NAIVE BAYESIAN LEARNING

YOUYUN ZHANG

The goal of the program is to implement a naive Bayesian classifier that classifies images of handwritten digits. This is useful in applications like reading zip codes for post offices or reading phone numbers or ids in surveys.

1. Data

The database used in this program is the MNIST database (yann.lecun.com/exdb/mnist), which has 60,000 training images and their corresponding labels; and 10,000 testing images and their corresponding labels. The MNIST database is in the 4 files located in the foler mnist_data/:

train-images-idx3-ubyte.gz: training set images (9912422 bytes)
train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

File 1 contains the 60,000 images of the training set, each is a 28×28 grayscale image of a handwritten digit and its correct integer label (0-9). Each of the $28 \times 28 = 784$ pixels in one image is a 8-bit color channel, representing a number between 0 and 255. To reduce the complexity of this project, the data set will be binarized, i.e., a pixel with a value lower than 128 will be set to value 0 and a pixel with a value equals to or higher than 128 will be set to value 1.

This project is provided with skeleton code in the src/ directory in class to parse the MNIST data files: mnist_reader_common.hpp, mnist_reader_less.hpp, mnist_reader.hpp, and mnist_utils.hpp.

Following are the functions that I need to program the classifier:

```
Code:
#define MNIST_DATA_DIR "../mnist_data"
    //Read in the data set from the files
    mnist::MNIST_dataset<std::vector, std::vector<uint8_t>, uint8_t> dataset =
    mnist::read_dataset<std::vector, std::vector, uint8_t, uint8_t>(MNIST_DATA_DIR);
    //Binarize the data set (so that pixels have values of either 0 or 1)
    mnist::binarize_dataset(dataset);
```

The first two lines of the above code load the MNIST database; the third line binarizes the training and test images, so that each pixel value in each image will be either a 0 (representing black) or 1 (representing white).

```
Code:
    // get training images
    std::vector<std::vector<unsigned char>> trainImages = dataset.training_images;
    // get training labels
    std::vector<unsigned char> trainLabels = dataset.training_labels;
```

The variable trainImages is a vector of 60,000 vectors of unsigned chars, each representing a training image. For each $i \in 0, ..., 59, 999, trainImages[i]$ is a vector of 784 unsigned chars, each representing one of the 784 pixel values of training image i (linearized in row major order). The variable trainLabels is a vector of 60,0000 unsigned chars. For each $i \in 0, ..., 59, 999, trainImages[i]$ is an unsigned char (a single byte) indicating the true digit represented by $trainingImages[i]$. To use these values, they should first be converted to integers (or uint8_ t) as follows:

```
Code:
    // get pixel value j of training image i (0 for black, 1 for white)
    int pixelValue = static_cast<int>(trainingImages[i][j]);
    //get label of training image i
    int label = static_cast<int>trainingLables[i];
```

Testing images and their labels can be extracted similarly.

2. Algorithm

The naive Bayesian model build for this program considers each pixel individually as a binary feature (0 for a black pixel and 1 for a white pixel). Therefore for all images, feature F_j is a random variable representing the value of pixel j, $\forall j \in \{0, 1, ..., 783\}$. The class variable, C, will have 10 possible values, one for each possible digit 0-9.

In the bynetwork.h, first we construct a class(C++) $BysCl$ that takes an input of number of class values($numLabels$) and number of features($numFeatures$). Within $BysCl$, we create variable nl to store $numLabels$ and nf to store $numFeatures$, then a few vectors for various use later.

```
Code:
    class BysCl{
    public:
    BysCl(int numLabels, int numFeatures){
        nl=numLabels;
        nf=numFeatures;
        for (int i=0;i<nl;i++){
            priorP_p.push_back(0);
            priorP_ct.push_back(0);
            vector<double> v;//row
            for (int i=0;i<nf;i++){
                v.push_back(0);}
            pixelP_ct.push_back(v);pixelP_pl.push_back(v);}}
    vector<vector<double>> GetPixelP(){
        return pixelP_pl;}
    vector<double> GetPriorP(){
        return priorP_p;}

    private:
     int nl; int nf;
     std::vector<double> priorP_p;
     std::vector<double> priorP_ct;//count
     vector<vector<double>> pixelP_ct;
     vector<vector<double>> pixelP_pl;
```

We first learn the prior probability of each class $c \in C, P(c)$:

$$P(C = c) = \frac{\text{number of images of label c}}{\text{total number of images}}$$

```
Code:
    //calc prior p
    for (int i=0;i<lb.size();i++){
        priorP_ct[static_cast<int>(lb[i])]++;
        priorP_p[static_cast<int>(lb[i])]+=1/((double)lb.size());}
```

Note that $\sum_{c=0}^{9} P(c) = 1$.

Now we learn the conditional probability of each feature given its class. That is, $\forall c \in C$, we learn $P(F_j|c)$, where $P(F_j|c)$ is a distribution over the possible values that F_j can take on (we take 1 in this case) given that the class(label) is c.

$$P(F_j = 1|C = c) = \frac{numberofLabelcWherePixelF_jisWhite + 1}{\text{total number of images of label c} + 2}$$

Note that Laplace smoothing is used here to prevent the classifier to predict the conditional probability to be 0 when there is no white occurs in that pixel in all images.

```
Code:
    //calc conditional p for pixels
    for (int i=0;i<ti.size();i++){//for every image
        for (int j=0;j<nf;j++){//for every feature
            if (ti[i][j]==1){//when white
            pixelP_ct[static_cast<int>(lb[i])][j]++;}}}

    for (int i=0;i<nl;i++){//grid nfxnl
        for (int j=0;j<nf;j++){
            pixelP_pl[i][j]=((double)pixelP_ct[i][j]+1)/((double)priorP_ct[i]+2);}}
```

Once we have learned these probabilities from the training set, we will evaluate our model on the test images. Let T_i represent test image i and let $t_{i,j}$ denote the value (0 or 1) of pixel j in test image i. For each test image i and let $t_{i,j}$, we use our model to compute $P(C|F_0 = t_{i,0}, F_1 = t_{i,1}, ..., F_{783} = t_{i,783})$. Let c_i^* represent the correct class(label) of test image T_i. If $c_i^* = \hat{c}_i$, our model has successfully classified image T_i. The accuracy of our model on the test set is:

$$\frac{\sum_{i=0}^{9,999} \mathbb{1}(c_i^* = \hat{c}_i)}{10,000}$$

Note that $\mathbb{1}$ is the indicator function, which returns 1 when $c_i^* = \hat{c}_i$ and 0 otherwise. So the accuracy of our model is simply the ration of successful classifications to total test images.

Specifically, for each test image T_i, compute the probability that it belongs to each class $c \in \{0, 1, ...9\}$:

$$P(C = c|F_0 = t_{i,0}, F_1 = t_{i,1}, ...F_{783} = t_{i,783})$$
$$= \frac{P(F_0 = t_{i,0}, F_1 = t_{i,1}, ..., F_{783} = t_{i,783}|C = c) \times P(C = c)}{P(F_0 = t_{i,0}, F_1 = t_{i,1}, ...F_783 = t_{i,783})}$$

Due to the naive Bayesian assumption that every variable is independent of each other, we have:

$$P(C = c|F_0 = t_{i,0}, F_1 = t_{i,1}, ...F_{783} = t_{i,783})$$
$$= \frac{(\prod_{j\in\{0,1,...,783\}} P(F_j = t_{i,j}|C = c)) \times P(C = c)}{P(F_0 = t_{i,0}, F_1 = t_{i,1}, ...F_783 = t_{i,783})}$$

The test image T_i is classified as belonging to the class c with the highest posterior probability $\hat{c_i} = argmax_c P(C = c|F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$. The term $P(F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$ can be dropped from calculation without changing the classification.

Since the intermediate probabilities can be quite small, we use the summation of log probabilities instead of the multiplication of probabilities in order to avoid underflow. In other words, instead of calculating $\prod_i P_i$, we calculate $\sum_i logP_i$. Therefore, we classify image T_i as belonging to the class $\hat{c_i}$ that maximizes:

$$argmax_c(\sum_{j\in\{0,1,...,783\}} logP(F_j = t_{i,j}|C = c)) + logP(C = c)$$

```
Code:
    int classify(std::vector<unsigned char> t){//helper function to classify
        int maxc=0;
        double max;
        double sump=0;

        for (int j=0;j<nf;j++){
            if (static_cast<int>(t[j])==1){
                sump+= log (pixelP_pl[0][j]);}
            else {
                sump+= log (1 - pixelP_pl[0][j]);}}

        max=sump + log (priorP_p[0]);
        for (int i=1;i<nl;i++){
            sump=0;
            for (int j=0;j<nf;j++){
                if (static_cast<int>(t[j])==1){
                    sump+= log (pixelP_pl[i][j]);}
                else {
                    sump+= log (1 - pixelP_pl[i][j]);}}
            sump+= log (priorP_p[i]);
            if (max<=sump){
                max=sump;
                maxc=i;}}
        return maxc;//return the class label with largest probability}

    vector<vector<int>> elv(vector<vector<unsigned char>> ti, vector<unsigned char> tl){
    //evaluate test images(ti) and test labels(tl)
        std::vector<vector<int> > mt;
        for (int i=0;i<nl;i++){
            vector<int> v;//row
            for (int i=0;i<nl;i++){
                v.push_back(0);}
            mt.push_back(v);}
        for (int i=0;i<ti.size();i++){
            int k=classify(ti[i]);
            mt[static_cast<int>(tl[i])][k]++;}
        return mt;}
```

3. Evaluations and Output

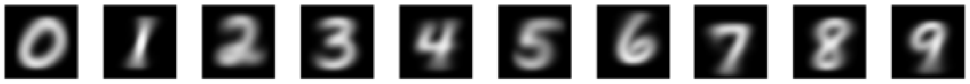
For the first part of the evaluation, we use the provided Bitmap class(bitmap.cpp and bitmap.hpp) to output images representing the conditional probabilities that each feature takes on value 1 given class $C = c(i.e., P(F_j = 1|C = c))$. Following code is used for visualizing the naive Bayesian model that we have learned:

```
Code:
    //There are ten possible digits 0-9 (classes)
    int numLabels = 10;
    //There are 784 features (one per pixel in a 28x28 image)
    int numFeatures = 784;
    BysCl byscl(numLabels,numFeatures);
    byscl.calcP(trainLabels, trainImages);

    vector<vector<double>> pixelP = byscl.GetPixelP();
    std::vector<double> priorP = byscl.GetPriorP();

    for (int c=0; c<numLabels;c++){
        std::vector<unsigned char> classFs(numFeatures);
        for (int f=0; f<numFeatures;f++){
            double p=pixelP[c][f];//assign to the variable p the probability that
                                    the pixel at location f is white given that
                                    the class is c after learning those probabilities
                                    from the training set
            uint8_t v=255*p;
            classFs[f]=(unsigned char) v;}
        std::stringstream ss;
        ss<<"../output/digit"<<c<<".bmp"<<endl;
        Bitmap::writeBitmap(classFs, 28,28,ss.str(),false);}
```

This writes 10 bitmap files to the output directory, which looks like:



The program also output two other files:

- network.txt, which lists the values of all conditional probabilities parameterizing the network. The first 784 lines are $P(F_j = 1|C = 0)$; the next 784 lines are $P(F_j = 1|C = 1)$, etc. The final 10 lines are the prior probabilities of each class 0 to 9.

- classification-summary.txt, which contains a 10×10 matrix of integers. The integer in row r and column c is the number of images in the test set of digit r which our model predicted was an image of digit c. The final line of the file is the final accuracy of this naive Bayesian model (number correctly classified/10,000) on the test set of 10,000 images.

  Output:
  ```
  890 0 4 5 2 37 20 1 21 0
  0 1083 9 3 0 12 5 0 23 0
  17 11 848 34 20 3 28 14 54 3
  ```

```
4 14 35 850 0 13 8 16 49 21
1 7 4 0 805 2 20 2 18 123
22 10 5 126 23 632 19 10 25 20
15 16 14 2 12 32 861 0 5 1
1 29 16 6 16 0 0 865 27 68
16 26 12 74 14 20 12 6 759 35
10 13 6 8 68 7 0 24 23 850
accuracy: 84.43%
```

Code:

```
    ofstream nw("../output/network.txt");
    if (nw.is_open()){
        for (int c=0; c<numLabels;c++){
            for (int f=0; f<numFeatures;f++){
                nw << pixelP[c][f] <<endl;}}
            for (int i=0;i<numLabels;i++){
                nw<<priorP[i]<<endl;}
        nw.close();}
    else cout<<"Error in opening file"<<endl;
    std::vector<vector<int> > mt = byscl.elv(testImages,testLabels);

    ofstream cs("../output/classification-summary.txt");
    int cc=0;//correct classification
    if (cs.is_open()){
        for (int c=0; c<mt.size();c++){
            for (int f=0; f<mt[c].size();f++){
                cs << mt[c][f] <<" ";
                if (c==f){
                    cc+=mt[c][f];}}
        cs<<endl;
        }
        double acu=(double)cc/(double)testImages.size();
        cs<<"accuracy: "<<acu*100<<"\%"<<endl;
        cs.close();}
    else cout<<"Error in opening file"<<endl;
```

4. Discussion

Note that there are some ”fuzziness” around the bitmap files the program outputted. This might be the result of binarizing data. The reduction of information produces the images in low resolution.

A important assumption in naive Bayesian model is that it assumes every feature is independent, which in this case is certainly not true, for that naturally neighbor pixels are related to each other because the strokes are connected in handwritten scripts. Nevertheless, a naive Bayesian model gives a relatively high accuracy, 84.84%. And we can observe from the output of the classification-summary.txt that correct predictions (i.e., the numbers on the diagonal) obviously outnumber the incorrect ones.

The reason that naive Bayesian model works so well even if the features are correlated might be that the correlation between the features are not as strong as we thought. Human brains have very good pattern recognition ability, therefore even if we consider the pixels must be very related each other, in the real computation world, these relations may just be very small. Another possible explanation is that in the same feature context the weight of the dependent features maybe similar or may cancel each other, stated by Harry Zhang in his paper.

5. Reference

USC CSCI 360 Artificial Intelligence class contents and assignments. Stuart J. Russell. Peter Norvig. "Artificial Intelligence: A modern Approach. Third Edition." Harry Zhang. "The Optimality of Naive Bayes."