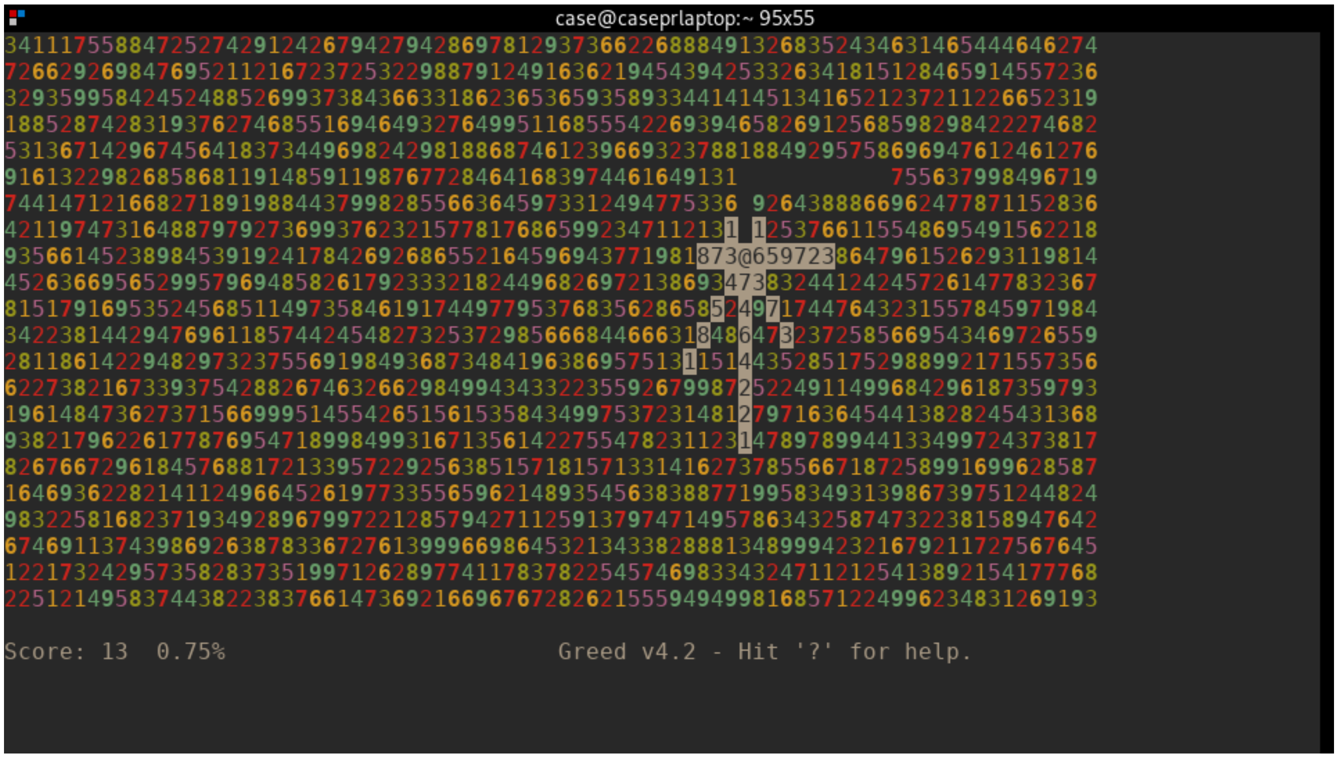
Mimicking a Brute-Force Solve of “Greed” with BigML’s Framework

Case Regan

What is greed?

Greed is a simple mathematical game played on an *n* by *n* grid of digits. A token (the ‘@’ symbol in the image) is placed on a random spot on the grid. Each turn, it can move a number of squares in any direction equal to the first number in the path it is moving on as shown in the image. Squares it passes over are blanked out, and if a player ever runs into a wall or blanked out square the game is over. When this happens, the player’s score is equal to the number of squares they were able to blank out in total.



```
case@caseprlaptop:~ 95x55
3411175588472527429124267942794286978129373662268884913268352434631465444646274
7266292698476952112167237253229887912491636219454394253326341815128465914557236
3293599584245248852699373843663318623653659358933441414513416521237211226652319
1885287428319376274685516946493276499511685554226939465826912568598298422274682
5313671429674564183734496982429818868746123966932378818849295758696947612461276
91613229826858681191485911987677284641683974461649131           755637998496719
74414712166827189198844379982855663645973312494775336 9264388866962477871152836
42119747316488797927369937623215778176865992347112131 1253766115548695491562218
93566145238984539192417842692686552164596943771981873@6597238647961526293119814
4526366956529957969485826179233321824496826972138693473832441242457261477832367
8151791695352456851149735846191744977953768356286585249717447643231557845971984
3422381442947696118574424548273253729856668446663184864732372585669543469726559
2811861422948297323755691984936873484196386957513115144352851752988992171557356
6227382167339375428826746326629849943433223559267998725224911499684296187359793
1961484736273715669995145542651561535843499753723148127971636454413828245431368
9382179622617787695471899849931671356142275547823112314789789944133499724373817
8267667296184576881721339572292563851571815713314162737855667187258991699628587
1646936228214112496645261977335565962148935456383887719958349313986739751244824
9832258168237193492896799722128579427112591379747149578634325874732238158947642
6746911374398692638783367276139996698645321343382888134899942321679211727567645
1221732429573582837351997126289774117837822545746983343247112125413892154177768
2251214958374438223837661473692166967672826215559494998168571224996234831269193

Score: 13  0.75%                        Greed v4.2 - Hit '?' for help.
```

Greed is useful for demonstrating the ineffectiveness of greedy algorithms on certain problems, hence its name. Although certain expert systems perform better than others on average, the only way to ensure that a game of greed is played perfectly is to construct a full tree of possible moves and pick the highest-scoring path. For this reason, finding anything close to an optimal solution on a board such as the one above is extremely difficult.

What is the problem I am trying to solve?

Ideally, I would aim to create a bot that can take in a large serialized greed board and use a BigML model to output a good move (that is, one that is significantly more effective than random chance). However, due to computational power and time limits, such a result is outside the scope of what I am realistically able to achieve. Because of this, I have scaled my value of *n* down to 4 and made my board contain only numbers between 1 and 3. With this smaller scope, I hope to make a bot that performs well on random boards of this size. I have seen bots use approaches like this to play similar but slightly less complex games like tic-tac-toe, and wanted to see if I could replicate those results on a more complex problem.

How did I approach this problem?
I generated a dataset using a programmatic implementation of greed that could output the best path using brute force for a given board as a series of board states. I serialized my data by essentially flattening all vital information about a board state into one row of data. Since boards are randomly generated and different for each game, a decision process approach was out of the question, meaning that each board would contain no references to boards other than itself. The representation I ended up settling on was 18 columns representing the 16 squares in the board and the 2 coordinates of the cursor, with a 19th output column corresponding to the optimal move on that board as determined via brute force. Since I could generate these programmatically, I made a 5000-board training set and a 500-board testing one.

How did I use BigML's tools?
I played around with several of the supervised learning tools I had available, and although I experimented with almost everything I knew I would likely get the best performance from a deepnet, possibly supported by other models. The strength of deepnets and similar learning models is that they can identify patterns in data using multiple layers to simulate the process of abstraction, which is what I knew I would need. Even so, I knew my problem was especially complex and a single deepnet might need support from something else. I tried different types of deepnets, some created manually and others created using BigML's optimization processes. I evaluated all of them on their own and tried the better-performing ones in conjunction with other supervised models I thought would help them find patterns in the data.

What did I conclude from my work?
Although my most successful attempt performed better than random chance, they were less effective than I would have wanted them to be. The best r^2 values I got were between 0.4 and 0.5 and the mean errors were around 0.9, which was about twice as good as a guess but nowhere near enough to outperform a human. My best model was a fusion based on an OptiML with some slight adjustments, but not far behind were the models I initially expected to perform the best: fusions between a deepnet and one other model. Even though my results were passable in terms of metrics like r^2, the bot implementing my best

model frequently made mistakes and even attempted illegal moves. From this I can conclude that even my highest-performing models were relatively far away from my goal.

What could I have done differently?
One thing I underestimated in my approach to greed was the how big the complexity difference between it and games like tic-tac-toe where this approach could have worked really was. The fact that tic-tac-toe can be modeled as a decision tree starting from an empty board is huge, as it limits the amount of scenarios a bot could encounter by many orders of magnitude. Greed, even on a 4 by 4 board, is probably closer to the complexity level of checkers than it is to tic-tac-toe. Even so, there are other approaches I would have liked to try, the main one being unsupervised learning with neural networks. Since the game is about maximizing a score, it’s easy to write an objective function that a neural network could use for measuring its performance. Projects made using this kind of deep learning have had fantastic performance even on massively complex games like go and chess, so greed should be very doable with the same tools.