

Section Notes 11

CS51—Spring 2009

Week of April 27, 2009

1 Outline

1. Clustering
2. Impossible Programs
3. List of Topics

By the end of this section, you should be familiar with clustering and recognizing impossible programs.

2 Clustering

Clustering is the problem of dividing unlabeled data into groups. Because the data is usually coming from a source that doesn't give us its "true" classification, this problem is somewhat ill-specified—there isn't an obvious metric for what the "right answer" is. However, it is still a problem that we often face in the real world, so we have to solve it anyway. There are many approaches: two that we're covering in CS51 are the K-means and K-centers algorithms.

2.1 The K-means algorithm

The K-means algorithm is based on the model that the error of our clustering is the sum of the squared distances to the cluster centers.

For our purposes here, we'll assume that we know k —the number of clusters we want¹. Then, the algorithm is extremely simple:

1. Pick k points to be initial cluster centers.
2. Assign each point to the cluster whose center it's closest to.
3. Choose new cluster centers to be the centroid (average) of all the points in the cluster.
4. Repeat until no points change their cluster assignment.

Some properties of K-means:

- Each repetition improves the clusters—the new ones have smaller error. (Why this is true is not obvious, but it is true).
- Will always terminate.
- Tends to be very fast.
- Answer depends on starting points.
- Returned clustering may not be very good—the error function is not convex, so it may find a local minimum.

¹There do exist ways of picking k if it's not known, but we won't cover them here.

2.2 The K-centers algorithm

We talked about the fact that it doesn't always make sense to use the same error metrics when fitting lines to data. A similar story arises with clustering. So, let's consider the case where we don't want to use the sum of square errors from the cluster centers as our error metric. Instead, we want to say that the quality of a cluster is the distance of the *furthest* point from the cluster center. This leads to the K-centers algorithm:

1. Pick a point to be the center of the first cluster.
2. Pick the point that is furthest away from that first point to be the center of the second cluster.
3. Pick the next point that is furthest away from all the existing clusters to be the next center.
4. Repeat until you have k clusters.

This is also very simple, and turns out to get an error within a factor of 2 of the optimal choice of cluster centers when using this error metric.

Properties of K-centers:

- Also fairly fast.
- May also find local minima. The result mentioned above bounds how bad it can be.

2.3 Clustering discussion

- Random restart can help find better clusters—use multiple starting points.
- Use the error function for your algorithm to evaluate how good the result was—pick the best one.
- There is a lot of work on trying to pick starting points more intelligently than just random. It's unclear how often they end up being better than random restart. (Depends on dimensionality, amount of data, number of clusters, etc...)

3 Impossible Programs

In computer science (and in the world!), there are a number of problems that are unsolvable, no matter what kind of clever or algorithm you write to try to solve it. The example given in lecture was lossless compression - it is impossible to write a lossless compression program that always works, and always produces an output that's smaller than the input. Because it reduces the size of a file, since you can keep applying it to its output to shrink the file size to 0. It turns out that even if you allow the output size to be less than or equal to the size of the input, it's still impossible (assuming the program actually does compress at least one input). This can be shown using an argument based on the pigeon-hole principle.

The quintessential impossible program is the halting problem. The halting problem is to write a program which determines, given another program and an input to that program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily amounts of storage space before halting. The question is simply whether the given program will ever halt on a particular input. (Adapted from Wikipedia.) Using a technique called diagonalization, it can be proven that it is impossible to write a program that solves the halting problem—details in the Lecture 21 notes.

Remember reduction from Lecture 13? There, we proved that a problem X is as solvable as another problem Y by showing that a solution to X can be used to solve Y (ie, reducing Y to X). If Y is already proven to have a known characteristic (impossible to solve), and Y can be reduced to X , that would mean

that X has the same characteristic (impossible to solve). Consider this "real life" example: you know that the problem of staying awake forever (A) is impossible. You want to figure out the problem of how to make a pill that eliminates the need for sleep (S). John Scheme has told you that he has created a pill that indeed fits the criteria for S. Wow, so in figuring out how to solve S, he has managed to solve A! But, like always, you should be cautious of John Scheme's schemes - you know that A is unsolvable, so S must actually be impossible.

Reduction is a very useful way to prove impossibility: that is, if the halting problem can be reduced to P, where we solve the halting problem by solving P, then we know that P shares the proven characteristic of the halting program (impossible to solve). There is, in fact, a large class of problems that the halting problem can be reduced to; they are all impossible to solve.

Knowing how to recognize provable impossible problems is important. For example, if someone writes a program and it claims to be able to solve one of the impossible problems, you know that there must be something wrong with their code. Here are some problems that we know are impossible:

- Whether a program halts on empty input.
- Whether a program accepts or rejects empty input.
- Whether two programs are exactly equal to each other.
- Whether a stream will ever terminate
- Whether a number appears in a stream
- Whether a program will ever produce any output / some particular output
- Whether a program computes some function (basically the "exactly equal" case)
- Whether a scheme function will throw a type error ("divide by zero" to be specific)
- Whether this program that doesn't use loops or recursion halt
- Whether this program ever make a connection to the network
- Whether this program ever (try to) write outside some region in memory
- Whether this program has any bugs (not well specified here, but impossible in many instances)

In general, just about anything of the form "does this arbitrary program do X" is impossible.

3.1 So what do we do?

Your boss comes to you with a problem, and you recognize that it's not possible to solve. What to do? All is not lost! If a program turns out to be impossible to write, that just means you have to relax some assumptions. For example, you can't write a lossless compression program that always works and never returns an output larger than its input. This tells you that something has to give—either make it lossy, make it not always work, or accept that some inputs will end up with an output that's larger (could be by just one bit!) than the input.

There are two common approaches to the "Does this arbitrary program do X" problem: one is to restrict the set of allowed programs, so they are no longer arbitrary. For example, it's easy to tell whether a program that doesn't have loops or recursion halts on some input. Another, used when you want to make sure that

the program doesn't do something "bad", is to modify the program to check that it doesn't do that at runtime. For example, while we can't tell whether an arbitrary program will have a divide-by-zero error, we can insert some code just before every division to do a check.

4 Things we've learned this semester!

This is not an exhaustive list; for an exhaustive list of topics covered in CS51, revisit all lecture slides, section notes, problem sets, and projects - anything we have learned this semester is fair game!

Data Structures: Graphs, Trees, Queues, PQueues

Abstraction and Contracts

Higher-Order Functions (lambda, fold, map, filter)

Time Complexity, Big-O, and Induction

Dijkstra's Algorithm

Parsing and Parse combinators

Reductions

Side Effects and Mutation

Object-Oriented Design

Lazy-List/Streams

Macros

Line-fitting (Linear Fitting, Least Squares, Least Median Squares)

Clustering

Impossible Programs