Term project

110550039 劉詠

Implementation Details

Data preprocessing

First, since there are lost data in the trainWithLabel.csv and testWithoutLabel.csv , I use float number “-1.0” to fill in the blanks of both .csv file, the implementation is written in function read_csv_file() in helper.cpp, like the below shows:

```
vector<float> row;
if (!line.empty()&&line[line.size()-2]==',') {    //因為test的資料最後一行可能會有空白，所以要先
    line.insert(line.size() - 1, "-1.0");         //插入數字以免之後出現stof abort()的錯誤
}
stringstream ss(line);
string cell;
while (getline(ss, cell, ',')) {                  //以逗號作為分隔進行讀取
    if (cell.empty()) {                           //如果那一格是空白就插入數字 -1.0
        row.push_back(-1.0);
    } else {
        row.push_back(stof(cell));                //如果那一格不是空白就將內容轉為float並存起來
    }
}
```

Note that since I had modified “helper.cpp”, I also attach the modified “helper.cpp” in my homework **.zip** file.

When writting the class of Preprocessor, I only deal with categorical and numeric data for the reason that the data files seem did not contain ordinal data.

In function _preprocess_categorical(), since from feature 18 to 77, I view them as categorical data, when there is a “-1.0” in the lattice (which means it is a blank in original data file), I will use the mode (眾數) of that column to replace “-1.0”, the implementation is like following:

```
int zerocount = 0;
int onecount = 0;
for (float& val : column) {
    if(val == 0){
        zerocount++;
    }
    if(val == 1){
        onecount++;
    }
}
for (float& val : column) {
    if(val == -1){
        val = (onecount>=zerocount)? 1 : 0 ;
    }
}
```

And in function _preprocess_numeric(), I first calculate the mean of each column, and then, use that mean to fill with the data with “-1.0”, and since I want the scale of data can be in the range of [0,1], I also normalize the numeric data using range normalization, the implementation is like the below:

```
    std::vector<float> column = get_column(i);
    if(!validation){
        float mean = calculate_mean(column);
        for (float& val : column) {
            if(val == -1){
                val = mean;
            }
        }
        mean0[i] = mean;
        vector<float> temp;
        for(int i=0;i<column.size();i++){
            temp.push_back(column[i]);
        }
        std::sort(temp.begin(),temp.end());
        min0[i] = temp[0];
        max0[i] = temp[temp.size()-1];
        for (float& val : column) {
            val = (val-temp[0])/(temp[temp.size()-1]-temp[0]);
        }
    }
```

In the above code, the get_column() function is used to get the specific column that I want to deal with, and set_column() function is used to set the data after preprocessing back to the original data set that is going to be return. The above is the whole of my implementation of data preprocessing.

Naïve Bayes

$$Class_{pred} = argmax_{c_i \in C} P(C = c_i | f_1 = v_1, f_2 = v_2, \dots, f_d = v_d) =$$
$$argmax_{c_i \in C} P(f_1 = v_1, f_2 = v_2, \dots, f_d = v_d | C = c_i) \cdot P(C = c_i) =$$
$$argmax_{c_i \in C} P(C = c_i) \cdot \prod_{i=1}^{d} P(f_i = v_i | C = c_i)$$

In the above formula, we know that if we want to using naïve bayes, we need to first calculate the probability of each possible class as the following shows:

```
// Count the number of samples for each class
for (const auto& label : y) {
    int cls = static_cast<int>(label[0]);  //y contains 0 or 1
    if (cls == 0) count0++;
    else count1++;
}

// Compute prior probabilities
priorProb0 = static_cast<float>(count0) / totalSamples;        //計算每種class的機率
priorProb1 = static_cast<float>(count1) / totalSamples;
```

Next, I need to get every conditional probability of each feature in the above formula. I use two vectors of unorder map “featureProbabilities0” and “featureProbabilities1” for storing the conditional probabilities in the future.

In fit() function:

```
// Calculate conditional probabilities
for (int i = 0; i < X.size(); ++i) {
    int cls = static_cast<int>(y[i][0]);

    for (int j = 0; j < numFeatures; ++j) {
        if (cls == 0) {
            featureProbabilities0[j][X[i][j]]++;//第 j feature 出現 X[i][j] 的次數+1
        } else {
            featureProbabilities1[j][X[i][j]]++;//第 j feature 出現 X[i][j] 的次數+1
        }
    }
}
// Normalize probabilities
normalizeProbabilities(featureProbabilities0, count0);
normalizeProbabilities(featureProbabilities1, count1);
```

as the above code shows, according to the class that instance belongs to, either featureProbabilities0[j][X[i][j]] or featureProbabilities1[j][X[i][j]] will be increased of value 1, meaning that the value X[i][j] appears once in feature j. After recording the number of occurrences of different values in one specific column, we need to calculate the conditional probabilities by using functions of normalizeProbabilities().

```
void normalizeProbabilities(vector<unordered_map<float, float>>& probabilities, int count) {
    int i =0;
    for (auto& probMap : probabilities) {
        for (auto& [value, countValue] : probMap) { //value是那個feature的一個值，countValue是value出現過幾次
            if(i>16){
                countValue = countValue/count;
            }else{
                countValue = (countValue + 3) / (count + probMap.size() * 3);  // Using a small epsilon
            }
        }
        i++;
    }
}
```

As the above code shows, since feature 1 to feature 17 are numeric data, the conditional probabilities may be too small, so I use a small epsilon to replace the probabilities, and from feature 18 to 77, since they only contain either 1 or 0, I just calculate their own proportions to be the conditional probabilities.

```
vector<vector<float>> predict(vector<vector<float>> &X) override {
    // Implement the prediction logic for Naive Bayes classifier
    vector<vector<float>> predictions;

    int count=0;//
    for (const auto& instance : X) {
        float posteriorProb0 = log(priorProb0) + computeLogConditionalProb(instance, featureProbabilities0);
        float posteriorProb1 = log(priorProb1) + computeLogConditionalProb(instance, featureProbabilities1);

        int predictedClass = (posteriorProb1 > posteriorProb0) ? 1 : 0;
        predictions.push_back({static_cast<float>(predictedClass)});
    }

    return predictions;
}
```

```
float computeLogConditionalProb(const vector<float>& instance, const vector<unordered_map<float, float>>& featureProbabilities) {
    float logProb = 0.0f;

    for (int j = 0; j < instance.size(); ++j) {
        auto it = featureProbabilities[j].find(instance[j]);

        if (it != featureProbabilities[j].end()) {
            logProb += log(it->second);
        } else {
            // Handle cases where the value is not present in training data
            logProb += log(1e-3f);  // Using a small epsilon

        }
    }
    return logProb;
}
```

After knowing each conditional probability, we are going to have test data be predicted. In function predict(), it calls another function computeLogConditionalProb(),in computeLogConditionalProb(), it first check whether the value of each feature of the current instance is in the vector of featureProbabilies, if yes, add that probability to logProb (note that the reason that I convert the probability into log values is to prevent 數字越乘越小的問題), if not, add a small value to logProb. When logProb is return back to function predict(), compare two posterior probabilities, we will get the predict class of that instance.

```
vector<unordered_map<int, float>> predict_proba(vector<vector<float>> &X) {
    // Implement probability estimation for Naive Bayes classifier
    vector<unordered_map<int, float>> probabilities;

    for (const auto& instance : X) {
        float posteriorProb0 = log(priorProb0) + computeLogConditionalProb(instance, featureProbabilities0);
        float posteriorProb1 = log(priorProb1) + computeLogConditionalProb(instance, featureProbabilities1);

        float prob0 = exp(posteriorProb0);
        float prob1 = exp(posteriorProb1);

        probabilities.push_back({{0, prob0}, {1, prob1}});
    }

    return probabilities;
}
```

In function predict_proba(), first convert the two posterior probabilities back to the understandable probabilities using exp() function and then save them into “probabilties”, which record the probability of each class that the instance may belongs to(儲存 instance 屬於每個 class 的機率).

The above is the whole implementation of Naïve Bayes classifier.

KNN

```
vector<vector<float>> predict(vector<vector<float>> &X) override {
    // Implement the prediction logic for KNN
    std::vector<std::vector<float>> predictions;
    for (const auto& sample : X) {
        auto prediction = _predict_single_sample(sample);
        predictions.push_back({prediction});
    }
    return predictions;
}
```

```
float _predict_single_sample(const std::vector<float>& sample) {
    std::vector<std::pair<float, float>> distances_labels;  // Pair of (distance, label)

    // calculate distances between the test sample and all training datas
    for (size_t i = 0; i < X_train.size(); ++i) {
        float dist = _euclidean_distance(sample, X_train[i]);
        distances_labels.push_back({dist, y_train[i][0]});
    }
    // sort based on distances and consider only the closest k datas
    std::sort(distances_labels.begin(), distances_labels.end(), [](const auto& a, const auto& b) {
        return a.first < b.first;
    });
    // take a vote among the k nearest neighbors
    std::unordered_map<float, int> label_count;
    for (int i = 0; i < k; ++i) {
        label_count[distances_labels[i].second]++;
    }
    // Find the majority class
    float max_count = 0;
    float predicted_label = -1;
    for (const auto& [label, count] : label_count) {
        if (count > max_count) {
            max_count = count;
            predicted_label = label;
        }
    }
    return predicted_label;

}
```

Since KNN classifier uses nearest k neighbors to predict the test data, predict() function calls _predict_single_sample() function, in the above code, I first calculate each distance using Euclidean distance between a test sample and all the training samples, then push the distances into a vector called “distances_labels”, which also record that neighbor’s class. After that, sort the distances_labels with “dist”, then we can easily get the nearest k neighbors. Then, use an unorder map (label_count) to record the count of each class in those k neighbors. Finally, return the label with the maximum count as the predict target value. The vector “predictions” contains all predict target values of all test samples.

```
vector<unordered_map<int, float>> predict_proba(vector<vector<float>> &X) {
    // Implement probability estimation for KNN
    std::vector<std::unordered_map<int, float>> probabilities;
    for (const auto& sample : X) {
        std::vector<std::pair<float, float>> distances_labels;  // pair of (distance, label)
        // calculate distances between the test sample and all training datas
        for (size_t i = 0; i < X_train.size(); ++i) {
            float dist = _euclidean_distance(sample, X_train[i]);
            distances_labels.push_back({dist, y_train[i][0]});
        }
        // sort based on distances and consider only the closest k datas
        std::sort(distances_labels.begin(), distances_labels.end(), [](const auto& a, const auto& b) {
            return a.first < b.first;
        });
        // take a vote among the k nearest neighbors
        std::unordered_map<float, int> label_count;
        for (int i = 0; i < k; ++i) {
            label_count[distances_labels[i].second]++;
        }
        // calculate probabilities
        std::unordered_map<int, float> class_probabilities;
        for (const auto& [label, count] : label_count) {
            class_probabilities[static_cast<int>(label)] = static_cast<float>(count) / k;  // convert count to probability
        }
        probabilities.push_back(class_probabilities);
    }

    return probabilities;
}
```

In the part of predict_proba(), it is similar as function _predict_single_sample(), the difference is that I store the probability of each class of that sample may belongs to into a vector “probabilities”.

The above is the whole of the implementation of KNN classifier.

MLP

```
void fit(vector<vector<float>>& X, vector<vector<float>>& y) override {
    for (int epoch = 0; epoch < epochs; ++epoch) {
        //learning_rate = (20*learning_rate)/(20+epoch);
        for(size_t i = 0; i<X.size(); i++){
            vector<vector<float>> input = {X[i]};
            vector<vector<float>> target = {y[i]};
            _forward_propagation(input);
            _backward_propagation(target);
        }
    }
}
```

In MLP classifier, I first make each instance in training data to do forward propagation and backward propagation to get proper weights and layer values.

```
void _forward_propagation(vector<vector<float>>& X) {
    //only a single sample is passed in X.
    input_layer = X;

    // hidden Layer
    hidden_layer.resize(1, vector<float>(hidden_size)); // resizing for one sample
    for (int j = 0; j < hidden_size; ++j) {
        float sum = 0.0;
        for (size_t k = 0; k < input_size; ++k) {
            sum += X[0][k] * weights_input_hidden[k][j];
        }
        hidden_layer[0][j] = sigmoid(sum);
    }
    // output Layer
    output_layer.resize(1, vector<float>(output_size)); // resizing for one sample
    for (int j = 0; j < output_size; ++j) {
        float sum = 0.0;
        for (int k = 0; k < hidden_size; ++k) {
            sum += hidden_layer[0][k] * weights_hidden_output[k][j];
        }
        output_layer[0][j] = sigmoid(sum); // as output_layer[j];
    }

}
```

In forward propagation, from input layer to hidden layer and from hidden layer to output layer, I need to calculate each "**net**", after calculate each net value, use sigmoid function to get each node output value in hidden layer and output layer.

```
void _backward_propagation(vector<vector<float>>& target) {
    // Assuming only a single target value for the output layer
    float output_error = (target[0][0] - output_layer[0][0]) * output_layer[0][0] * (1 - output_layer[0][0]);

    // Error for the hidden layer
    vector<float> hidden_error(hidden_size, 0.0);
    for (int j = 0; j < hidden_size; ++j) {
        float error = output_error * weights_hidden_output[j][0];
        hidden_error[j] = error * hidden_layer[0][j] * (1 - hidden_layer[0][j]);
    }

    // Update weights between hidden and output layers
    for (int i = 0; i < hidden_size; ++i) {
        float gradient = hidden_layer[0][i] * output_error;
        weights_hidden_output[i][0] += learning_rate * gradient;
    }

    // Update weights between input and hidden layers
    for (int i = 0; i < input_size; ++i) {
        for (int j = 0; j < hidden_size; ++j) {
            float gradient = input_layer[0][i] * hidden_error[j];
            weights_input_hidden[i][j] += learning_rate * gradient;
        }
    }

}
```

$$\delta_j = o_j \cdot (1 - o_j) \sum_{m \in Downstream(j)} \delta_m \cdot w_{mj}$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = -\eta \cdot \frac{\partial E_d}{\partial net_j} \cdot x_{ji} = \eta \cdot \delta_j \cdot x_{ji}$$

In function _backward_propagation(), the first **for loop** which computes error of hidden layer is using the first formula as the above shows, since I only care the single target value, I ignore the summation, which means that oj * (1 – oj) corresponds to hidden_layer[0][j] * (1 - hidden_layer[0][j]) and $\delta m \cdot wm$j corresponds to "error" in function _backward_propagation(). As the second and third **for loop** in _backward_propagation(), they are used for updating weights as the computation of the second formula shows.

```
vector<vector<float>> predict(vector<vector<float>>& X) override {
    vector<vector<float>> predictions;
    for (size_t i = 0; i < X.size(); i++) {
        vector<vector<float>> input = {X[i]};
        _forward_propagation(input);
        vector<float> sample_predictions;
        for (size_t j = 0; j < output_layer.size(); ++j) {
            sample_predictions.push_back((output_layer[j][0] > 0.5) ? 1.0 : 0.0);
        }
        predictions.push_back(sample_predictions);
    }
    return predictions;
}
```

After we get proper weights and layer output values, we can use predict() function to predict the test data. Again, make each instance of test data to do forward propagation, then after getting the value in the output layer, I use a step function with threshold of 0.5 to get target class of that instance for reason that we are dealing with a classification problem, if not using a step function, the value in the output layer is a numeric number, not a type of a class.

```
vector<unordered_map<int, float>> predict_proba(vector<vector<float>>& X) {
    vector<unordered_map<int, float>> probabilities;
    for (size_t i = 0; i < X.size(); i++) {
        vector<vector<float>> input = {X[i]};
        _forward_propagation(input);

        unordered_map<int, float> prob_map;

        // a binary classification task
        float prob_positive_class = output_layer[0][0];
        float prob_negative_class = 1.0 - prob_positive_class;

        prob_map[1] = prob_positive_class; // mapping class 1 to its probability
        prob_map[0] = prob_negative_class; // mapping class 0 to its probability

        probabilities.push_back(prob_map); // store the probabilities for this sample
    }
    return probabilities;
}
```

Finally, in predict_prob() function, since it is a two classes classification problem, the network-based learning algorithm will see the value in the output layer as the

probability of the target class being positive. Then after getting the probabilities of positive class (target is 1) and negative class (target is 0), store then into the vector “probabilities” and return.

The above is my whole implementation of MLP classifier.

Discussion

1. **Implementation challenges:**

   For Naïve bayes classifier, I think it is the most understandable one in the three classifiers, but as we had known, it is based on the assumption that the features are independent with each other, in reality, not all the dataset can fit this property. And, when I was writing the Naïve bayes classifier, it has a problem that it may suffer from the issue with too small or even zero probability, if there is a very large dataset with too many zero probabilities, and if we replace all of them with a small epsilon, I will think the classifier is not reliable.

   For KNN classifier, it is also a more understandable classifier, but it will have a problem that if there is a dataset with large amount of data instances, the computation of calculating distances between a node to the others may be much more complex. Also, how to decide a better k value for KNN classifier is another problem, is a small k better (may be too overfitting) or large k better (may be too underfitting)?

   For Multilayer perceptron classifier, honest to say, it is really difficult for me to understand it. How to decide layers with neurons is the most brain-burning problem, and the computation is also too complex. As for training time, it is much slower than the other two classifier, so if there is a larger dataset, it may cause time-consuming problem.

2. **Prediction result:**

model	accuracy	f1	precision	recall	mcc	auc
3	0.75	0.470588	0.444444	0.5	0.308607	0.683036
3	0.916667	0.769231	0.833333	0.714286	0.72193	0.857143
3	0.75	0.571429	0.6	0.545455	0.396418	0.770909
3	0.833333	0.727273	0.727273	0.727273	0.607273	0.843636
3	0.861111	0.8	0.833333	0.769231	0.695182	0.886288
3	0.916667	0.842105	0.888889	0.8	0.787726	0.980769
3	0.805556	0.740741	0.714286	0.769231	0.586555	0.846154
3	0.888889	0.8	0.8	0.8	0.723077	0.953846
3	0.914286	0.857143	1	0.75	0.814532	0.963768
3	0.942857	0.9	0.818182	1	0.869048	0.987179

2	0.722222	0.375	0.375	0.375	0.196429	0.59375
2	0.694444	0.266667	0.25	0.285714	0.075032	0.591133
2	0.666667	0.333333	0.428571	0.272727	0.131204	0.638182
2	0.777778	0.555556	0.714286	0.454545	0.435936	0.787273
2	0.777778	0.555556	1	0.384615	0.534191	0.687291
2	0.722222	0.444444	0.5	0.4	0.265197	0.811538
2	0.777778	0.6	0.857143	0.461538	0.507372	0.729097
2	0.777778	0.428571	0.75	0.3	0.37275	0.811538
2	0.742857	0.571429	0.666667	0.5	0.401363	0.675725
2	0.657143	0.333333	0.333333	0.333333	0.102564	0.521368

1	0.777778	0.2	0.5	0.125	0.162051	0.511161
1	1	1	1	1	1	0.842365
1	0.972222	0.952381	1	0.909091	0.934947	0.607273
1	0.944444	0.9	1	0.818182	0.870388	0.916364
1	0.888889	0.846154	0.846154	0.846154	0.759197	0.772575
1	1	1	1	1	1	0.723077
1	0.916667	0.88	0.916667	0.846154	0.817861	0.650502
1	0.916667	0.823529	1	0.7	0.792203	0.901923
1	0.971429	0.956522	1	0.916667	0.937268	0.855072
1	0.885714	0.8	0.727273	0.888889	0.72823	0.683761

As the above shows, they are predictions for each validation dataset of each classifier (model 1 is Naïve bayes, model 2 is KNN, model 3 is Multilayer perceptron) In the above we can see that the precision, recall, mcc and auc have tendency with consistency between three prediction results, that is in the above results, Naïve bayes with the higher values of those four measurements get the highest accuracy, and in the result of KNN, it has lower values of the four measurements then getting the worst accuracy.

3. **Key insights:**

First, in this homework, my training results shows that Naïve bayes classifier may have better performance than Multilayer perceptron, I think this tells me that the features in the dataset may be independent with each other. On the other hand, if once the features are not independent, Multilayer perceptron classifier may be a better choice for it is more flexible when using. And as for KNN, the prediction result is based on the closest neighbors rather than all data points, so the problem is that we may think KNN classifier is a lazy learner. Then when going to comparable assessment, on the side of complexity and performance, Multilayer perceptron classifier has much more complicated structure and is able to generally achieve higher accuracy, KNN classifier is simple but may suffer from the curse of

dimensionality, so it may not do well in high-dimensional problem, as for Naïve bayes, it is efficient and straightforward but need strong feature independent assumption.

The above is the whole of my report for the term project, thank you for reading it.