Final Project

Due Exam Week (Friday Dec. 19 at noon)

The project will be done in groups of three, with students assigned randomly.

The project, when split amongst the group members, is not intended to be a huge undertaking. The goals of the project are to give you experience in working collaboratively and developing a well-designed, well-tested piece of software.

The project will be graded as a letter grade and will count for about as much as two problem sets/quizzes in your final grade.

Please use standard citation practices to cite any papers/code/online resources/chatbots whose ideas you make use of. Please do not consult with any people (in particular class members) who are not in your group. You are of course welcome to ask Chris or João questions.

You can use AI, but you should limit it to brainstorming and help with small components of the coding. In particular, you should decide on the structure (OOP vs. functional programming, what functions/methods, the arguments) yourself. You should carefully check and understand any code produced by AI assistance.

Problem

Your task is to implement and experiment with the use of genetic algorithms for variable selection. Some details on genetic algorithms are available in Section 3.4 of the Givens and Hoeting book on Computational Statistics and the baseball data discussed there are in the class GitHub repository as `project/baseball.dat`. You should also be able to find plenty of other information about genetic algorithms online. The result should be an Python package that I can easily use, as discussed below in detail. In particular, I will be testing your code on my own test cases). My grading will be largely based on the following items.

1. Your solution should follow the template in `https://github.com/paciorek/GA-dev`. With regard to the inputs and outputs, you'll need to conform to the template, but you can allow for additional input arguments and additional elements of the output dictionary. I would have liked to allow more flexibility for you to think more about the interface yourself, but for my grading, I need some degree of uniformity so I don't spend large amounts of time adapting to various interfaces. Your fitness metrics should be (10-fold) cross-validated R^2 (calculated as one minus the ratio of the summed squared prediction errors divided by the sum squared deviations around the mean of the observations.

2. Your solution should involve modular code, with functions or OOP methods that implement discrete tasks. You should have an overall design and style that is consistent across the components.

3. In terms of efficiency, the generations are inherently sequential, apart from the cross-validation. You should try to vectorize as much as possible within a generation. You can also explore parallel processing on a single machine for the cross-validation or perhaps for when working with the population in a given generation, in particular the evaluation of the fitness function. If you do explore parallelization, you must include an argument `n_workers`, with a default value of 1, that controls the amount of parallelization.

4. Show the results of using your implementation on several examples (some of these could potentially be simulated data).

5. Formal testing is required, with a set of tests where results are compared to some known truth. For testing the overall function, since the algorithm is stochastic, you'll need to think carefully about how to set this up. You should also have unit tests for individual functions that carry out the individual computations that make up the algorithm. I have an initial template test file in the `tests` directory of the `GA-dev` repository as a starting point.

6. Do not use code or packages found online except for the model fitting and other standard functionality available in Python/numpy/scipy. You can also use `scikit` utilities (such as for objective functions). If you'd like to use any other code or packages, please consult with me first.

7. Your code must pass the tests in my template repository, either when run via GitHub Actions in your repository or manually. You can also check your results with an example dataset by running `assess.py` in the `tests` directory. Note that passing the testing ensures that I am able to easily use your code and to test it on other scenarios. If I am not able to do so, that will badly hurt your grade. When I am grading the project, I will have expanded versions of `test.py` and `assess.py` that I will use as a large part of the grade. This will run your GA on some additional datasets. This is not a "bake-off", so I am not grading based on you achieving the absolute best predictor selection or R^2 results, but I do expect reasonable performance and reasonable run-times under the default settings for your implementation. However, for more intensive prediction algorithms, run-times may increase quite a bit compared to basic regression. To allow the user (in particular me) to control the run time, make sure that `pop_size` and `n_gen` are arguments to `select` that control the number of individuals in the population and the number of generations (i.e., iterations).

8. You should start your work by getting the genetic algorithm up and running with simple linear regression. Once that is done you can explore more interesting prediction methods, potentially choosing something other than regression as your default fitting method. This provides the opportunity for one or more group members to explore other methods that might be of interest to them. For some prediction methods, regularization is built in and the methods may essentially ignore some of the potential predictors. This means that the fitness score, even when based on CV, may not penalize non-sparse solutions. For such situations, the `penalty` argument should provide a value λ, with the penalized fitness being $R^2 - \lambda f$ where f is the fraction of potential predictors that are selected. In my grading I'll use this argument for some of my test cases (set to some "moderate" value that aims to produce reasonably sparse solutions when there are a limited set of predictors that really influence the outcome). The key thing is that your default implementation (default GA settings and default prediction method) do a reasonable job of

finding a good set of predictors for the penalty value I choose.

9. You can also provide arguments that allow a user to try out different GA settings or prediction methods. If you allow different prediction methods or (in the README) suggest a small number of alternative GA settings, I will try to make use of that for my test cases during grading.

Formatting requirements and additional information

1. Your solution to the problem should have two parts:

   a. A Python package named `GA`, following the placeholder package in my template repository. Your work should be done in a private repository named `GA-dev` within the Berkeley GitHub account or github.com account of one of the project members. **Make sure to share the repository with me.** Note that if you use `github.berkeley.edu`, you won't be able to automate your testing via GitHub Actions.

      The package should include:

      i. A primary function called `select` that carries out the variable selection, including appropriate code comments.

      ii. Other supporting code in the same or additional files (please think about clear organization), including appropriate code comments.

      iii. Formal tests set up using `pytest` and included in the package.

      iv. Help information for the main function, in the form of a standard Python doc string for `select`. As part of this, you should have brief working examples in the example section. You do not need extensive doc strings for your auxiliary functions but there should be a brief doc string just stating what each function does.

   b. The README for your GitHub repository should present your package and how to use it, including an overview of your approach and demos of how to use the package. It must include a paragraph describing the specific contributions of each team member and which person/people were responsible for each component of the work. Part of your grade will be based on the clarity and helpfulness of your README. You can find lots of examples of software package READMEs at the GitHub repositories for those packages. If you want to be able to generate demo results automatically, you should be able to have a qmd version of the README that you render as `quarto render README.qmd --to md` before committing `README.md`.

2. You should start the process by mapping out the modular components you need to write and how they will fit together, as well as what the primary function will do. After one person writes a component, another person on the team should test it and, with the original coder, improve it. You could also consider using pair programming for some of your development.

3. You should use Git and GitHub to manage your collaboration from the start of your work. Consider using branches and pull requests, particularly once you have a basic working package.

4. To submit your finished project, fill out this form.