CSCI 4360/6360 Data Science II
Department of Computer Science
University of Georgia

Assignment 2: Guest Lecturer Edition

DUE: Thursday, September 14 by 11:59:59pm

Out August 31, 2017

Overview

1 Submitting

All submissions will go to **AutoLab**. You can access AutoLab at:

- https://autolab.cs.uga.edu

You can submit deliverables to the **Assignment 2** assessment that is open. When you do, you'll submit two files:

1. `assignment2.py`: the Python script that implements your algorithms, and
2. `assignment2.pdf`: the PDF write-up with any questions that were asked

These should be packaged together in a tarball; the archive can be named whatever you want when you upload it to AutoLab, but the files in the archive should be named **exactly** what is above. Deviating from this convention could result in the autograder failing!

To create the tarball archive to submit, run the following command (on a *nix machine):

```
> tar cvf assignment2.tar assignment2.py assignment2.pdf
```

This will create a new file, `assignment2.tar`, which is basically a zip file containing your Python script and PDF write-up. Upload the archive to AutoLab. There's no penalty for submitting as many times as you need to, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on September 14, so give yourself plenty of time! A late submission because the server got hammered at the deadline will *not* be acceptable (there is a *small* grace period to account for unusually high load at deadline, but I strongly recommend you avoid the problem altogether and start early).

Also, to save time while you're working on the coding portion, you are welcome to create a tarball archive of just the Python script and upload that to AutoLab. Once you get the autograder score you're looking for, you can then include the PDF in the folder, tarball everything, and upload it. AutoLab stores the entire submission history of every student on every assignment, so your autograder (code) score will be maintained and I can just use your most recent submission to get the PDF.

2 Reminders

- If you run into problems, ping the `#questions` room of the Slack chat. If you still run into problems, ask me. But please please please, **do NOT** ask Google to give you the code you seek! I will be on the lookout for this (and already know some of the most popular venues that might have solutions or partial solutions to the questions here).
- Prefabricated solutions (e.g. `scikit-learn`) are NOT allowed! You have to do the coding yourself!
- If you collaborate with anyone, just mention their names in a code comment and/or at the top of your homework writeup.

Questions

1 Linear Regression **[20pts]**

Assume we are given n training examples $(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)$, where each data point $\vec{x}_i$ has m real-valued features (i.e., $\vec{x}_i$ is m-dimensional). The goal of regression is to learn to predict y from $\vec{x}$, where each y_i is also real-valued (i.e. continuous).

The linear regression model assumes that the output Y is a linear combination of input features X plus noise terms ϵ from a given distribution, with weights on the input features given by β.

We can write this in matrix form by stacking the data points $\vec{x}_i$ as rows of a matrix X, such that x_{ij} is the j-th feature of the i-th data point. We can also write Y, β, and ϵ as column vectors, so that the matrix form of the linear regression model is:

$$Y = X\beta + \epsilon$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix}, \text{and } X = \begin{bmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_n \end{bmatrix},$$

and where $\vec{x}_i = \begin{bmatrix} x_1, x_2, ..., x_n \end{bmatrix}$.

Linear regression seeks to find the parameter vector β that provides the best fit of the above regression model. There are lots of ways to measure the goodness of fit; one criteria is to find the β that minimizes the squared-error loss function:

$$J(\beta) = \sum_{i=1}^{n} (y_i - \vec{x}_i^T \beta)^2,$$

or more simply in matrix form:

$$J(\beta) = (X\beta - Y)^T (X\beta - Y), \tag{1}$$

which can be solved directly under certain circumstances:

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{2}$$

(recall that the "hat" notation $\hat{\beta}$ is used to denote an *estimate* of a true but unknown–and possibly, unknow*able*–value)

When we throw in the ϵ error term, assuming it is drawn from independent and identically distributed ("i.i.d.") Gaussians (i.e., $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$), then the above solution is also the MLE estimate for $P(Y|X;\beta)$.

All told, then, we can make predictions $\hat{Y}$ using $\hat{\beta}$ (X could be the training set, or new data altogether):

$$\hat{Y} = X\hat{\beta} + \epsilon$$

Now, when we perform least squares regression, we make certain idealized assumptions about the vector of error terms ϵ, namely that each ϵ_i is i.i.d. according to $\mathcal{N}(0, \sigma^2)$ for some value of σ. In practice, these idealized assumptions often don't hold, and when they fail, they can outright implode. An easy example and inherent drawback of Gaussians is that they are sensitive to outliers; as a result, noise with a "heavy tail" (more weight at the ends of the distribution than your usual Gaussian) will pull your regression weights toward it and away from its optimal solution.

In cases where the noise term ϵ_i can be arbitrarily large, you have a situation where your linear regression needs to be *robust* to outliers. Robust methods start by weighting each observation *unequally*: specifically, observations that produce large residuals are down-weighted.

[15pts] In this problem, you will assume $\epsilon_1, ..., \epsilon_n$ are i.i.d. drawn from a Laplace distribution (rather than $\mathcal{N}(0, \sigma^2)$); that is, each $\epsilon_i \sim \text{Lap}(0, b)$, where $\text{Lap}(0, b) = \frac{1}{2b}\exp(-\frac{|\epsilon_i|}{b})$.

Derive the loss function $J_{\text{Lap}}(\beta)$ whose minimization is equivalent to finding the MLE of β under the above noise model.

Hint #1: Recall the critical point above about the form of the MLE; start by writing out $P(Y_i|X_i; \beta)$.

Hint #2: Logarithms nuke pesky terms with exponents without changing linear relationships.

Hint #3: Multiplying an equation by -1 will switch from "argmin" to "argmax" and vice versa.

[5pts] Why do you think the above model provides a more robust fit to data compared to the standard model assuming the noise terms are distributed as Gaussians? Be specific!

2 Regularization [30pts]

When the number of features m is much larger than the number of training examples n, or very few of the features are non-zero (as we saw in Assignment 1), the matrix X^TX is not full rank, and therefore cannot be inverted. This wasn't a problem for logistic regression which didn't have a closed-form solution anyway; for "vanilla" linear regression, however, this is a show-stopper.

Instead of minimizing our original loss function $J(\beta)$, we minimize a new loss function $J_R(\beta)$ (where the R is for "regularized" linear regression):

$$J_R(\beta) = \sum_{i=1}^{n}(Y_i - X_i^T\beta)^2 + \lambda\sum_{j=1}^{m}\beta_j^2,$$

which can be rewritten as:

$$J_R(\beta) = (X\beta - Y)^T(X\beta - Y) + \lambda||\beta||^2 \tag{3}$$

[5pts] Explain what happens as $\lambda \to 0$ and $\lambda \to \infty$ in terms of J, J_R, and β.

[15pts] Rather than viewing β as a fixed but unknown parameter (i.e. something we need to solve for), we can consider β as a random variable. In this setting, we can specify a prior distribution $P(\beta)$ on β that expresses our prior beliefs about the types of values β should take. Then, we can estimate β as:

$$\beta_{\text{MAP}} = \text{argmax}_{\beta} \prod_{i=1}^{n} P(Y_i | X_i; \beta) P(\beta), \tag{4}$$

where MAP is the *maximum a posteriori* estimate.

(aside: this is different from the MLE, which is the frequentist strategy for solving for a parameter. think of MAP as the Bayesian version.)

Show that maximizing Equation 4 can be expressed as *minimizing* Equation 3 with the assumption of a Gaussian prior on β, i.e. $P(\beta) \sim \mathcal{N}(0, I\sigma^2/\lambda)$. In other words, show that the L_2-norm regularization term in Equation 3 is effectively imposing a Gaussian prior assumption on the parameter β.

Hint #1: Start by writing out Equation 4 and filling in the probability terms.

Hint #2: Logarithms nuke pesky terms with exponents without changing linear relationships.

Hint #3: Multiplying an equation by -1 will switch from "argmin" to "argmax" and vice versa.

[10pts] What is the probabilistic interpretation of $\lambda \to 0$ under this model? What about $\lambda \to \infty$? Take note: this is asking a related but *different* question than the first part of this problem!

Hint: Consider how the prior $P(\beta)$ is affected by changing λ.

3 Computational Botany **[10pts]**

One model for estimating the economic net impact of importing a plant species is as follows:

$$ENB = [\pi * TPR * (V_L - V_T)] - [(1 - \pi) * FPR * V_T],$$

where π is the probability the proposed species is invasive, TPR and FPR are the true and false positive rates respectively, V_T is the expected benefit, and V_L is the expected losses (assuming the species is truly invasive). The result, ENB, is the *expected net benefit.*

This is a very general process for assessing the potential economic damage of plant species proposed for importation. One weakness of the approach is the assumption of independence: there is no explicit definition of time or space in the model, and despite the use of regression trees, the features are generally assumed to be independent as well (i.e., any dependences in the features is discovered as part of the learning procedure, rather than explicitly modeled *a priori*).

[10pts] Devise an approach to investigate violations of the various independence assumptions. Take care to be explicit about which independence assumption you are testing, and how you can prove (either formally or through empirical experiments) whether a violation is occurring.

4 Evolutionary Computing **[40pts]**

[5pts] Compare generational versus steady state genetic algorithms. Discuss the advantages and disadvantages of each.

[5pts] Compare Michigan versus Pittsburgh classifier systems. Discuss the advantages and disadvantages of each.

[30pts] In this part, you'll re-implement your logistic regression code from the previous assignment to use a simple genetic algorithm to learn the weights, instead of gradient descent.

Your script `assignment2.py` should accept the following required arguments:

1. a file containing training data (same as Assignment 1)
2. a file containing training labels (same as Assignment 1)
3. a file containing testing data (same as Assignment 1)

It should also be able to accept the following *optional* arguments:

- `-n`: a population size (default: 200)
- `-s`: a per-generation survival rate (default: 0.3)
- `-m`: a mutation rate (default: 0.05)
- `-g`: a maximum number of generations (default: 50)
- `-r`: a random seed (default: -1)

The handout on AutoLab contains a skeleton script with the command-line parsing ready to go. It also contains subroutines that ingest and parse out the data files into NumPy arrays. You'll use the same dataset as before: the training set for your evolutionary algorithm to learn good weights, and the testing set to evaluate the weights.

Your evolutionary algorithm for learning the weights should have a few core components:

Random population initialization. You should initialize a full array of weights *randomly* (don't use all 0s!); this counts as a single "person" in the full population. Consequently, initialize n arrays of weights randomly for your full population. You'll evaluate each of these weights arrays independently and pick the best-performing ones to carry on to the next generation.

Fitness function. This is a way of evaluating how "good" your current solution is. Fortunately, we have this already: the objective function! You can use the weights to predict the training labels (as you did during gradient descent); the fitness for a set of weights is then the *average classification accuracy.*

Reproduction. Once you've evaluated the fitness of your current population, you'll use that information to evolve the "strongest." You'll first take the top s%–the ns arrays of weights with the highest fitness scores–and set them aside as the "parents" of the next generation. Then, you'll "breed" random pairs of these parents parents to produce "children" until you have n arrays of weights again. The breeding is done by simply averaging the two sets of parent weights together.

Mutation. Each individual weight has a mutation rate of m. Once you've computed the "child" weight array from two parents, you need to determine where and how many of the elements in the child array will mutate. First, flip a coin that lands on heads (i.e., indicates mutation) with probability m (the mutation rate) for each weight w_i. Then, for each mutation, you'll generate the new w_i by sampling from a Gaussian distribution with mean and variance set to be the empirical mean and variance of *all* the w_i weights of the *previous* generation. So if W_p is the $n \times |\beta|$ matrix of the previous population of weights, then we can define $\mu_i = W_p\left[:, i\right].\text{mean}()$ and $\sigma_i^2 = W_p\left[:, i\right].\text{var}()$. Using these quantities, we can then draw our new weight $w_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$.

Generations. You'll run the fitness evaluation, reproduction, and mutation repeatedly for g generations, after which you'll take the set of weights from the final population with the highest fitness and evaluate these weights against the testing dataset.
Your script should be able to be invoked as follows:

```
> python assignment2.py train.data train.label test.data
```

with the optional parameters then able to be stated at the end. The data files (`train.data` and `test.data`) contain three numbers on each line:

```
<document_id> <word_id> <count>
```

Each row of the data files contains the count of how often a given word (identified by ID) appears in certain documents (also identified by ID). The corresponding labels for the data has only one number per row in the file: the label, 1 or 0, of the document with ID corresponding to the row of the label in the label file. For example, a 0 on the 27^{th} line of the label file means the document with ID 27 has the label 0.

After you've found your final weights and used them to make predictions on the test set, your code should print a predicted label (0 or 1) by itself on a single line, *one for each document*–this means a single line of output per unique document ID (or per line in one of the `.label` files). The output will be used to autograde your GA on AutoLab. For example, if the following `test.data` file has four unique document IDs in it, your program should print out four lines, each with a 1 or 0 on it, e.g.:

```
> python assignment2.py train.data train.label test.data
0
0
1
1
```

Evolutionary programs **will take longer** than logistic regression's gradient descent. I strongly recommend staying under a population size of 300, with no more than about 300 generations. **Make liberal use of NumPy vectorized programming** to ensure your program is running as efficiently as possible. The AutoLab autograder timeout will be extended to about 10 minutes, but you should be able to get reasonable training performance without having to go even half that long.