AI and Machine Learning, Homework6

Author: Ying Yiwen
Number: 12210159

Contents

1 Principle of KNN

KNN, or K-Nearest Neighbor algorithm, is a basic classification and regression method. Its main principle is to perform classification by measuring the distance between different feature values.

Training: In the KNN algorithm, we first need to have a training dataset that has been labeled with categories. For the training process, all we have to do is to add all the data points to the model.

Predicting: When we need to classify a new data point, we calculate its distance from all the points in the training dataset and select the K points with the closest distance. Then, we decide the category of the new data point based on the categories of these K points. If most of these K points belong to a certain category, then the new data point is categorized into that category.

How to define the distance and how to select K are the keys to the KNN algorithm. In practice, the most commonly used **distance metric** is the Euclidean distance. Euclidean distance calculates the straight line distance between two points in a multidimensional space. The formula is as follows:

$$\|\mathbf{x}^{(a)} - \mathbf{x}^{(b)}\|_2 = \sqrt{\sum_{j=1}^{d} \left(x_j^{(a)} - x_j^{(b)}\right)^2} \tag{1}$$

For the **choice of k**, if k is very small, it may be very sensitive to localized noise, leading to overfitting and the model does not take into account the overall distributional information. Whereas a larger k will give better results, according to the rule of thumb, k must be satisfied $k \leq \sqrt{n}$. However, too large k may cause the judgment to be based on samples that are very far away from the test point, and the decision boundary of the model becomes very vague, which can lead to underfitting.

KNN is able to generate complex decision boundaries naturally and it can perform well for problems with a large number of samples. However, the KNN algorithm is sensitive to sample noise and has high requirements for the number of samples. It may not be applicable to all scenarios.

2 Main Logic

2.1 Prediction

The main function of knn is the prediction of new point. With all samples added into the model, we can easily get the label of the new point.

```
# predict the label of a single point
dist, idx = self.kdtree.query(x, k=self.k, p=2) # Find k nearest neighbors
neighbors_labels = [self.y_train[i] for i in idx[0]]  # Get labels of neighbors
prediction = max(set(neighbors_labels), key=neighbors_labels.count) # Majority votes
```

2.2 Normalization

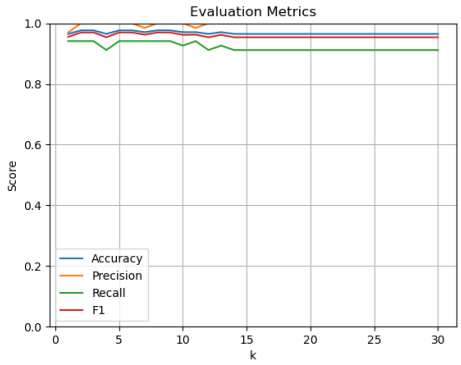
At the same time, the normalization of samples may do well with knn. We can use mean normalization, which normalizes the data to a normal distribution with mean 0 and variance 1. This better balances the weights of the different features.

$$x' = \frac{x - \mu}{\sigma} \tag{2}$$

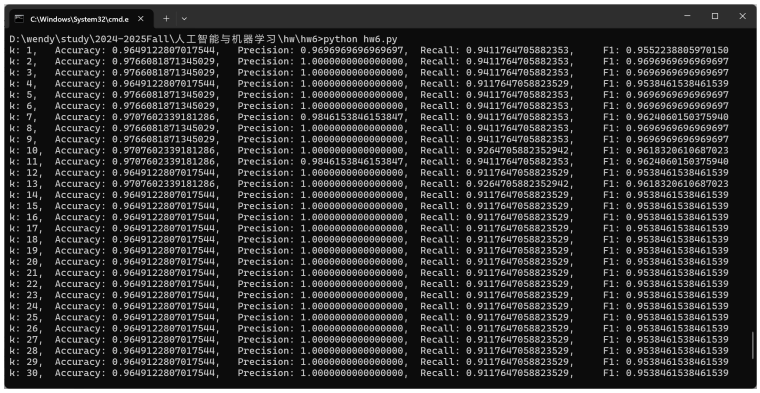
3 Effect of Different Methods

I implemented the model of knn using the principles mentioned before. For the core of the model, Euclidean distance is used. The size of k and normalization or not was discussed. Evaluating the model using accuracy, precision, recall, and F1 score, we found that knn achieved good results with this breast cancer dataset.

From the figure, we can see that:
When k is very small, it is true that it is not possible to take into account all the samples, and thus the model does not perform well. When k is increased to a certain level (roughly 5-10 here), the model performance is already excellent. When k is larger, the model performance does not improve significantly, but rather tends to decrease slightly.
Normalization is really useful for knn, as it balances the weights of the individual features so that they can all be taken into account, thus giving a better fit. Comparing the two graphs it can be seen that especially when the number of k is very small, normalization is able to improve the model very well; and for all cases of k, normalization is able to make the fit better.

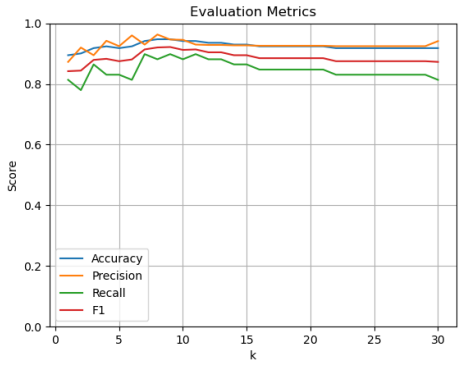


(a) Visualized Trend

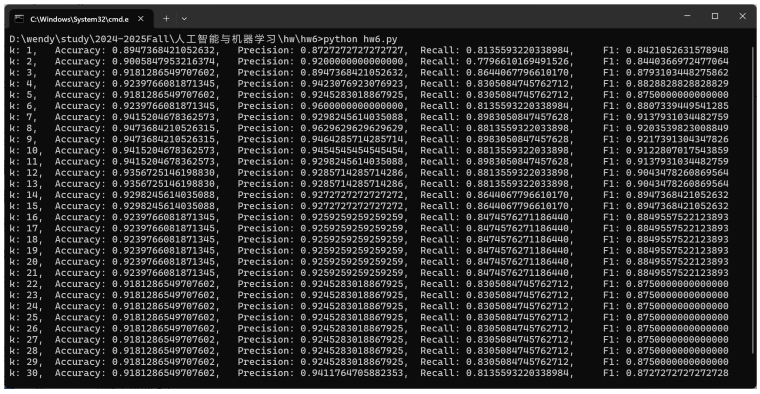


(b) Data in the Figure

Fig. 1: With Normalization



(a) Visualized Trend



(b) Data in the Figure

Fig. 2: Without Normalization

4 Related Code Implementation

4.1 Data Preprocess

In reading the data, we give the feature name for the features for our convinence.

```
def read_dataset():
  # read dataset
  column_names = [
    'ID number', 'Diagnosis',
    'radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean','smoothness_mean', '
        compactness_mean', 'concavity_mean', 'concave_points_mean','symmetry_mean', '
        fractal_dimension_mean', 'radius_se', 'texture_se', 'perimeter_se', 'area_se','
        smoothness_se', 'compactness_se', 'concavity_se', 'concave_points_se','symmetry_se',
         'fractal_dimension_se', 'radius_worst', 'texture_worst', 'perimeter_worst', '
        area_worst','smoothness_worst', 'compactness_worst', 'concavity_worst', '
        concave_points_worst','symmetry_worst', 'fractal_dimension_worst',
  ]
  data = pd.read_csv('wdbc.data', header=None, names=column_names)
  return split_data(data)
```

Then, we can separate features and labels, so as to make it easy for the model to use the dataset.

```
def split_data(data):
  # split data into features and labels
  data = data.sample(frac=1)
  # separate features and labels
  x = data.iloc[:, 2:]
  y = data.iloc[:, 1]
  return split_test_and_train(x, y)
```

End up picking up the labels, we can split them into training set and testing set.

```
def split_test_and_train(x, y):
  # split data into train and test
  x = x.values
  x = normalize_data(x)
  x_train = x[:int(len(x)*0.7), :]
  x_test = x[int(len(x)*0.7):, :]
  y = y.values
  y_train = y[:int(len(y)*0.7)]
  y_test = y[int(len(y)*0.7):]
  return x_train, x_test, y_train, y_test
```

Here is our choice for normalization, we can choose whether to use x = normalize_data(x) or not.

```
def normalize_data(x):
  # normalize data
  x = (x - np.mean(x, axis=0)) / np.std(x, axis=0)
  return x
```

4.2 Evaluation

It's important for us to evaluate the model after training (though the training here is so simple). We still use accuracy, precision, recall and f1 score to evaluate it.

```
def evaluate(self, X_test, y_test):
  # predict the label of multiple points
  predictions = self.predict_multiple(X_test)
```

```
    # calculate TP, TN, FP, FN
    TP = np.sum((predictions == 'M') & (y_test == 'M'))
    TN = np.sum((predictions == 'B') & (y_test == 'B'))
    FP = np.sum((predictions == 'M') & (y_test == 'B'))
    FN = np.sum((predictions == 'B') & (y_test == 'M'))
    # calculate accuracy, precision, recall, f1
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP)
    recall = TP / (TP + FN)
    f1 = 2 * precision * recall / (precision + recall)
    return accuracy, precision, recall, f1
```

4.3 Main Function

All in all, the main function is like this:

```
# read dataset
x_train, x_test, y_train, y_test = read_dataset()

# test different k values
accuracy_list, precision_list, recall_list, f1_list = [], [], [], []
for k in range(1, 11):
    # Create KNN instance and fit the model
    knn = KNN(k=k)
    knn.fit(x_train, y_train)
    # Evaluate the model
    accuracy, precision, recall, f1 = knn.evaluate(x_test, y_test)
    accuracy_list.append(accuracy)
    precision_list.append(precision)
    recall_list.append(recall)
    f1_list.append(f1)
    print(f"k: {k},\tAccuracy: {accuracy:.16f},\tPrecision: {precision:.16f},\tRecall: {
        recall:.16f},\tF1: {f1:.16f}")

# plot the evaluation metrics
plot_evaluation_metrics(accuracy_list, precision_list, recall_list, f1_list)
```