CS434 Machine Learning and Data Mining – Homework 2

Linear Models for Regression and Classification

Overview and Objectives. In this homework, we are going to do some exercises about alternative losses for linear regression, practice recall and precision calculations, and implement a logistic regression model to predict whether a tumor is malignant or benign. There is substantial skeleton code provided with this assignment to take care of some of the details you already learned in the previous assignment such as cross-validation, data loading, and computing accuracies.

How to Do This Assignment.

- Each question that you need to respond to is in a blue "Task Box" with its corresponding point-value listed.
- We prefer typeset solutions (LaTeX / Word) but will accept scanned written work if it is legible. If a TA can't read your work, they can't give you credit.
- Programming should be done in Python and numpy. If you don't have Python installed, you can install it from here. This is also the link showing how to install numpy. You can also search through the internet for numpy tutorials if you haven't used it before. Google and APIs are your friends!

You are NOT allowed to...

- Use machine learning package such as `sklearn`.
- Use data analysis package such as `panda` or `seaborn`.
- Discuss low-level details or share code / solutions with other students.

Advice. Start early. There are two sections to this assignment – one involving working with math (20% of grade) and another focused more on programming (80% of the grade). Read the whole document before deciding where to start.

How to submit. Submit a zip file to Canvas. Inside, you will need to have all your working code and `hw1-report.pdf`. You will also submit test set predictions to a class Kaggle. This is required to receive credit for Q8.

1 Written Exercises: Linear Regression and Precision/Recall [5pts]

I'll take any opportunity to sneak in another probability question. It's a small one.

1.1 Least Absolute Error Regression

In lecture, we showed that the solution for least squares regression was equivalent to the maximum likelihood estimate of the weight vector of a linear model with Gaussian noise. That is to say, our probabilistic model was

$$y_i \sim \mathcal{N}(\mu = \mathbf{w}^T\mathbf{x}_i, \sigma) \quad \longrightarrow \quad P(y_i|\mathbf{x}_i, w) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y_i - \mathbf{w}^T\mathbf{x}_i)^2}{\sigma^2}} \tag{1}$$

and we showed that the MLE estimate under this model also minimized the sum-of-squared-errors (SSE):

$$\text{argmax} \underbrace{\prod_{i=1}^{N} P(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Likelihood}} = \text{argmin} \underbrace{\sum_{i=1}^{N} (y_i - \mathbf{w}^T\mathbf{x}_i)^2}_{\text{Sum of Squared Errors}} \tag{2}$$

However, we also demonstrated that least squares regression is very sensitive to outliers – large errors squared can dominate the loss. One suggestion was to instead minimize the sum of *absolute* errors.

In this first question, you'll show that changing the probabilistic model to assume Laplace error yields a least absolute error regression objective. To be more precise, we will assume the following probabilistic model for how y_i is produced given $\mathbf{x}_i$:

$$y_i \sim \texttt{Laplace}(\mu = \mathbf{w}^T\mathbf{x}_i, \sigma) \quad \longrightarrow \quad P(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{2b} e^{-\frac{|y_i - \mathbf{w}^T\mathbf{x}_i|}{\sigma}} \tag{3}$$

▶ Q1 Linear Model with Laplace Error [2pts]. Assuming the model described in Eq.3, show that the MLE for this model also minimizes the sum of absolute errors (SAE):

$$SAE(\mathbf{w}) = \sum_{i=1}^{N} |y_i - \mathbf{w}^T \mathbf{x}_i| \tag{4}$$

Note that you do *not* need to solve for an expression for the actual MLE expression for $\mathbf{w}$ to do this problem. Simply showing that the likelihood is proportional to SAE is sufficient.

1.2 Recall and Precision

y	P(y|x)	y	P(y|x)
0	0.1	0	0.55
0	0.1	1	0.7
0	0.25	1	0.8
1	0.25	0	0.85
0	0.3	1	0.9
0	0.33	1	0.9
1	0.4	1	0.95
0	0.52	1	1.0

Beyond just calculating accuracy, we discussed recall and precision as two other measure of a classifier's abilities. Remember that we defined recall and precision as in terms of true positives, false positives, true negatives, and false negatives:

$$\texttt{Recall} = \frac{\#\texttt{TruePositives}}{\#\texttt{TruePositives} + \#\texttt{FalseNegatives}} \tag{5}$$

and

$$\texttt{Precision} = \frac{\#\texttt{TruePositives}}{\#\texttt{TruePositives} + \#\texttt{FalsePositives}} \tag{6}$$

▶ Q2 Computing Recall and Precision [3pts]. To get a feeling for recall and precision, consider the set of true labels (y) and model predictions $P(y|x)$ shown in the tables above. We compute Recall and Precision as a specific threshold t – considering any point with $P(y|x) > t$ as being predicted to be the positive class (1) and $\leq t$ to be the negative class (0). Compute and report the recall and precision for thresholds t = 0, 0.2, 0.4, 0.6, 0.8, and 1.

2 Implementing Logistic Regression for Tumor Diagnosis [20pts]

In this section, we will implement a logistic regression model for predicting whether a tumor is malignant (cancerous) or benign (non-cancerous). The dataset has eight attributes – clump thickness, uniformity of cell size, uniformity of cell shape, marginal adhesion, single epithelial cell size, bland chromatin, nomral nucleoli, and mitoses – all rated between 1 and 10. You will again be submitting your predictions on the test set via the class Kaggle. You'll need to download the `train_cancer.csv` and `test_cancer_pub.csv` files from the Kaggle's data page to run the code.

2.1 Implementing Logistic Regression

Logistic Regression. Recall from lecture that the logistic regression algorithm is a binary classifier that learns a linear decision boundary. Specifically, it predicts the probability of an example $\mathbf{x} \in \mathbb{R}^d$ to be class 1 as

$$P(y_i = 1 \mid \mathbf{x}_i) = \sigma(\mathbf{w}^T \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}}, \tag{7}$$

where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector that we want to learn from data. To estimate these parameters from a dataset of n input-output pairs $D = \{\mathbf{x}_i, y_i\}$, we assumed $y_i \sim Bernoulli(\theta = \sigma(\mathbf{w}^T \mathbf{x}_i))$ and wrote the negative log-likelihood:

$$-logP(D|\mathbf{w}) = -\sum_{i=1}^{n} logP(y_i|\mathbf{x}_i, \mathbf{w}) = -\sum_{i=1}^{n} \left(y_i log\sigma(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) \right) \tag{8}$$

▶ Q3 Negative Log Likelihood [2pt]. Implement the `calculateNegativeLogLikelihood` function in `logreg.py`. The function takes in a $n \times d$ matrix X of example features (each row is an example) and a $n \times 1$ vector of labels $\mathbf{y}$. It should return the result of computing Eq.8 (which results in a scalar value). Note that np.log and np.exp will apply the log or exponential function to each element of an input matrix.

Gradient Descent. We want t find optimal weights $\mathbf{w}^* = argmin_{\mathbf{w}} - logP(D|\mathbf{w})$. However, taking the gradient of the negative log-likelihood yields the expression below which does not offer a closed-form solution.

$$\nabla_{\mathbf{w}}(-logP(D|\mathbf{w})) = \sum_{i=1}^{n}(\sigma(\mathbf{w}^T\mathbf{x}_i) - y_i)x_i \tag{9}$$

Instead, we opted to minimize $-logP(D|\mathbf{w})$ by gradient descent. We've provided pseudocode in the lecture but to review the basic procedure is written below (α is the stepsize).

1. Initialize $\mathbf{w}$ to some initial vector (all zeros,random, etc)
2. Repeat until max iterations:
   (a) $\mathbf{w} = \mathbf{w} - \alpha * \nabla_{\mathbf{w}}(-logP(D|\mathbf{w}))$

For convex functions (and sufficiently small values of the stepsize α), this will converge to the minima.

▶ Q4 Gradient Descent for Logistic Regression [5pt]. Finish implementing the `trainLogistic` function in `logreg.py`. The function takes in a $n \times d$ matrix X of example features (each row is an example) and a $n \times 1$ vector of labels $\mathbf{y}$. It returns the learned weight vector and a list containing the observed negative log-likelihood after each epoch (uses `calculateNegativeLogLikleihood`). The skeleton code is shown below.

```
def trainLogistic(X,y, max_iters=2000, step_size=0.0001):
    # Initialize our weights with zeros
    w = np.zeros( (X.shape[1],1) )
    # Keep track of losses for plotting
    losses = []

    # Take up to max_iters steps of gradient descent
    for i in range(max_iters):

        # Make a variable to store our gradient
        w_grad = np.zeros( (X.shape[1],1) )

        # Compute the gradient over the dataset and store in w_grad

        ##### TODO: Implement equation 9.

        # This is here to make sure your gradient is the right shape
        assert(w_grad.shape == (X.shape[1],1))

        # Take the update step in gradient descent
        w = w - step_size*w_grad

        # Calculate the negative log-likelihood with w
        losses.append(calculateNegativeLogLikelihood(X,y,w))

    return w, losses
```

To complete this code, you'll need to implement Eq.9 to compute the gradient of the negative log-likelihood of the dataset with respect to the weights $\mathbf{w}$. Note that an approach that loops over the dataset to compute Eq.9 takes about 15x slow than a fully matrix-form version. The `max_iters` variable can be set lower (200ish) during development to avoid this slowness being too annoying – but when optimizing performance later you may want to raise it back up. Either solution is fine for this assignment if you're patient.

If you've implemented this question correctly, running `logreg.py` should print out the learned weight vector and training accuracy. You can expect something around 86% for the train accuracy. Provide your weight vector and accuracy in your report.

2.2 Playing with Logistic Regression on This Dataset

Adding a Bias. The model we trained in the previous section did not have a constant offset (called a bias) in the model – computing $\mathbf{w}^T\mathbf{x}$ rather than $\mathbf{w}^T\mathbf{x} + b$. A simple way to include this in our model is to add an new column to X that has all ones in it. This way, the first weight in our weight vector will always be multiplied by 1 and added.

▶ Q5 Adding A Dummy Variable [1pt]. Implement the `dummyAugment` function in `logreg.py` to add a column of 1's to the left side of an input matrix and return the new matrix.

Once you've done this, running the code should produce the training accuracy for both the no-bias and this updated model. Report the new weight vector and accuracy. Did it make a meaningful difference?

Observing Training Curves. After finishing the previous question, the code now also produces a plot showing the negative log-likelihood for the bias and no-bias models over the course of training. If we change the learning rate (also called the step size), we could see significant differences in how this plot behaves – and in our accuracies.

▶ Q6 Learning Rates / Step Sizes. [2pt] Gradient descent is sensitive to the learning rate (or step size) hyperparameter and the number of iterations. Does it look like the gradient descent algorithm has converged or does it look like the negative log-likelihood could continue to drop if `max_iters` was set higher?

Different values of the step size will change the nature of the curves in the training curve plot. In the skeleton code, this is originally set to 0.0001. Change the step size to 1, 0.1, 0.01, and 0.00001. Provide the resulting training curve plots and training accuracy. Discuss any trends you observe.

Cross Validation. The code will also now print out K-fold cross validation results (mean and standard deviation of accuracy) for K = 2, 3, 4, 5, 10, 20, and 50. This part may be a bit slow, but you'll see how the mean and standard deviation change with larger K.

▶ Q7 Evaluating Cross Validation [2pt] *Come back to this after making your Kaggle submission.*

The point of cross-validation is to help us make good choices for model hyperparameters. For different values of K in K-fold cross validation, we got different estimates of the mean and standard deviation of our accuracy. How well did these means and standard deviations capture your actual performance on the leaderboard? Discuss any trends you observe.

2.3 Make Your Kaggle Submission

Great work getting here. In this section, you'll submit the predictions of your best model to the class-wide Kaggle competition. You are free to make any modification to your logistic regression algorithm to improve performance; however, it must remain logistic regression! For example, you can change feature representation, adjust the learning rate, and max_steps parameters.

▶ Q8 Kaggle Submission [8pt]. Submit a set of predictions to Kaggle that outperforms the baseline on the public leaderboard. To make a valid submission, use the train set to build your logistic regression classifier and then apply it to the test instances in `test_cancer_pub.csv` available from Kaggle's Data tab. Format your output as a two-column CSV as below:

```
id,type
0,0
1,1
2,1
3,0

.
.
.
```

where the id is just the row index in `test_cancer_pub.csv`. You may submit up to 10 times a day. In your report, tell us what modifications you made for your final submission.

Extra Credit and Bragging Rights [1.25pt Extra Credit]. The TA has made a submission to the leaderboard. Any submission outperforming the TA on the *private* leaderboard at the end of the homework period will receive 1.25 extra credit points on this assignment. Further, the top 5 ranked submissions will "win HW2" and receive bragging rights.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?