HW2: Binary Search

Objectives:
To practice solving problems by modeling them as binary search problems.

Due date: Sunday, January 21th by 11:00pm.

Materials: starter code (hw2-binary-search.tar.gz)

What to turn in:

- Your final version of `TeamModeler.java` and any other Java files that you create or change.
- A PDF file containing your answers to the questions from problems 2 and 3 below.

How to submit:
Use the canvas page (https://canvas.uw.edu/courses/1124392/assignments/4062734) to submit your solution.

Overview

In this assignment, you will write code to fit a model of NFL team offensive and defensive quality so that it best matches the data from a collection of NFL games. To do so, you will implement the coordinate descent algorithm described in lecture, which solves the problem of fitting the model by using a sequence of calls to **ternary search**.

When you are done, you can use your model to simulate games between the teams using this web site (https://homes.cs.washington.edu/~kevinz/football-sim/) (link). By performing a large number of simulations, the web site can also predict the win probability for each team (and the corresponding Vegas point spread).

Provided Data

You are provided with data from the entire 2016 NFL season along with the first six weeks of the 2017 NFL season. We will use the 2016 data to determine the penalty coefficient for our model (see below) that makes the best predictions. You will then fit the model with that penalty coefficient using the 2017 data in order to make predictions for weeks seven and later of the 2017 season.

Each row in the two provided data files describes a single "drive" that took place during an NFL game. The first three columns identify the team on offense, the team on defense, and the week of the NFL season (from 1 to 17). The last two columns give the "expected points" for the positions on the field where the drive started and ended. The expected points are the historical average number of points scored by an offense in that particular situation. A drive that starts closer to the opponent's goal line will have higher expected points at the start.

NFL Model

The goal of the model is to explain the change in expected points on each drive from where it starts to where it ends. Our model will have one parameter for each team's offense and one for each team's defense. There will also be one constant term, C, included for every drive. With 32 NFL teams, that is a total of 65 parameters.

For a drive with team A on offense and team B on defense, the model tries to explain the change in expected points as:

$$\text{change} \sim \text{offense}_A - \text{defense}_B + C$$

The quality of the fit of the model is described by the loss function. We will use a loss function that is the sum of the squared errors on each drive — i.e., the square of the difference between the actual change in expected points and the amount predicted by the formula above — and an additional term that is proportional to the sum of the absolute values of the parameters. The proportionality constant on the second term can be set by the user. The larger it is, the fewer non-zero parameters will appear in the model that best fits the data. (This model is just linear regression plus an "L1 regularization" term.)

Problems

To complete the assignment, you need to fill in the bodies of the three empty methods in `TeamModeler.java`. You can do so in any order. However, we recommend the following approach.

1. Implement `findBestModel` using coordinate descent. You can start from a model with all parameters set to zero. You can stop when the change in the model, as measured by the `norm0` method applied to the *difference* between the two models, is less than the passed in `tol` parameter. On each iteration, call ternary search once for each of the 65 model parameters to find its best value with all other parameters fixed. You can do so in any order you choose, although you may find that some orders give better performance.

   Note that the provided implementation of ternary search requires you to give the range over which to search for the minimum. Hence, you will need to decide what range to use in order to call the function. In this case, it is possible to use domain knowledge to limit the range. In particular, from the description above, you can put an absolute bound on how large or small any parameter could be.

   To execute your code, you can use the `run` target of the `build.xml` file provided.

2. Implement `train` as described below and use it to figure out how sparse — specifically, how many non-zero parameters — gives the best predictions with six weeks of data.

   Your method should examine each period of `WEEKS+1` consecutive weeks in the passed-in (2016) data, not including week 17, where `WEEKS` is a static constant in the code. `WEEKS` is currently set to six, so these are **seven week periods**. You will use the first `WEEKS` weeks to fit the model using the penalty (described below) and the last week to test the fitted model produced.

   For each period, you will also try a number of different penalties. Specifically, try each penalty coefficient from 0.050 down to 0.000, stepping by 0.001 at a time. In total, there are 10 different periods (testing in weeks 7 .. 16) and 51 different penalties, for a total of 510 different period/penalty combinations to try.

You can load all of the drives during a particular sequence of weeks using the `loadDrives` method, which is provided for you. To find the best model, use the `findBestModel` method you wrote in problem 1, passing in the drives from the **first `WEEKS` weeks of the period only**. To test the model on the last week, just call `model.evalLoss(drives, 0.0)`, passing in the drives from the **last week only**. We'll call this last number the "test error". Note that the penalty is used to find the model, but *not to test it*.

To get the final numbers that you will output, we need to normalize the test errors. For each period, compute the minimum of all the test errors from that period. Subtract this from each of the test errors to get the normalized error for that period/penalty combination. (Since we are subtracting the minimum, one penalty from each period will have a normalized error of zero and every normalized error will be non-negative.)

For each period/penalty combination, you also need to record the number of non-zero parameters. (That's the quantity we're trying to optimize.) You can get that by calling `model.countNonZeroParameters(0.005)`.

Finally, print out one line of output for each period/penalty combination, showing the number of non-zero parameters and the normalized error. Use the printf format "`%2d %g`", where the "`%2d`" is filled in with the number of non-zero coefficients and the "`%g`" is filled in with the normalized error.

To execute your code, you can use the `train` target of the `build.xml` file provided.

Now answer the following questions:

a. How does the speed of coordinate descent relate to the penalty used? That is, does it seem faster or slower in certain circumstances?

   (Hint: The easiest way to figure this out is to temporarily add a line of code that prints after each model is computed. You should see a fairly clear pattern in how quickly these are printed after watching the computations for just a couple of periods.)

b. Create a scatter plot with the number of non-zero parameters on the X axis and the errors on the Y axis.

c. What number of non-zero parameters seems to give the least error?

3. Answer the following question:

   a. Suppose that you wanted to add the ability to find the model with a specific number of non-zero parameters (e.g., the number that you found was safest above) instead of the model that minimizes the loss for a particular choice of penatly. Given an algorithm to find that model.

   Extra credit: Implement your solution in the method `findBestSparseModel` of `TeamModeler`.

Provided Code

The starter code is organized into the following classes:

- **TeamModel**: Stores the 65 parameters in the model. It also includes a function `evalLoss` that will compute the loss function defined above, given a choice for the penalty coefficient and a list of drives.
- **TeamModeler**: The entry point for the program and the only file that you must change. It starts out with

an implementation of `main` and a helper method to parse the data. You will need to provide the methods that implement coordinate descent and related functionality.

- **Optimizer**: Helper class providing implementations of binary and ternary search. The latter is implemented by `findMinimumOfUnimodal` and `findMaximumOfUnimodal`.
- **Drive**: Records the information (from the data files) about a particular drive.
- **CsvParser**: Helper class for parsing the data files.
- **ArgParser**: Helper class for parsing Java command-line arguments.

The starter code also includes a `build.xml` file that includes these targets:

- **build**: Compiles the code.
- **clean**: Deletes the compiled code.
- **test**: Runs the JUnit tests. (JUnit tests for the provided code are included. Feel free to add your own JUnit tests to help make sure that your code is correct.)
- **train**: Invokes `TeamModeler.train` using the 2016 NFL data.
- **run**: Invokes `TeamModeler.findBestModel` or `TeamModeler.findBestSparseModel` with the 2017 NFL data. The latter function is invoked if the parameter `num-nonzero` is defined; otherwise, the former is invoked. With `findBestModel`, you can change the penalty constant using the `penalty` parameter, which otherwise defaults to 0.0.

If you are working from the command-line, you can invoke `findBestModel` by running the command `ant -Dpenalty=X run` , and you can invoke `findBestSparseModel` by running the command `ant -Dnum-nonzero=Y run` .