Machine Learning: HW2 Report

October 28, 2016

Professor Hung-Yi Lee

電機三 B03901018 楊程皓

1. Logistic regression function.

使用$z = b + \vec{w} \cdot \vec{x}$, x 取資料的57 項feature，故 w 亦有 57 項。加上常數項 b, 總體 logistic regression function 需要train 58 項參數。

使用adagrad 並每輪iteration randomize 更新w[i] 的順序，防止bias 發生，並在更新b, w 時以linear time 更新z 提高效率。此方式會讓每次training 的結果皆有所不同。

logistic regression code:

```
void Table::logisticRegression(const double& eta, double& b, double* const w, const double& deltaStop) const {
    // b for constant term, w for linear terms, z for quadratic terms.
    // lambda is 0, in this case, since not using regularization
    // G_b_t, G_w_t, G_z_t for adagrad
  double preError, error = 1, gradient_b = 0, G_b_t = 0;
  std::vector<int> order;
  for (int i = 0, length = numCols - 1; i < length; ++i)
     order.push_back(i);

  double   *z = new double[numTrains],
        *gradient_w = new double[numCols - 1],
        *G_w_t = new double[numCols - 1];

  for (int i = 0, length = numCols - 1; i < length; ++i)
     G_w_t[i] = 0;

  int counter = 0, idle = 0;
    // trains are array of training data, numTrains = 4001 in this case
  for (int i = 0; i < numTrains; ++i)
     z[i] = trains[i].linear_z(b, w);
    // iteration begins
  while (true) {
      // randomize stochastic order
     std::random_shuffle(order.begin(), order.end());

     ++counter;
     preError = error, error = 0;
        // recalculate z, redundant actually
     for (int i = 0; i < numTrains; ++i)
        z[i] = trains[i].linear_z(b, w);

     gradient_b = 0;
     for (int i = 0; i < numTrains; ++i)
```

```
        gradient_b += trains[i].gradient(func_sigma(z[i]), -1);
     gradient_b /= numTrains;
     // adagrad update b
     G_b_t += square(gradient_b);

     double prev_b = b;
     b -= eta * gradient_b / sqrt(G_b_t);
     // linear time update z
     for (int i = 0; i < numTrains; ++i)
        trains[i].update_z(z[i], -1, prev_b, b);

        // stochastic regression
     for (int i = 0, length = order.size(); i < length; ++i) {
        int index = order[i];
        gradient_w[index] = 0;

        for (int j = 0; j < numTrains; ++j)
           gradient_w[index] += trains[j].gradient(func_sigma(z[j]), index);

        gradient_w[index] /= numTrains;
        // G_w_t[index] /= 2;
        G_w_t[index] += square(gradient_w[index]);
            // adagrad update w[index]
        double prev_w_index = w[index];
        if (G_w_t[index] != 0)
           w[index] -= eta * gradient_w[index] / sqrt(G_w_t[index]);
        else
           w[index] -= eta * gradient_w[index];
        // linear time update z
        for (int j = 0; j < numTrains; ++j) {
           trains[j].update_z(z[j], index, prev_w_index, w[index]);
        }
     }

     for (int i = 0; i < numTrains; ++i)
        error += trains[i].cross_entropy(func_sigma(z[i]));

     error /= numTrains;
     cout << counter << ": " << error << endl;

     if (preError - error < deltaStop)
        ++idle;
     else
        idle = 0;
        // if iteration over 100k times or improvement too small for five times, end training
     if (counter == 100000 || idle == 5)
        break;
  }
}
```

2. Method 2.

使用 neural network regression 作為第二種方法。

實作部份為一層 layer, 第一層 layer node 數為30, layer 的推算方式為前一個layer(對第一個 layer 使用原57個 features 作為參數) 做 logistic regression, 即大部分為上述的 code, 只有為效率考量只做最多

200 iteration.

使用前一個 layer 做 logistic regression 的 code 雖有異於上面的 code, 但相似度極大, 只有使用 features 的不同，故沒有放上程式碼。

```
  void Table::neuralNetworkRegression(const int& layer, const int* const numOfNodes
    , const double& eta, double& b, double* const w, const double& deltaStop,
    const char* const outputModel_fileName) const {
    // layer is numOfLayer, in this case, 1
    // numOfNodes is num of nodes for each layer
   fstream fout;
   fout.open(outputModel_fileName, ios::out);

   // output layer and numOfNodes to model
   fout << layer << endl;
   for (int i = 0; i < layer; ++i)
      fout << numOfNodes[i] << ' ';
   fout << '\n';

   double _b;
   double* _w;
   for (int layerIndex = 0; layerIndex < layer; ++layerIndex) {
      cout << "layerIndex: " << layerIndex << endl;
      if (layerIndex == 0) {
         const int nodes = numCols - 1;
         _w = new double[nodes];

         // clear layer data
         for (int i = 0; i < numTrains; ++i)
            trains[i].clear();
             // times = num of nodes of next layer
         const int times = numOfNodes[layerIndex];
         for (int i = 0; i < times; ++i) {
            cout << "i: " << i << endl;
            _b = b;
            for (int i = 0; i < numCols - 1; ++i)
               _w[i] = w[i];
                 // do logistic regression
            logisticRegression(eta, _b, _w, deltaStop);
            // update next layer by the result of logistic regression
            for (int j = 0; j < numTrains; ++j) {
               double pred = func_sigma(trains[j].linear_z(_b, _w));
               pred -= 0.5; // shift the prediction
               trains[j].push_back(pred);
            }
            // output the result of logistic regression to model
            fout << _b << ' ';
            for (int i = 0; i < numCols - 1; ++i)
               fout << _w[i] << ' ';
            fout << '\n';
         }
         fout << '\n';
         delete[] _w;
      } else {
          // initialize _b and _w for logistic regression
         _b = 0;
         const int nodes = numOfNodes[layerIndex - 1];
         double temp_w = 1.0 / nodes;
         _w = new double[nodes];
         for (int i = 0; i < nodes; ++i)
```

```
                _w[i] = temp_w;

            for (int i = 0; i < numTrains; ++i)
                trains[i].clear();
            const int times = numOfNodes[layerIndex];
            for (int i = 0; i < times; ++i) {
                // do logistic regression, but by previous layer
                layer_logisticRegression(nodes, eta, _b, _w, deltaStop);
                for (int j = 0; j < numTrains; ++j) {
                    double pred = func_sigma(trains[j].layer_linear_z(nodes, _b, _w));
                    pred -= 0.5; // shift the prediction
                    trains[j].push_back(pred);
                }
                // output the result of logistic regression to model
                fout << _b << ' ';
                for (int i = 0; i < numCols - 1; ++i)
                    fout << _w[i] << ' ';
                fout << '\n';
            }
            fout << '\n';
            delete[] _w;
        }
    }
    // calculate the final prediction of label with last layer
    _b = 0;
    const int nodes = numOfNodes[layer - 1];
    double temp_w = 1.0 / nodes;
    _w = new double[nodes];
    for (int i = 0; i < nodes; ++i)
        _w[i] = temp_w;

    layer_logisticRegression(nodes, eta, _b, _w, deltaStop);
    delete[] _w;

    // output the result of logistic regression to model
    fout << _b;
    for (int i = 0; i < nodes; ++i)
        fout << _w[i] << ' ';

    fout.close();
}
```

3. Discussion.

這次作業因為我使用 random order 做 stochastic gradient descient, 每次的結果不一樣, 而選到的 private set 剛好沒過 baseline... 並 neural network 因時間較趕, 還有些小 bug 跟參數未調整, 故沒有放上 kaggle 做競賽。

在 model far from target 時, logistic regression 的斜率因比 linear regression 大，model 進步幅度比 linear 快很多，故實作時 iteration 輪數約500 輪內就會收斂; linear regression 則須2000輪才會收斂。而因為此次題目所要的答案是 discrete (0 or 1), sigmoid function 可以在model close to target 時斜率比 linear 更平滑，使小幅度的改動 w 在z 於兩端區的影響並不大，使 gradient 可以在其他z 於中間地帶的影響力突顯出來，使 model 優化能朝著優化 ambiguous training set 的方向。

然而此種 model 在處理本類題目時不一定是個好方式，畢竟 spam email 的判定方式(姑且假設人的判斷方式是100% 正確, 即是否是 spam 由人定義) 可能比較像 decision tree 或分類判定，某些 feature 在某些情況下是不會看的，而 logistic regression 和 linear regression 對每個 individual email 的每個

feature 的加權都是一樣，這樣的 model 雖然做起來比較簡單, 卻並不一定符合現實情況。
Neural network 與 random forest 可能是解這類問題比較好的方式，如 pokemon 進化後 CP 值的公式也是需要 classification 才會比較準, 沒有分類直接做 model 的結果往往會有不少 noise 與 error.