" + datafile + ".csv")

    # generate dummy indicator variables to replace categorical values,
    # droppingthe redundant first indicator variable
    robotsFixed = pd.get_dummies(
            robotsRaw[list(robotsRaw.columns.values)],drop_first=True)

    # extract the attributes columns and the classification column
    # for DecisionTreeClassifier
    attributes = robotsFixed.drop("Friendly", axis=1)
    classification = robotsFixed["Friendly"]

    # built the tree
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(attributes, classification)

    # output the results
    attributeNames = list(robotsFixed.columns.values)
    attributeNames.pop(0)
    dot_data = tree.export_graphviz(
            clf, out_file=None,
            feature_names=attributeNames,
            class_names=['UnFriendly', 'Friendly'],
            label=None, impurity=False, rounded=True)
    graph = graphviz.Source(dot_data)
    graph.format = 'png'
    graph.render(datafile)

    # generated a Python function representing the tree
    tree_to_code(clf, attributeNames)

# This is just 'main' code to execute the functional logic
# using the two data sets
classifyRobots(r"C:\Projects\DMCommunity\201901", "robotsSmall")
classifyRobots(r"C:\Projects\DMCommunity\201901", "robotsLarge")
```