Classical Boundary Trees Implementation

Jay Ricco
riccoj@wit.edu

June 20, 2017

1 An Introduction

This guide aims to be everything you need to implement (and test) the Boundary Tree algorithm. Initially, the assumption is that you have no prior knowledge in machine learning - if this is not the case, sections where you are able to skim over and still understand will be marked as such.

In the Boundary Tree paper's introduction, the authors' motivations were defined as seeking an algorithm that is:

1. Fast to train.
2. Fast to query.
3. Able to deal with arbitrary data distributions.
4. Able to learn incrementally, in an online setting.

blah blah blacoskjdflkj

2 The Real Deal

Let's start by talking about features, and the conventions surrounding datasets. These terms show up quite a bit in the conversation surrounding machine learning, and it always helps to start from the bottom and solidify the foundations.

2.1 I Swear, It's a Feature! (and Datasets, too)

A **feature** is some variable or predictor that represents a measurable quantitative or categorical value from an example, as part of some machine learning problem.
To provide an example, let's say we want to know if a tumor is benign or malignant. One feature could be the tumor's width, measured from x-rays. Perhaps another could be cell

size, and yet another, cell wall smoothness. The classes, of course, would be malignant and benign. Then, by viewing thousands of examples and being told what the actual class is - machine learning algorithms aim to correctly classify new examples without future help.

From the chosen features, the values they take on defines all we will ever know (in terms of the algorithm's point of view) about a given example. Features are the basis from which all machine learning models are built, so taking the time to choose informative ones is important.

Speaking of which, if you're trying to build a model, especially a complicated one - there could be an infinite number of possible features to use; so how do you know which are informative - which are "good"? Well, that's a complex question, and in fact there's a whole sector of research dedicated specifically to choosing appropriate features. It's called **feature selection**, the process of identifying the distinguishing characteristics of an example, with respect to what class it falls into. A good feature would be one that represents a value, highly correlated with what class it is.
Identifying and utilizing good features is essential in tackling classification problems for a few key reasons. Bringing up two, first and most obviously, the algorithm will intuitively do a better job if it has more relevant information to work with. Classifiers given poor features end up like a family game of Pictionary. The classifier sits there trying to guess what two misshapen triangles, seemingly random dots, and last night's dinner menu represent - and there's just no way... Second, the goal is few and informative. Having a large number of features means a high-dimensional feature vector, which then means slow, heavy computation. Large, floating point vector operations are always a bad thing.

These "examples" that have been spoken of so much originate from datasets. To make the definition official, datasets are usually large, often massive collections of labeled examples used to train and then test the accuracy of machine learning algorithms. There are plenty of informational resources and libraries to peruse with a quick Google search, so further explanation won't be provided.

2.2 Implementing Boundary Trees

Let's start by importing NumPy[1] under the alias *np*. We need this because most of NumPy's functions are pre-compiled c, meaning they're very fast. Also, let's import the python library copy. Everything in Python is pass-by-reference, so we'll need a way to copy an object such that we don't edit the original.

```
1    import numpy as np
     import copy
```

Next, we're going to define the class that represents a Boundary Tree. We don't necessarily need to do this, but it helps with logical grouping, abstraction, and testing. The assumption is that you understand the intricacies of defining classes in Python, if not - Google it (I need

[1]If you're unfamiliar with NumPy, you can look at an introduction and helpful resources here: http://www.numpy.org/

to finish this ASAP).
The class is going to be a subclass of Python's main object class. We're going to need a constructor for initializing the class's properties, as well - the methods `query`, `train`, and a static method that is the implementation of the distance function. The "**" means perhaps additional arguments, if you see fit - the symbolism has no meaning to Python, and in fact - I assume the program won't run if you add specifically that in.

```
class Boundary Tree(object):
  @staticmethod
  def distance(x1, x2):
    //Distance function implementation

  def __init__(self, k, root_x, root_y, **):
    //Constructor implementation

  def query(self, test_x, **):
    //Query Implementation

  def train(self, new_x, new_y, **):
    //Train Implementation
```

2.2.1 Implementing The Distance Function

The distance function, according to the paper, can be any functional that satisfies the axioms of a distance metric[2]. In our case, we'll be using Euclidean distance:

$$d(\vec{p},\ \vec{q}) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

.
The implementation here is more important than any other, because this function gets called for every child, at every level of the Boundary Tree. If this function isn't implemented with speed in mind, the algorithm as a whole is going to perform poorly.
This is where NumPy comes in; because most of the library's functions are implemented as pre-compiled C, we'll get the best performance for the least effort.
As another example of why we use NumPy is for a reason known as *vectorization*. **Vectorization** is the process of codifying your algorithm in such a way that operations on vectors are done atomically as opposed to in loops through the vector's individual elements. To give an example, here's how easily element-wise vector multiplication is in NumPy:

```
In [1]: import numpy as np
In [2]: x1 = np.array([1, 2, 3, 4, 5, 6, 7, 8], dtype=np.float32)
In [3]: x2 = np.array([2, 7, 2, 8, 4, 6, 2, 8], dtype=np.float32)
In [4]: np.multiply(x1, x2)
Out [5]: array([2., 14., 6., 32., 20., 36., 14., 64.], dtype=float32)
```

[2]https://en.wikipedia.org/wiki/Metric_(mathematics)

Of course, these functions can also handle scalars, so you can see why using this library is essential - it handles everything.

Now, let's actually implement the distance function. We're making it a static method only to be able to logically group it with the class. Essentially, the process boils down to taking the operations in the equation above, and applying NumPy's defined methods for each, to our feature vectors.

```
        @staticmethod
        def distance(x1, x2):
          return np.sqrt(np.sum(np.square(np.subtract(x2, x1))))
```

Note: the square-root function is not technically necessary - since square roots are monotonic. (i.e. if $x1 < x2$, then $sqrt(x1) < sqrt(x2)$). Another benefit to not having the square root applied is that our arithmetic can be done totally as integers - speedy.

2.2.2 Implementing The Constructor

The class constructor is where we're going to initialize all the properties and variables that are required. Remember that the functions which define the Boundary Tree algorithm require a pre-rooted tree, an integer value k, which represents the maximum number of children allowed per node.
Speaking of that - we need a tree, and what the heck do we do for that? Well - there's no base implementation of a tree in Python, and speed is always a concern - so we'll implement our own, lightweight tree which leverages on Python's Dictionary class.

In order to do this, we'll have two dictionaries - one which represents all data stored in the tree. The keys will be unique ID's, created from the counter which stores the number of nodes in the tree, the values will be the data themselves. We'll also have a dictionary which is keyed again by a node's ID, and contains as a value, a list of ID's which are the children of that node. $root_x$ and $root_y$ are a feature vector and class label, respectively, randomly sampled from the training set - in order to "root" our tree.

```
        def __init__(self, k, root_x, root_y):
          self.max_children = k
          self.root_node_id = 0
          self.count = 1 # We've already initialized root as 0, so start with
    1.

          # Make sure all of the data is the same type, so we'll force it -
    here
          # and anywhere else we get data from the "outside".
          root_data = (np.asarray(root_x), np.asarray(root_y))

```

```
            self.data = {self.root_node_id: root_data}
            self.children = {self.root_node_id: []}
```

There we go, we've now written the constructor for our tree.

2.2.3 Implementing the Query Function

```
        def query(self, test_x, internal=False):
            current_node_id = self.root_node_id
            while True:
                children = self.children[current_node_id]
                if self.max_children == -1 or len(children) < self.max_children:
                    children = copy.copy(children)
                    children.append(current_node_id)

                closest_node_id = min(children, key=lambda child_node_id:
    BoundaryTree.distance(self.data[child_node_id][0], test_x))
                if closest_node_id == current_node_id:
                    break
                current_node_id = closest_node_id
            if internal:
                return current_node_id # return the node id
            else:
                return self.data[current_node_id][1] # return the class label
    itself, for convenience.

```

2.2.4 Implementing the Train Function

```
        def train(self, new_x, new_y):
            closest_node_id = self.query(new_x, internal=True)
            if not np.array_equal(self.data[closest_node_id][1], new_y):
                # create a new node in the tree
                new_node_id = self.count
                self.count = self.count + 1
                self.data[new_node_id] = (np.asarray(new_x), np.asarray(new_y))
                self.children[new_node_id] = []
                # link it, as a child of the closest
                self.children[closest_node_id].append(new_node_id)
```

2.3 Putting It All Together

Now that you have the base knowledge of a working implementation, play around with it! Let's load up MNIST, a popular classification benchmark dataset, and evaluate the classifier.

2.3.1 Opening MNIST

Yes, this is a complete subsection. MNIST is packaged in a unique file type which requires either a decoder written by yourself - or by someone else.
We're going to use a prepackaged solution[3], and here's what you need to do to make it all work.
First, you'll need to make sure you have PIP installed. (If you're using Anaconda - it's already done.) Once you do, install the MNIST decoding package like so:

```
pip install python-mnist
```

Once that is complete, let's program a driver to handle all of this.

2.3.2 The Driver

```
if __name__ == "__main__":
    import random
    from mnist import MNIST

    data = MNIST('./MNIST_data/', return_type='numpy')

    """ TRAINING PROCEDURE """
    x_train, y_train = data.load_training()

    num_examples = len(x_train)
    selection_list = range(num_examples)
    random.shuffle(selection_list)
    training_examples = [(x_train[i], y_train[i]) for i in selection_list]

    # Initialize the Boundary Tree
    root_example = training_examples[random.randint(0, num_examples)]
    boundary_tree = BoundaryTree(k=-1, root_x = root_example[0], root_y = root_example[1])

    iter_count = 1
    max_iters = num_examples # You can set this to whatever you want, if time is of the essence.

    print("Beginning Training...")
    for (ex_feats, true_class) in training_examples:

      boundary_tree.train(ex_feats, true_class)

      # Other stuff for keeping track of progress and when to stop.
      percent_complete = round((iter_count/float(num_examples)*100.0), 1)
      if percent_complete % 1 == 0 or iter_count == 1:
        print("%d percent complete" % int(percent_complete))
```

[3] https://pypi.python.org/pypi/python-mnist/

```
            iter_count += 1
            if iter_count >= max_iters:
              break
          print("Training Done!")

          """ TESTING PROCEDURE """
          x_test, y_test = data.load_testing()

          num_examples = len(x_test)
          selection_list = range(num_examples)
          random.shuffle(selection_list)
          test_examples = [(x_test[i], y_test[i]) for i in selection_list]
          num_correct = 0.0
          iter_num = 0.0

          print("Running Monte Carlo Accuracy Test...")
          for (ex_feats, true_class) in test_examples:
            class_guess = boundary_tree.query(ex_feats)
            if np.array_equal(class_guess, true_class):
            num_correct += 1.0
            iter_num += 1.0
            print("Accuracy ( @ Iteration %d): %.5f" % (iter_num, (num_correct
    /iter_num)*100.0))
          print("Testing Complete!\nFINAL ACCURACY: %.5f percent correct." %
    ((num_correct/iter_num)*100.0))
```

I'll add more to this if needed, but I think this gets you up and running.

Appendices

A Algorithm Pseudo-Code [1]

A.1 Querying Procedure

Algorithm 1: The Boundary Tree Query Function

Data: `bt`, tree with root initialized.
Data: k, the maximum number of children allowed per node.
Input: $\vec{x_q}$, the feature vector for the tree to classify.
Output: n, the node in the tree that has the *closest* features to those in $\vec{x_q}$.

```
1  begin
2      n := bt.root                          // Initalize transition variable to tree's root
       /* Iterate, moving down the tree until it reaches a node that is either a leaf,
          or has no children that are closer                                          */
3      while true do
4          possibilities = set(n.children)        // Copy over the node in the transition
             variable's children, store it as possible options to move to next
5          if possibilities.length < k then
6              possibilities.add(n)  // If we have room for more children, the node itself
                 becomes a possiblity
7          closest_node = arg min_{node ∈ possibilities}(dist(node.x⃗, x_q⃗))
8          if closest_node == n then
9              break
10         n = closest_node
11     return n
```

[1] The pseudo-code has been modified slightly from it's appearance in the original work for the purposes of readability.

A.2 Training Procedure

Algorithm 2: The Boundary Tree Training Function

Data: `bt`, tree with root initialized.
Input: $\vec{x}_{new}$, the new training example's feature vector.
$\vec{y}_{new}$, the new training example's one-hot encoded class label.

begin
    $closest\_node = BOUNDARYTREE\_QUERY(\vec{x}_{new})$; `// Start by finding the closest node in the tree to our new example.`
    `/* If the closest node's class is the same as our new example's, the tree's decision boundary doesn't need to be updated. Otherwise, add in the new example as a child to the closest node. */`
    if $closest\_node.\vec{y} \neq \vec{y}_{new}$ **then**
        $n_{new} = Node\,(\vec{x},\ \vec{y}_{new})$
        `bt`$.insert\_node\_as\_child(from:\ closest\_node,\ to:\ n_{new})$