Solutions to problem set 11

Learning

Problem 1. Learning parameters of the Bayesian belief network

Assume the Bayesian belief network for the diagnosis of the engine problem in the figure below.

All variables in the network are binary with true, false values.

Assume we have collected the data from different car repair shops in the neighborhood and we want to learn the parameters of the network. The data are given as vectors of values for every variable in the network. The data for variables are given in the following order: (Battery, Radio, Lights, Ignition, Gas, EngineStarts).

The values are encoded using binary values: 1 for True and 0 for False. Your dataset is as follows:
Part a. Compute the ML estimates of the following probability parameters of the BBN network:

- $P(\text{Gas})$
- $P(\text{Lights}|\text{Battery} = T)$
- $P(\text{Lights}|\text{Battery} = F)$
- $P(\text{Ignition}|\text{Battery} = T)$
- $P(\text{EngineStarts}|\text{Ignition} = T, \text{Gas} = T)$
- $P(\text{EngineStarts}|\text{Ignition} = T, \text{Gas} = F)$

Solutions:

- $P(\text{Gas}): (14/20, 6/20)$
- $P(\text{Lights}|\text{Battery} = T): (12/17, 5/17)$
- $P(\text{Lights}|\text{Battery} = F): (1/3, 2/3)$
• \( P(Ignition|Battery = T) : (\frac{13}{17}, \frac{4}{17}) ; \)

• \( P(EngineStarts|Ignition = T, Gas = T) : (\frac{8}{10}, \frac{2}{10}) ; \)

• \( P(EngineStarts|Ignition = T, Gas = F) : (\frac{3}{4}, \frac{1}{4}) \).

Problem 2. Classification of handwritten digits with a one-layer neural network

In this problem we use a logistic regression model to learn and to discriminate between the two digits (3 and 5) that can appear on the input. The input consists of a two dimensional array of 1s and 0s representing the grid of pixels and their colors (black and white), e.g.

```
#    ######
 #    ##
 ##    #
####    
 #    ##
 #    ##
```

Our objective is to train the neural network to classify digits 3 and 5 from the set of handwritten digit patterns and their labels. The data are divided into two sets:

• training set used for training the model (files \textit{digit}\_x.dat, \textit{digit}\_y.dat) ;

• testing set used for testing the performance of the learned model (files \textit{digit}\_x\_test.dat and \textit{digit}\_y\_test.dat).

The data are available on-line on the class web page. The data in files are organized in rows. In file \textit{digit}\_x.dat rows represents the binary description of either a digit 3 or 5. File \textit{digit}\_y.dat reflects its corresponding label, 1 for 3 and 0 for 5. Test files \textit{digit}\_x\_test.dat and \textit{digit}\_y\_test.dat have the same structure. There are 100 sample digits in the training files, and 400 in the testing files.

To simplify the data access we have prepared the following C/C++ functions for you (included in file \textit{main}\_11.c):

• \textit{init}\_digits() which initializes the digit system by loading digits them into DIGIT structures and initializing random number generator. Datasets are represented as arrays of DIGITS and are stored in \textit{testing}\_set and \textit{training}\_set arrays.
- `get_random_training_digit()` returns a random training example from the training data set. It is returned in the DIGIT structure.
- `get_random_testing_digit()` returns a random example from the testing data set.
- `get_nth_training_digit(int nth)` returns nth training example from the training data set. There are 100 samples in the training set.
- `get_nth_testing_digit(int nth)` returns nth example from the testing data set. There are 400 samples in the testing set.
- `print_digit(DIGIT)` which prints a digit as seen above.

**Part a.** In the logistic regression model we express the distribution of outputs \( p(y = 1|x, w) \) as:

\[
p(y = 1|x, w) = \frac{1}{1 + \exp\left(-\left[w_0 + \sum_{j=1}^{k} w_j x_j\right]\right)},
\]

where exp stands for the exponential function, and \( w \) is a vector of weights \( w_0, w_1, \cdots, w_{64} \), such that \( w_0 \) corresponds to the bias weight and \( w_1, \cdots, w_{64} \) to the weights for digit pixels. \( x_j \) is the j-th component of the input vector \( x \).

Write a C/C++ function for computing the probability of \( y = 1 \) given the set of weights \( w \) and the input vector \( x \), \( p(y = 1|x, w) \). Definitions in file `main10.h` give you appropriate structures.

**Answer:**

```c
double compute_probability(double weights[], double x[])
{
    double sum_res=0;
    for (int j=0; j < 65; j++)
        {sum_res=sum_res+ weights[j]*x[j];}
    /* now do the logistic function */
    return (((double)1.0)/(((double)1.0)+exp(- sum_res)));
}
```

**Part b.** In the logistic regression we want to maximize the likelihood of the predictions. The optimization of the likelihood can be performed using the online algorithm with gradient-based updates. The \((i + 1)\)th on-line update of a weight \( w_j \) with a data sample \(<x, y>\), is

\[
w_j^{(i+1)} = w_j^{(i)} + \alpha(i + 1) \cdot \left[ y - p(y = 1|x, w^{(i)}) \right] \cdot x_j,
\]
where $\alpha(i + 1)$ is the learning rate that changes with the update step, $x^{(j)}$ is the $j$th component of the input vector $x$.

Implement the online gradient-based procedure: online_gradient_descent(int iterations) for updating the weight parameters of the logistic regression model. Your program should start from zero weights (all weights $w$ set to 0 at the beginning). To update weights use the annealed learning rate $\alpha(i) = \frac{1}{2\sqrt{i}}$ where $i$ indexes the $i$th update step. Repeat the update for 1500 steps.

To learn the weights use digits from the training set only. In every step pick samples randomly using function get_random_training_digit().

Answer:

```c
void online_gradient_descent(int iterations)
{
    double weights_old[65];
    double weights_new[65];
    double learning_rate = 1;/*note this is not same as zero on beginning*/
    DIGIT current_digit;

    printf("The training set was 100 samples, iterations were %d\n", iterations);
    reset_weights(weights_old);
    reset_weights(weights_new);
    for(int i=0; i < iterations; i++, learning_rate++)
    {
        current_digit = get_random_training_digit();
        probability = compute_probability(weights_old, current_digit.x);
        // calculate the new set of weights
        for(int k=0; k < 65; k++)
        {
            weights_new[k] = weights_old[k] +
                (((double) 1.0 )/( 2* sqrt(learning_rate)))*
                (current_digit.y - probability)*current_digit.x[k];
        }
        //copy the new weights to old weights
        copy_weights(weights_new, weights_old);
    }
    /* copy the resulting weights to the global var weights_opt */
    copy_weights(weights_new,weights_opt);
}
```
/* reset weights */
void reset_weights(double *weights)
{
    for(int i = 0; i < 65; i++)
    {
        weights[i] = 0;
    }
}

/* copy weights*/
/* weights are assumed to be stored in a double array */
/* weights w_0 (bias) w[1:64] - weights for digit inputs */
void copy_weights(double weights_from[], double weights_to[]) {
    for(int l=0; l < 65; l++)
    {
        weights_to[l] = weights_from[l];
    }
}

Part c. The model (weights) learned in part b can be used to classify digits. The output label \(\{0, 1\}\) for an input \(\mathbf{x}\) can be obtained using the following simple rule:
If \(p(y = 1|\mathbf{x}, \mathbf{w}) \geq 0.5\) then output 1
else output 0.

Write C/C++ function int classify(DIGIT digit) for classifying input digits.

Answer:

int classify(DIGIT digit) {
    double probability = 0;
    probability = compute_probability(weights_opt, digit.x);
    if(probability >= 0.5)
    {
        return 1;
    }
    else
return 0;
}

**Part d.** In this part we are interested in analyzing the quality of the learned logistic regression model. To test the quality we use average misclassification error. The average misclassification error for the dataset with $N$ samples is defined as:

$$ Error_{AMC} = \frac{\# \text{ of misclassified digits}}{N}. $$

Write C/C++ function to compute the average misclassification error of the model for both the training and testing datasets. Use functions `get_nth_training_digit(int nth)` and `get_nth_testing_digit(int nth)` to access the data. The dataset sizes are given by constants `NUM_TEST_DIGS` 400 and `NUM_TRAIN_DIGS` 100.

**Answer:**

```c
void compute_errors()
{
    int err_test = 0;
    for(int i = 0; i < NUM_TEST_DIGS; i++)
    {
        if(classify(testing_set[i]) == testing_set[i].y)
        {
        }
        else
        {
            err_test++;
        }
    }
    printf("In TEST SET, Error is: %d out of %d samples\n", err_test, 400);
    printf("In TEST SET, Average error is: %lf \n", (double)err_test/(double)400);
    err_test = 0;
    for(int i = 0; i < NUM_TRAIN_DIGS; i++)
    {
        if(classify(training_set[i]) == training_set[i].y)
        {
        }
        else
        {
            err_test++;
        }
    }
    ...
```
printf("In TRAINING SET, Error is: %d out of %d samples\n", err_test, 100);
printf("In TRAINING SET, Average error is: %lf \n", (double)err_test/(double)100);
}

Part e. Write and submit main11.c program that (1) learns the weights based on the
training data set (as described above) and (2) tests the quality of the resulting model
according to part d. Run the program multiple times and output (1) the weights learned
and (2) average training and testing misclassification error. Remark: Although the resulting
models can differ because of the random choice of the training examples you should be able
to see models with the average misclassification error for the testing data below 0.15 most
of the time.

/* Part e. */
/* main function that runs the online learning and tests for errors */

int main()
{
    init_digits();
    print_digit(training_set[5]);
    print_digit(training_set[57]);
    online_gradient_descent(1500);
    compute_errors();
    return 0;
}

Part f. Give mean misclassification errors for both the training and test data in your
report. Which misclassification error is higher? Training or testing? Can you explain it?

Answer: The misclassification error is higher on the testing samples. Good answers are in
range of 0.01-0.02 for the training set, and 0.10 for the testing set.

Part g. Optional (Extra credit). You may want to experiment a bit with number
of updates and learning rate schedules. For example, try to learn using only 100, 200,
500 updates before returning the weights. In terms of learning schedule you can try e.g.
$\alpha(i) = \frac{1}{i}$. Report results of your experiments and any hypothesis or conjectures about the
performances.

Answer: The figures (with a little try) I got are:
(100 iterations: 0.07 training and 0.1575)
(200 iterations: 0.05 training and 0.1175)
(500 iterations: 0.04 training and 0.18)
The results suggest that more iterations allows us to fit better the training set which makes sense since all examples used to adapt the weights were selected from this set. On the other hand, we see higher and varying errors for the test set. This suggests a higher variance of our weight estimates. This is due to two facts: smaller number of samples in the training set and larger learning rates applied in updates.

With $\alpha(i) = 1/i$ the error results were 0.04 for training and 0.205 for testing sets.

This result suggest that the sequence of learning rates goes down too fast leading to the following two problems: (1) the weights converge at much slower rate. (2) the weights for a few initial examples affect the result the most.

To overcome these problems one would have to run more examples. In general finding of good learning rates and an appropriate number of iterations is a highly exploratory endeavour.