CS 240 : Lab 4
Logistic Regression and Naïve Bayes Classifiers

TAs: Onkar Borade, Ramsundar

Instructions

- This lab will be **graded**. The weightage of each question is provided in this PDF.
- Please read the problem statement and the submission guidelines carefully.
- All code fragments need to be written within the `TODO` blocks in the given Python files. Do not change any other part of the code.
- **Do not** add any *print* statements to the final submission since the submission will be evaluated automatically.
- For any doubts or questions, please contact either the TA assigned to your lab group or one of the 2 TAs involved in making the lab.
- The deadline for this lab is **Monday, 5 February, 5 PM**.
- The submissions will be checked for plagiarism, and any form of cheating will be appropriately penalized.

Starting this Lab the submissions will be on Gradescope. You need to upload the following Python files q1.py, q2.py, q3.py. In Gradescope, you can directly submit these Python files using the upload option (you can either drag and drop or upload using browse). No need to create a tar or zip file.

1 Logistic Regression [55 marks]

In statistics and machine learning, *classification* refers to a type of supervised learning. For this task, training data with known class labels are given and used to develop a classification rule for assigning new unlabeled data to one of the classes. A special case of the task is binary classification, which involves only two classes. Some examples:

- Classifying an email as spam or non-spam.
- Classifying a tumor as benign or malignant.

The algorithms that sort unlabeled data into labeled classes are called classifiers. Many advanced libraries, such as `scikit-learn`, make it possible for us to train various models on labeled training data, and predict on unlabeled test data, with a few lines of code. However, it does not give an insight into the details of what happens underneath when we run those codes. In this code, we implement a logistic regression classifier from scratch, without using any advanced libraries, to understand how it works in the context of binary classification. The basic idea is to segment the computations into pieces and write functions to compute each piece sequentially so that we can build a function based on the previously defined functions.

1.1 Problem Statement & Dataset

In particle physics, an event refers to the results just after a fundamental interaction takes place between subatomic particles, occurring in a very short time span, at a well-localized region of space. The problem is to classify an event produced in a particle accelerator as a **background** or **signal**, based on relevant feature variables.

Information on 250,000 events is included in the dataset. For each event, it has information on 31 features (2 integer-type features and 29 float-type features). Additionally, the dataset contains the object-type target variable labels and float-type variable weights. The target variable can take two possible values: b (indicating a background event) and s (indicating a signal event).

In the question that follows, you will be required to implement certain functions defined as follows.

- **Log loss:** Contrary to linear regression, which employs *squared loss*, logistic regression makes use of the *log loss* function, given by

$$\textbf{Logistic Loss:} \quad L(y, y') = -y \log(y') - (1-y) \log(1-y'),$$

  where y is the true value of a binary target (taking values 0 or 1) and y' is the prediction, which can be thought of as the predicted probability of y being 1. Observe that the loss is 0 when the true value and predicted value agree with each other, i.e., $L(0,0) = L(1,1) = 0$. On the other hand, the loss explodes towards infinity if the predicted value approaches 1 when the true value is 0, or it approaches 0 when the true value is 1. Mathematically, $\lim_{t\to 1-} L(0,t) = \lim_{t\to 0+} L(1,t) = \infty$.

- **Cost Function:** Let $\mathbf{y} = (y_1, y_2, \ldots, y_n)$ be the true values (0 or 1) and $\mathbf{y}' = (y'_1, y'_2, \ldots, y'_n)$ be the corresponding predictions (probabilities). Then, the *cost function* is given by the average loss:

$$C(\mathbf{y}, \mathbf{y}') = \frac{1}{m} \sum_{i=1}^{m} L(y_i, y'_i).$$

  An important structural distinction from the log loss function is that here the arguments $\mathbf{y}$ and $\mathbf{y}'$ (denoted `y` and `y_dash` in the supplied code) are vectors, not scalars.

- **Gradient Descent:** The gradient descent algorithm is a first order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a local maximum of that function. This procedure is then known as *gradient ascent*.

  Algorithm for Gradient Descent: In the context of minimising the cost function J, with respect to the model parameters $\mathbf{w}$ and w_0, the gradient descent algorithm is given by:

$$\text{repeat until convergence:} \quad w_j := w_j - \alpha \frac{\partial J(\mathbf{w}, w_0)}{\partial w_j}, \text{ for } j = 0, 1, 2, \ldots, n.$$

where α is the learning rate, n is the dimension of the data points, and the parameters $\mathbf{w} = (w_1, w_2, \cdots, w_n)$ and w_0 are updated simultaneously in each iteration.

Computing gradient: As derived in the class, for the binary logistic regression, the loss function turns out to be (m = number of training data points)

$$J(\mathbf{w}, w_0)) = \sum_{i=1}^{m} \left(\log(1 + \exp\{-(\mathbf{w}^\top \mathbf{x}_i + w_0)\}) + (1 - y_i)(\mathbf{w}^\top \mathbf{x}_i + w_0) \right).$$

To implement the gradient descent algorithm, we need to compute the gradients. From the equation above, we compute the partial derivatives of J with respect to w_j and w_0 as follows:

$$\frac{\partial J(\mathbf{w}, w_0)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{1 + \exp\{-(\mathbf{w}^\top \mathbf{x}_i + w_0)\}} - y_i \right) x_{i,j}, \text{ for } j = 1, 2, \ldots, n;$$

$$\frac{\partial J(\mathbf{w}, w_0)}{\partial w_0} = \frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{1 + \exp\{-(\mathbf{w}^\top \mathbf{x}_i + w_0)\}} - y_i \right).$$

1.2 Tasks to be completed

We have provided a `dataset.csv` file which consists of a sample dataset. We have also divided the dataset into train and test splits. In the file `q1.py`, you are expected to:

- Task 1: Compute **the logistic (sigmoid) function** applied to an input scalar/array. [5 marks]
- Task 2: Compute **log loss** for inputs true value (0 or 1) and predicted value (between 0 and 1). Here class 0 refers to b, i.e., background, and class 1 refers to s, i.e., signal. [10 marks]
- Task 3: Compute the **cost function**, given data and model parameters and complete the function to compute gradients of the cost function with respect to model parameters. [15 marks]
- Task 4: Update `grad_logreg(X, y, w, b)` and `grad_desc(X, y, w, b, alpha, n_iter)`.

  Implement **gradient descent algorithm** to learn and update model parameters with a pre-specified number of interactions and learning rate. [25 marks]

To run the file, simply use the command line argument: `python3 q1.py`

1.3 Testing

The following components of your code will be tested. Make sure that the variables returned by these functions (and others in general) follow the prescribed format.

- The `logistic` function
- The `log_loss` function
- The `grad_desc` function

Note, for your verification purpose, expected values of (history of weights and history of costs) for the visible dataset is given in `expected_output_q1_cost.txt` and `expected_output_q1_params.txt`.

2 Implementation of the Softmax [25 marks]

Softmax regression, also called multinomial logistic regression, extends logistic regression to multiple classes. For this problem, the training set consists of

- dataset $\{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$,
- with $\mathbf{x}^{(i)}$ being a d-dimensional vector $\mathbf{x}^{(i)} = (x_1^{(i)}, \ldots, x_d^{(i)})$,
- the label $y^{(i)}$ of $\mathbf{x}^{(i)}$ can take K possible values, e.g., with $K = 3$ classes, $y^{(i)} \in \{0, 1, 2\}$.

A **softmax regression model** has the following parameters:

- a separate real-valued weight vector $\mathbf{w}_k^\top = (w_k^{(1)}, \ldots, w_k^{(d)})$ for each class $k \in \{1, \ldots, K\}$. The weight vectors are typically stored as rows in a weight matrix $\mathbf{W}$,
- a separate real-valued bias $w_{k,0}$ for each class, the softmax function as an activation function, the cross-entropy loss function.

2.1 Code Tasks and Functions

The tasks are as follows.

1. In `predict(self, X)` function: [5 marks]

   For each class k compute a linear combination of the input features and the weight vector of class k, that is, for each training example compute a score for each class. For class k and input vector $\mathbf{x}^{(i)}$ we have:

   $$score_k(\mathbf{x}^{(i)}, \mathbf{W}) = \mathbf{w}_k^T \cdot \mathbf{x}^{(i)} + w_{k,0}$$

   where $\cdot$ is the dot product and $\mathbf{w}_k$ is the weight vector of class k. We compute the scores for all classes and training examples in parallel, using vectorization and broadcasting:

   $$\boldsymbol{score} = \mathbf{x} \cdot \mathbf{W}^T + \boldsymbol{w}_0$$

   where $\mathbf{x}$ is a matrix of shape `(# of samples, # of features)` that holds all training examples, and $\mathbf{W}$ is a matrix of shape `(# of classes, # of features)` that holds the weight vector for each class and $\boldsymbol{w}_0$ is a k-dimensional vector representing the biases for each class.

   Note 1: It is not compulsory to do this using vectorization, you can complete the code using manual loops too.
   Note 2: We have already added bias to X in the code using `add_bias()` function

2. Apply the **softmax activation function** to transform the scores into probabilities. The probability that an input vector $\mathbf{x}^{(i)}$ belongs to class k is given by [10 marks]

   $$\hat{p}_k(\mathbf{x}^{(i)}, \mathbf{W}) = \frac{\exp\left(score_k(\mathbf{x}^{(i)}, \mathbf{W})\right)}{\sum_{j=1}^{K} \exp\left(score_j(\mathbf{x}^{(i)}, \mathbf{W})\right)}.$$

   Again we can perform this step for all classes and training examples at once using vectorization. The class predicted by the model for $\mathbf{x}^{(i)}$ is then simply the class with the highest probability.

3. Compute the cost over the whole training set. We want our model to predict a high probability for the target class and a low probability for the other classes. This can be achieved using the **cross entropy loss function**: [10 marks]

   $$J(\mathbf{w}, w_0) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[y_k^{(i)} \log(\hat{p}_k^{(i)}) \right].$$

In this formula, the target labels are **one-hot encoded**, i.e., $y_k^{(i)}$ is 1 if the target class for $\mathbf{x}^{(i)}$ is k, otherwise $y_k^{(i)}$ is 0.

Note: when there are only two classes, this cost function is equivalent to the cost function of logistic regression done in the previous question.

The gradient descent algorithm (batch-wise) is already implemented on our code in the `fit_data(self, X, y, batch_size=64, lr=0.001)` function. You just have to understand it and complete the above mentioned three tasks.

2.2 Testing

The following components of your code will be tested. Make sure that the variables returned by these functions (and others in general) follow the prescribed format.

- The `predict` function
- The `softmax` function
- The `cross_entropy` function

Note, for your verification purpose, expected values of final weights for the visible dataset is given in `expected_output_q2.txt`.

3 Naïve Bayes Implementation [20 marks]

Naïve Bayes is a popular classification algorithm widely employed in machine learning tasks. Its implementation is relatively straightforward, making it accessible for beginners and efficient for large datasets. The algorithm is particularly useful for text classification, spam filtering, and sentiment analysis.

The key idea behind naïve Bayes is the *assumption of feature independence, simplifying the probability calculations*. This assumption, though "naïve" in nature, *does not hinder its effectiveness in practice*. The algorithm works by **computing the probability of a particular class given a set of features**, making it a valuable tool for various coding applications.

One notable application is in natural language processing, where naïve Bayes shines in tasks like document categorization or sentiment analysis. Its efficiency, simplicity, and ability to handle high-dimensional data make it a go-to choice for many developers diving into machine learning.

$$P(y \mid x_1, \ldots, x_d) = \frac{P(x_1, \ldots, x_d \mid y)P(y)}{P(x_1, \ldots, x_d)} := \frac{\left(\prod_{j=1}^{d} P(x_j \mid y)\right) P(y)}{P(x_1, \ldots, x_d)}. \tag{1}$$

3.1 Functions

This question has the following functions.

- `def _calc_class_prior(self):` calculates the prior probability of the classes, $P(y)$.

  The formula for calculating the prior probability of a class ($P(y)$) is straightforward:

  $$P(y) = \frac{\text{Number of instances in the training set with class } y}{\text{Total number of instances in the training set}}.$$

- `def _calc_likelihoods(self):` and `def _calc_predictor_prior(self):` calculates the likelihood table for all features.

  $$P(x_j \mid y) = \frac{\text{Number of instances of } x_j \text{ in the training set of class } y + 1}{\sum_{x \in V} \text{Number of instances of } x \text{ in the training set of class } y + |V|}.$$

  Here V is the set of distinct items in the entire training set.

- `def predict(self, X):` calculate posterior probability for each class using Bayes rule under the naïve Bayes assumption. The class with the maximum probability is the outcome of the predictio, breaking ties arbitrarily. See Equation (1) for details.

3.2 Tasks

- Implement `def _calc_class_prior(self)` to calculate the prior. [5 marks]
- Complete `def predict(self, X)` by filling the `TODO`s. [15 marks]

Expected final output can be found in `expected_output_q3(on STDOUT).txt`.