Parallel Logistic Regression

(multi-class classification) with SGD (using OpenMP)

Team No.: 30

Submitted by:

Name:	**Roll No.:**
Abhinav Anand	2018201037
Lokesh Singh Mahar	2018201049
Amit Tiwari	2018201099

Introduction

The aim of the project was to provide a parallel implementation of logistic regression for multi-class classification. We have implemented logistic regression with stochastic gradient descent (SGD) for training multi-class classifiers used in the one-versus-all strategy of the multiclass problems and the parallel training process of classifiers. OpenMp has been used for parallel code. The dataset we have used to train and test the model is wine quality data set.

We have compared the time taken for execution of sequential and parallel implementations .Also running parallel code with different number of threads has been analysed.

Logistic regression

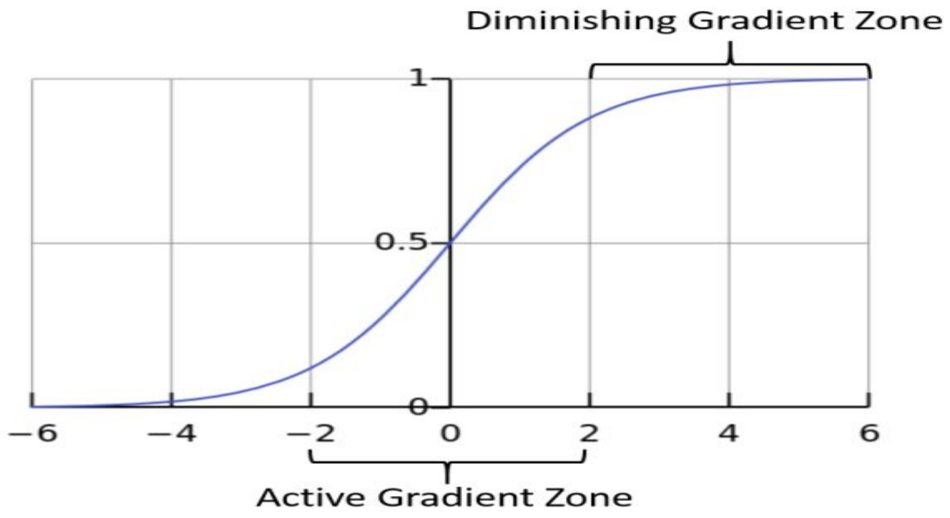
Logistic regression is the method for classification problems (problems with many possible class values).

Logistic Function (Sigmoid Function)

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. It is defined by the formula:

$$s(x) = e^x / (e^x + 1)$$

It can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.



Representation Used for Logistic Regression

Input values (x) are combined linearly using weights or coefficient values to predict an output value (y). A key difference from linear regression is that the output value being modeled is a binary values (0 or 1) rather than a numeric value. Below is an example logistic regression equation:

$$y = e^{(b0 + b1*x)} / (1 + e^{(b0 + b1*x)})$$

Where y is the predicted output, b0 is the bias or intercept term and b1 is the coefficient for the single input value (x). Each column in input data has an associated b coefficient (a constant real value) that must be learned from training data.

Stochastic gradient descent

Stochastic gradient descent is an iterative method for optimizing a differentiable objective function, a stochastic approximation of gradient descent optimization. It is called stochastic because samples are selected randomly (or shuffled) instead of as a single group or in the order they appear in the training set.

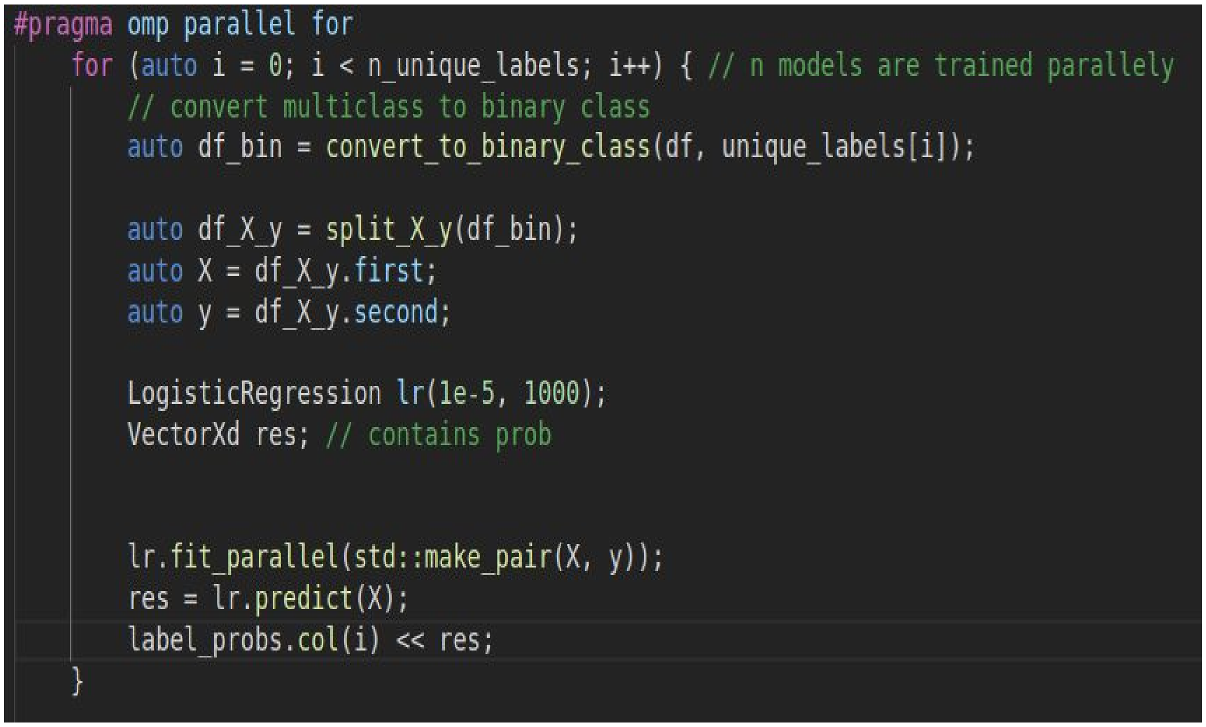
One vs All strategy

one-vs.-all strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives.

Parallel Algorithm

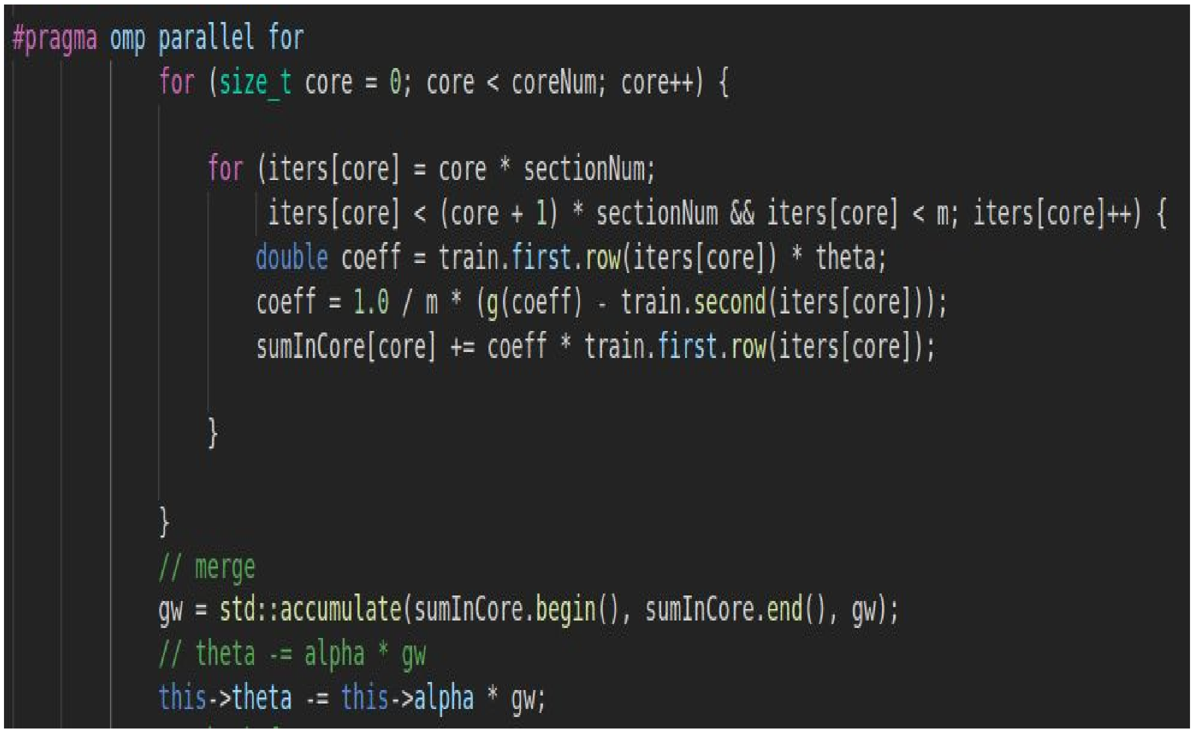
We have used nested parallelization in our implementation. Two of the areas where code has been parallelized are:

1. There are n models to be trained where n is the number of unique classes in the data set. Each of these model classifies the dataset as belonging to a particular class or not. Training of each of these models is independent of each other. So this for loop has been parallelized.



```
#pragma omp parallel for
    for (auto i = 0; i < n_unique_labels; i++) { // n models are trained parallely
        // convert multiclass to binary class
        auto df_bin = convert_to_binary_class(df, unique_labels[i]);

        auto df_X_y = split_X_y(df_bin);
        auto X = df_X_y.first;
        auto y = df_X_y.second;

        LogisticRegression lr(1e-5, 1000);
        VectorXd res; // contains prob


        lr.fit_parallel(std::make_pair(X, y));
        res = lr.predict(X);
        label_probs.col(i) << res;
    }
```

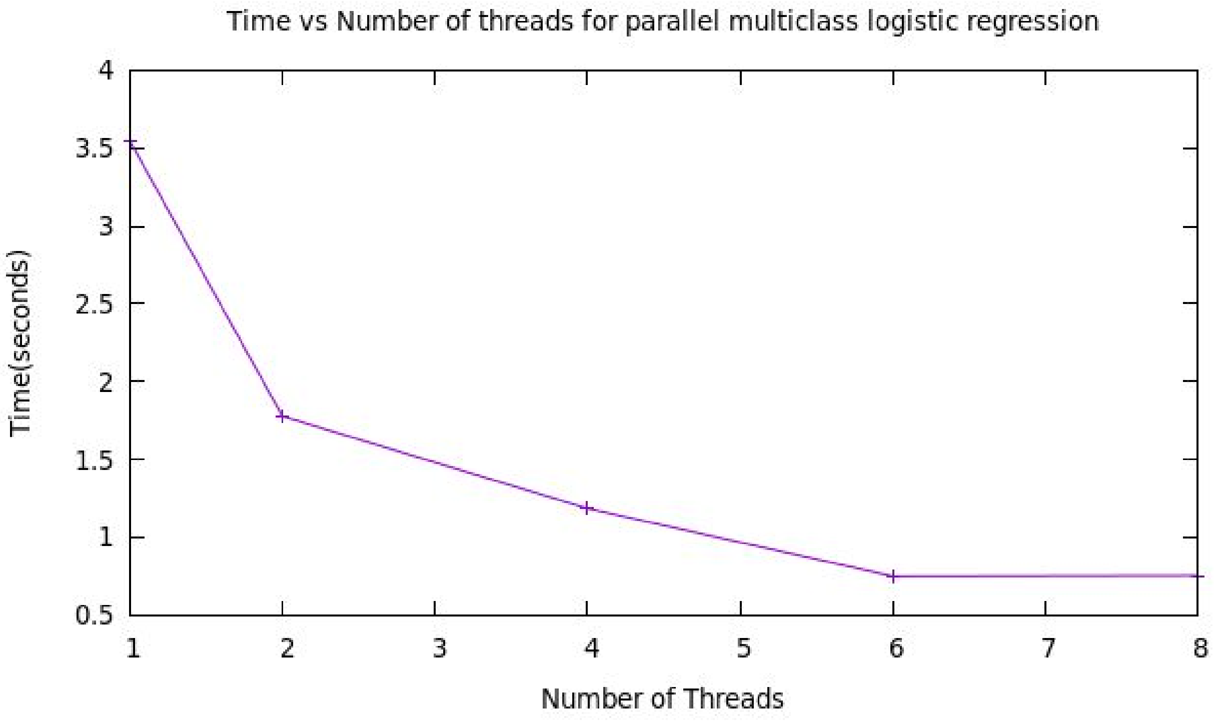
2. Stochastic gradient descent considers every row of dataset independently in each iteration. So this for loop has also been parallelized.



```
#pragma omp parallel for
            for (size_t core = 0; core < coreNum; core++) {

                for (iters[core] = core * sectionNum;
                     iters[core] < (core + 1) * sectionNum && iters[core] < m; iters[core]++) {
                    double coeff = train.first.row(iters[core]) * theta;
                    coeff = 1.0 / m * (g(coeff) - train.second(iters[core]));
                    sumInCore[core] += coeff * train.first.row(iters[core]);

                }

            }
            // merge
            gw = std::accumulate(sumInCore.begin(), sumInCore.end(), gw);
            // theta -= alpha * gw
            this->theta -= this->alpha * gw;
```

Results

Accuracy of around 44% has been obtained on given data set using both sequential and parallel implementations. The following plot shows the time for execution using different number of threads.



Conclusion

For lower number of threads, parallel implementation takes more time due to overhead involved in creation of threads. This can be observed from the fact that speedup is less than 1. But, as number of threads are increased expected behaviour is observed as parallel implementation executes faster than sequential. But with high number of threads speedup obtained saturates due to overhead involved with thread handling.

References

1. https://en.wikipedia.org/wiki/Logistic_regression
2. https://en.wikipedia.org/wiki/Sigmoid_function
3. https://machinelearningmastery.com/logistic-regression-for-machine-learnin
   g/
4. https://nanxiao.me/en/the-pitfalls-of-using-openmp-parallel-for-loops/
5. https://www.viva64.com/en/a/0054/