High-Dimensional Data
Statistics 650/750
Week 6 Thursday

Alex Reinhart and Christopher Genovese

5 Oct 2017

Announcements

- All projects are posted under the `challenges` tag in the problem bank
- Some good dynamic programming assignments up
- Second language assignments coming soon
- We're about a third of the way through the semester; you should aim to have 3 or 4 approved assignments by now

High-dimensional data

The nature of high-dimensional data

High-dimensional data is becoming increasingly common in statistics. It's easier and easier to use inexpensive technology to record zillions of variables at once, such as in a gene microarray or an online recommendation system. Even two- or three-dimensional data is getting more common: with a GPS unit or smartphone, you can record spatial or spatiotemporal data yourself.

It's tempting to think of high-dimensional data as just ordinary data with more variables, but it's more complicated than that. Even in two dimensions, simple tasks can get difficult with large datasets: How do I efficiently locate all points in a given region? How do I find the nearest neighbors of a point? How do I do kernel smoothing on millions of points in thousands of variables?

I'll use spatial data as a motivating example here, since it's easiest to think in two dimensions, but many of the methods we'll discuss generalize to higher dimensions.

A key theme today: trees and tree traversals can solve important statistical problems.

Another key theme: The *curse of dimensionality*. When we store data with more and more dimensions, the *volume* of the space increases dramatically, and the data is more and more sparse. Efficiently organizing the data so it's easy to find points we want becomes necessary.

Querying high-dimensional data

Efficiently searching and querying spatial data requires good data structures. A simple list of coordinates won't suffice – querying all points in a certain region, the nearest neighbor to a location, and so on, would all be O(n) or worse. We can do better with trees and indexes.

k-d trees

The k-d tree is a relative of binary trees in higher dimensions. k-d trees are effective for data with moderate numbers of dimensions – they work great for ordinary spatial data, but are less efficient for high-dimensional data.

See the `kd-tree` vignette for details on implementation. Broadly: the tree works by splitting the space in half with separating hyperplanes at each node. The top node, for example, might split the space in half along the x axis, with elements to the left in the left child node, and elements to the right in the right child node. Beneath, these nodes may be split along the y axis, and so on and so forth until the leaf nodes have only a few elements in them. Each node stores a bounding box that contains all the points inside it.

This is a bit different from the binary trees we used before, where each node contains a separate data point. Here, only the leaves contain data; the tree nodes organize the data and store the bounding boxes of their descendants.

k-d trees are efficient for querying nearest neighbors, finding points within a certain range, and other common operations. They work well in moderate numbers of dimensions – say, up to around 10 dimensions. In higher dimensions, they provide minimal benefits over a search over all points.

- Exercise

  At each level of a k-d tree, we have to decide which axis to split on and where to split it. What rules might we use to make the tree most efficient?

  A couple of dead-simple rules to start:

  1. Alternate which axis to split on in turn, without looking at the data at all.
  2. Split at the median of the chosen axis. (But if there's a lot of data, finding the median may be slow!)

  Suggestions for which axis?

  1. Axis with highest or lowest variance
  2. Which split reduces the in-group variance the most
  3. Choose at random

  Split point?

  1. Maximize separation between closest points in two halves
  2. Median, so there's half on each side
  3. Median of a random sample of the data

R trees

Often we need to store more than just points: we want to store arbitrary geometric shapes, like polygons and line segments. (For example, building outlines, road networks, city boundaries, rivers, and so on.) For these, a k-d tree will not suffice. There are other kinds of search trees useful for geometric data; one is R trees.

In an R tree, each node is labeled with the bounding box of the spatial features it contains. To search the tree, we traverse the tree, checking if the features we're looking for intersect the bounding box at each node; if so, we traverse the children as well. (Notice that a search may return features from multiple leaf nodes, instead of only returning a single node.)

Insertion requires clever heuristics, like inserting into the node whose bounding box has to be enlarged the least. Different heuristic choices result in different search characteristics, so different implementations use different methods in attempts to attain optimal performance.

R trees allow efficient (log time) querying of objects contained in a region, nearest objects, objects which overlap a region, and so on.

Other data structures for spatial data include GiST, or generalized search trees, a generalization of B+ trees, themselves a generalization of binary trees where nodes can have more than 2 children. GiST can be used to implement a variety of spatial data structures, including R trees; the PostGIS spatial querying system uses it extensively.

There will be an `r-tree` challenge among the projects to choose from at the end of the semester.

Traversing trees for fun and profit

We've talked about graphs before: representing graphs, traversing them, using traversal to solve common problems. Trees are, of course, a special case of graphs, and both k-d trees and R trees can be traversed like graphs. This can be useful in several ways.

For example, in code implementing a k-d tree, I have a method

```
TreeNode = namedtuple("TreeNode", ["left", "right", "split_axis", "split",
                                   "bounds", "num_points"])
LeafNode = namedtuple("LeafNode", ["bounds", "num_points"])

class KDTree:
    # constructor and other methods here
    ...

    def traverse(self, start, accum, pre_visit_fun=None, visit_fun=None,
                 post_visit_fun=None):
        """Traverse the tree, visiting each node in order from left to right.

        There are three callback functions, allowing implementations of
        preorder, inorder, and postorder traversals. pre_visit_fun is called
        before the node's children are visited; visit_fun is called after
        the left child is visited; and post_visit_fun is called after both
        have been visited. Each callback is expected to accept the node and
        an accumulator, then return the updated accumulator.
        """

        if pre_visit_fun is not None:
            accum = pre_visit_fun(start, accum)

        # Don't attempt to traverse leaves (nodes with no edges from them)
        if isinstance(start, TreeNode):
            accum = self.traverse(accum, pre_visit_fun, visit_fun,
                                  post_visit_fun, start.left)

            if visit_fun is not None:
                accum = visit_fun(start, accum)

            accum = self.traverse(accum, pre_visit_fun, visit_fun,
                                  post_visit_fun, start.right)

        if post_visit_fun is not None:
```

```
            accum = post_visit_fun(start, accum)

        return accum
```

This does an inorder traversal of the tree, using callback functions much in the same way we did for graph traversal. Normally, we don't need this: we're usually searching for a specific point or region, not trying to visit every node in the tree.

So why is this useful? Consider testing properties of a k-d tree. I generate random trees, then test properties like this one:

```
class KDTest(unittest.TestCase):
    # other tests here
    ...

    def test_no_bbox_overlap(self):
        """Check that child bounding boxes do not overlap."""
        def pre_visit_fun(node, accum):
            accum.append(node.bounds)
            return accum

        def post_visit_fun(node, accum):
            if isinstance(node, self.TreeNode):
                box1 = accum.pop()
                box2 = accum.pop()
                self.assertFalse(bbox.bboxes_overlap(box1, box2))
            return accum

        for _ in range(N):
            tree = self.rand_tree()
            tree.traverse(tree.root, [], pre_visit_fun, None, post_visit_fun)
```

As we traverse the tree, we keep a stack. When we visit a node, we put its bounding box on the stack. When we're done visiting a node, we pop the top two entries on the stack off – the bounding boxes of its children. We then check that the bounding boxes of the children do not overlap with each other, which would indicate a failure in the splitting algorithm.

- Exercise

  Another property of k-d trees is that the bounding box of a node must completely contain the bounding box of its child nodes.

  How would you write a test for this, following the example above? Sketch out the `pre_visit_fun` and `post_visit_fun` you might need.

```
class KDTest(unittest.TestCase):
    # other tests here
    ...

    def test_children_contained(self):
        """Check that child bboxes are contained in parent bboxes."""
```

```
        def pre_visit_fun(node, accum):
            accum.append(node.bbox)
            return accum

        def post_visit_fun(node, accum):
            if isinstance(node, self.TreeNode):
                bbox1 = accum.pop()
                bbox2 = accum.pop()
                self.assertTrue(bbox.contains(node.bbox, bbox1))
                self.assertTrue(bbox.contains(node.bbox, bbox2))
            return accum

        for _ in range(N):
            tree = self.rand_tree()
            tree.traverse(tree.root, [], pre_visit_fun, None, post_visit_fun)
```

Operations on high-dimensional data

Kernel density estimation

Suppose we have a bunch of points drawn iid from a multidimensional distribution. We don't know the density function of this distribution, so we'd like to estimate it.

One method is *kernel density estimation*. We take our data and place a *kernel function* on top of every data point. Summing these up in an intelligent way gives an estimated density function.

The kernel function is (usually) a symmetric nonnegative function which has a maximum at the origin, decays to zero as you move away, and integrates to 1. For example, a standard normal density function is a common choice of kernel.

The kernel density estimate (KDE) at any point x is

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{\|x_i - x\|}{h}\right),$$

where h is the bandwidth, which determines how smooth the estimate will be.

Now suppose I have a million points in six dimensions and need a density estimate at a certain spot. Some observations:

1. In high-dimensional space, most points are far away from the point we're interested in.
2. Far away points don't contribute much to the density, because the kernel decays to zero.
3. Clusters of points are basically one "big" point.

This suggests a fast algorithm to get an *approximate* KDE at a given query point. Make a k-d tree containing the data. Traverse the tree with a depth-first search. For each node we visit,

1. Calculate the upper and lower bounds on the distance between points in that node and the query point, using the bounding box of the node.
2. From these distance bounds, calculate the lower and upper bounds on the component of the kernel density estimate coming from the points in this node. That is, if we're looking at node j, we find the upper and lower bounds on

$$\hat{f}_j(x) = \sum_{i=1}^{n_j} K\left(\frac{\|x_i - x\|}{h}\right)$$

by using

$$\hat{f}_{j,\min} = n_j K\left(\frac{\text{maximum distance}}{h}\right)$$
$$\hat{f}_{j,\max} = n_j K\left(\frac{\text{minimum distance}}{h}\right)$$

3. If $\hat{f}_{j,\max} \approx 0$, drop this node and don't visit its children.
4. If $\hat{f}_{j,\max} \approx \hat{f}_{j,\min}$, add the average of the two to the estimated density and don't visit the children.

Steps 3 and 4 dramatically speed up the algorithm: instead of having to look at all n data points, we can skip entire groups of points or aggregate them into one. If the bandwidth is small compared to the range of the data, we'll save a lot of effort.

A more complicated version of this algorithm, described in the `dual-tree` challenge, uses two k-d trees and a priority queue to make it possible to calculate the density at many points at the same time.

- Exercise

  A depth-first search may not be the fastest route: ideally, you'd like to look at the most "promising" nodes (those that contribute most to the density) first.

  How could you modify the algorithm above to do this?

Nearest neighbors

Another common operation is finding the k nearest neighbors to a point. There are useful algorithms that use these neighbors:

***k*-NN classification** Suppose each point belongs to a certain group. If we have a new point and wish to know which group it likely belongs to, we find the k nearest neighbors to it (say, 10), and take the most common group among them.

***k*-NN regression** Suppose each point has a numerical value. To predict the value of a new point, take the average value of its k nearest neighbors.

But finding the nearest neighbors for any point requires looking at every other data point. If we need to know the nearest neighbors of many points, or find neighbors in an enormous dataset, it'll be slow.

We can again use a k-d tree. Suppose we have a point x for which we want the k nearest neighbors.

1. Search for x in the tree. Find the leaf node and save the closest point from that node.
2. As we did the search, recursing through children until we hit a leaf node, we kept a stack of the nodes we visited. Pop the nodes off this stack one at a time and look at their *other* child.
   - Find the minimum distance between the child's bounding box and x.
   - If the distance is shorter than the best distance found so far, recurse into this node.
   - If not, ignore this node and pop the next from the stack.

Instead of an O(n) search through all points, this takes O(log n) time to search only a small fraction of the nodes in the tree.

- Exercise

  A way to get even faster results is to look only for the *approximate* nearest neighbor: a point that's pretty close, but may not be the closest.

  Is there a straightforward modification of the algorithm above that would return an approximate nearest neighbor?

Resources

Software

- PostGIS, an extension to the PostgreSQL database system supporting spatial query
- QGIS, an open-source GIS desktop application

Packages

Spatial data:

- rgdal for reading and manipulating spatial data and sp for representing spatial objects in R. See the spatial task view on CRAN for others.
- geopy, a Python library for geocoding addresses, cities, geographic features, and so on, using external services like Google Maps or OpenStreetMap
- ggmap, a ggplot-like R system for plotting data on maps
- Cartopy, providing spatial plotting and mapping tools for Python
- proj.4, a library for transforming between reference systems and projections, with bindings in many languages

k-d trees and related algorithms:

- The FNN and RANN packages for R, for nearest neighbor calculation

Data

- OpenStreetMap, a freely accessible map dataset – like a Wikipedia for maps. Shapefile downloads are available.
- Natural Earth Data, free dataset of geographic and natural features
- Spatial Reference, a database of standard spatial reference systems
- TIGER from the Census Bureau