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

Questions

Linear Regression **[20pts]**

For this question, I watched a couple of videos on youtube about maximum liklihood estimation and sampling data points from probability distributions.

1

If the error terms ϵ_i are assumed to be i.i.d and sampled from a Gaussian distribution,

$$\epsilon \sim N(0, \sigma^2)$$
$$\epsilon = y - w^T x$$
$$y \sim N(w^T x, \sigma^2)$$

Similarly, If

$$\epsilon \sim Lap(0, b)$$
$$\epsilon = y - w^T x$$
$$y \sim Lap(w^T x, b)$$

$$P(Y|x_i;w) = \Pi_i Lap(w^T x, b)$$

$$P(Y|x_i;w) = \Pi_i \frac{1}{2b} e^{(\frac{-|y-w^T x|}{b})}$$

$$\log P(Y|x_i;w) = \sum -\log 2b + \left(\frac{-|y-w^T x|}{b}\right)$$

Now, $\log 2b$ is a constant,
So we can write $J_{Lap}(w)$ as,

$$J_{Lap}(w) = \sum \left(|y-w^T x|\right)$$

2

If the noise terms are distributed as Gaussians, the loss function is in the form of squared errors. This places a very high penalty on outliers and thus the model is highly reactive towards them. In case of a Laplacian distribution, the loss function is the mean absolute error. This is less reactive to outliers.

Regularization **[30pts]**

1

As λ decreases, the cost function penalises exploding weights to a lesser extent.
As $\lambda \to 0$ the cost function $J_R(\beta) \to J(\beta)$
As λ increases, the regularization term becomes more dominant. It starts heavily penalising all weights. As $\lambda \to \inf$, the cost function $J_R(\beta) \sim \lambda||\beta||^2$
The model will only fit the intercept.

2

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

$$\log \beta_{\text{MAP}} = \text{argmax}_\beta \sum_{i=1}^{n} (\log P(Y_i|X_i;\beta) + \log P(\beta))$$

$$\log \beta_{\text{MAP}} = \text{argmax}_\beta \sum_{i=1}^{n} \left(-(Y-W^T X)^2 + (-\log \frac{\lambda}{I\sigma^2} - \frac{\beta^2\lambda}{I\sigma^2})\right)$$

Getting rid of the constants.

$$\text{argmax}_\beta \log \beta_{\text{MAP}} = \text{argmax}_\beta \sum_{i=1}^{n} - \left((Y-W^T X)^2 + \frac{\beta^2\lambda}{I\sigma^2}\right)$$

If we assume unit variance, $\sigma^2 = 1$ and I is an identity matrix. And multiply and divide right hand side by -1 to change argmax to argmin.

$$\text{argmax}_\beta \log \beta_{\text{MAP}} = \text{argmin}_\beta \sum_{i=1}^{n} \left((Y - W^T X)^2 + \lambda \beta^2\right)$$

3

Now in this case β is a random variable with variance $\frac{I\sigma^2}{\lambda}$.
If $\lambda \to 0$ in this case, that means $Var[\beta] \to \text{inf}$, that is variance tends to infinity. So the samples will be far away.
Similarly, if $\lambda \to \text{inf}$, $Var[\beta] \to 0$ which means $p(\beta)$ will be a single point at 0 and we would essentially be fitting only the intercept.

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 $\mu_i = W_p[:, i].\text{mean}()$ and $\sigma_i^2 = W_p[:, i].\text{var}()$. 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.