Hw2

Runcheng Liu 2018010316

3/15/2021

1

In LDA, suppose that we model each class density as multivariate Gaussian,

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\Sigma_k|^{1/2}} \exp\left\{ -\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1} (x-\mu_k) \right\}$$

In comparing two classes k and l, it is sufficient to look at the log-ratio,

$$\log \frac{P(k \mid x)}{P(l \mid x)} = \log \frac{f_k(x)\pi_k}{f_l(x)\pi_l}$$

LDA arises in the speacial case when we assume that the classes have a common covariance matrix $\Sigma_k = \Sigma$, for any k, thus,

$$\begin{aligned} \log \frac{P(k \mid x)}{P(l \mid x)} &= \log \frac{f_k(x)\pi_k}{f_l(x)\pi_l} \\ &= \log \frac{\pi_k}{\pi_l} - \frac{1}{2}(\mu_k + \mu_l)^T \Sigma^{-1} (\mu_k - \mu_l) + x^T \Sigma^{-1} (\mu_k - \mu_l) \\ &= \left(\log \pi_k + x^\intercal \Sigma^{-1} \mu_k - \frac{1}{2}\mu_k \Sigma^{-1} \mu_k \right) - \left(\log \pi_l + x^\intercal \Sigma^{-1} \mu_l - \frac{1}{2}\mu_l \Sigma^{-1} \mu_l \right) \\ &= \delta_k - \delta_l \end{aligned}$$

Thus, $\log P(k \mid x) = \delta_k$, $\log P(l \mid x) = \delta_l$, so,

$$\begin{aligned} \hat{y}(x) &= \text{argmax}_k \, P(k \mid x) \\ &= \text{argmax}_k \, \log P(k \mid x) \\ &= \text{argmax}_k \, \delta_k(x) \end{aligned}$$

2.1

From LDA, we know that LDA rule classifies to class 2 if,

$$\log \frac{P(2 \mid x)}{P(1 \mid x)} > 1$$

That is,

$$\log \frac{\hat{\pi}_2}{\hat{\pi}_1} - \frac{1}{2}(\hat{\mu}_2 + \hat{\mu}_1)\Sigma^{-1}(\hat{\mu}_2 - \hat{\mu}_1) + x^\intercal \Sigma^{-1}(\hat{\mu}_2 - \hat{\mu}_1) > 0$$

Where $\hat{\pi}_2 = N_2/N$, $\pi_1 = \hat{N}_1/N$, thus, the LDA rule classifies to class 2 if,

$$x^\intercal \Sigma^{-1}(\hat{\mu}_2 - \hat{\mu}_1) > \frac{1}{2}(\hat{\mu}_2 + \hat{\mu}_1)\Sigma^{-1}(\hat{\mu}_2 - \hat{\mu}_1) - \log \frac{N_2}{N_1}$$

and class 1 otherwise.

2.2

Let $U_i \in \mathbb{R}^n$ be the class indicator vector of class i, and $U = U_1 + U_2$ be the vector with all entries equal to 1. In this problem, the vector of labels becomes $Y = (-N/N_1)U_1 + (N/N_2)U_2$. And the least square criterion $RSS(\beta_0, \beta)$,

$$RSS(\beta_0, \beta) = \sum_{i=1}^{N}(y_i - \beta_0 - x_i^\intercal \beta) = (Y - \beta_0 U - X\beta)^\intercal (Y - \beta_0 U - X\beta)$$

Where X is a $N \times p$ matrix. To obtain the minimization of RSS, take the derivative of $RSS(\beta_0, \beta)$ with respect to β_0 and β, and set them to 0,

$$\nabla_\beta RSS = -2X^\intercal Y + 2\beta_0 X^\intercal U + 2X^\intercal X\beta = 0$$

and

$$\nabla_{\beta_0} RSS = -2U^\intercal Y + 2\beta_0 U^\intercal U + 2U^\intercal X\beta = -2U^\intercal Y + 2N\beta_0 + 2U^\intercal X\beta = 0$$

From the second equation, we obtain that

$$\hat{\beta}_0 = \frac{1}{N}U^\intercal (Y - X\beta)$$

Take this into the first equation, we obtain that

$$(-\frac{1}{N}X^\intercal UU^\intercal X + X^\intercal X)\hat{\beta} = X^\intercal Y - \frac{1}{N}X^\intercal UU^\intercal Y$$

And we have that $X^\intercal U_i = N_i \hat{\mu}_i$ for $i = 1, 2$. so for the left-hand side,

$$(-\frac{1}{N}X^\intercal UU^\intercal X + X^\intercal X)\hat{\beta} = \left(-\frac{1}{N}\left(N_1^2 \hat{\mu}_1 \hat{\mu}_1^\intercal + N_1 N_2 \hat{\mu}_1 \hat{\mu}_2^\intercal + N_1 N_2 \hat{\mu}_2 \hat{\mu}_1^\intercal + N_2^2 \hat{\mu}_2 \hat{\mu}_2^\intercal\right) + X^\intercal X\right)\hat{\beta}$$

And the estimate of the covariance matrix used in LDA is given by:

$$\begin{aligned}(N-2)\hat{\Sigma} &= \sum_{i:y_i=-N/N_1}(x_i - \hat{\mu}_1)(x_i - \hat{\mu}_1)^\intercal + \sum_{i:y_i=N/N_2}(x_i - \hat{\mu}_2)(x_i - \hat{\mu}_2)^\intercal \\ &= X^\intercal X - N_1 \hat{\mu}_1 \hat{\mu}_1^\intercal - N_2 \hat{\mu}_2 \hat{\mu}_2^\intercal\end{aligned}$$

And denote $\hat{\Sigma}_B = \frac{N_1 N_2}{N^2}\left(\hat{\mu}_2 - \hat{\mu}_1\right)\left(\hat{\mu}_2 - \hat{\mu}_1\right)^T$, thus, the L.H.S. is equal to:

$$\left((N-2)\hat{\Sigma} + N\Sigma_B\right)\hat{\beta}$$

And the R.H.S. is:

$$\begin{aligned} X^\intercal Y - \frac{1}{N} U^\intercal Y &= X^\intercal(-\frac{N}{N_1}U_1 + \frac{N}{N_2}U_2) - \left(N_1\hat{\mu}_1 + N_2\hat{\mu}_2\right)\left(-\frac{N}{N_1}N_1 + \frac{N}{N_2}N_2\right) \\ &= X^\intercal(-\frac{N}{N_1}U_1 + \frac{N}{N_2}U_2) \\ &= N(\hat{\mu_2} - \hat{\mu_1}) \end{aligned}$$

So the equation is equal to:

$$\left[(N-2)\hat{\Sigma} + N\hat{\Sigma}_B\right]\hat{\beta} = N\left(\hat{\mu}_2 - \hat{\mu}_1\right),$$

Where $\hat{\Sigma}_B = \frac{N_1 N_2}{N^2}\left(\hat{\mu}_2 - \hat{\mu}_1\right)\left(\hat{\mu}_2 - \hat{\mu}_1\right)^T$.

2.3

And let a real number $\lambda = (\hat{\mu}_2 - \hat{\mu}_1)^\intercal\hat{\beta}$, then

$$\hat{\Sigma}_B\hat{\beta} = \frac{N_1N_2}{N^2}(\hat{\mu}_2 - \hat{\mu}_1)(\hat{\mu}_2 - \hat{\mu}_1)^\intercal\hat{\beta} = \frac{N_1N_2}{N^2}(\hat{\mu}_2 - \hat{\mu}_1)\lambda$$

Hence, $\hat{\Sigma}_B\hat{\beta}$ is in the direction of $(\hat{\mu}_2 - \hat{\mu}_1)$, and from the equation,

$$\begin{aligned} (N-2)\hat{\Sigma}\hat{\beta} &= N(\hat{\mu}_2 - \hat{\mu}_1) - N\hat{\Sigma}_B\hat{\beta} \\ &= \left(N - \frac{N_1N_2}{N}\lambda\right)\left(\hat{\mu}_2 - \hat{\mu}_1\right) \end{aligned}$$

Thus,

$$\hat{\beta} \propto \hat{\Sigma}^{-1}\left(\hat{\mu}_2 - \hat{\mu}_1\right)$$

Therefore the least-squares regression coefficient is identical to the LDA coefficient, up to a scalar multiple.

3.1

Use KNN classifier, and plot 5-fold cross-validation accuracy, here is the code

```
from random import seed
from random import randrange
from csv import reader
from math import sqrt
import matplotlib
import matplotlib.pyplot as plt
import numpy as np


# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset


# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())


# Convert string column to integer
def str_column_to_int(dataset, column):
    for row in dataset:
        # print(row[column])
        row[column] = int(row[column].strip())
        # print(row)


# Find the min and max values for each column
def dataset_minmax(dataset):
    minmax = list()
    for i in range(len(dataset[0]) - 1):
        col_values = [row[i] for row in dataset]
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax


# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
    new = []
    for row in dataset:
        for i in range(len(row) - 1):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])
    new.append(row)
```

```python
    return new


# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for _ in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split


# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0


# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores


# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1) - 1):
        distance += (row1[i] - row2[i]) ** 2
    return sqrt(distance)
```

```python
# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors


# Make a prediction with neighbors by max
def predict_classification_max(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction


# Make a prediction with neighbors by average
def predict_classification_average(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = round(sum(output_values) / len(output_values))
    return prediction


# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
    predictions = list()
    for row in test:
        output = predict_classification_max(train, row, num_neighbors)
        predictions.append(output)
    return (predictions)


# Test the kNN on the wine dataset
seed(1)
filename = 'winequality-red.csv'
dataset = load_csv(filename)
dataset = dataset[1:1600]

# Remove the delimiter
for i in range(len(dataset)):
    dataset[i] = dataset[i][0].split(';')

# Convert predicor column into float
for i in range(len(dataset[0]) - 1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0]) - 1)
```

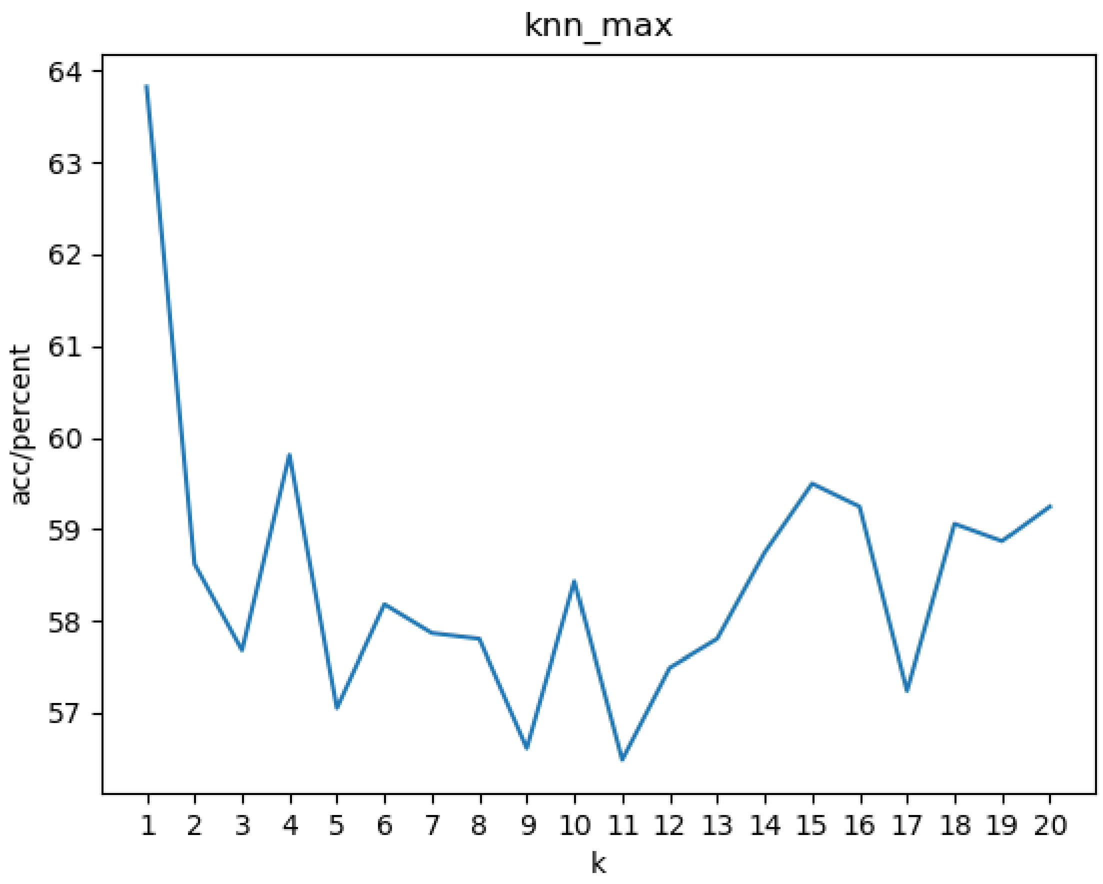
```
# Normalization
normalize_dataset(dataset=dataset, minmax=dataset_minmax(dataset=dataset))

# evaluate algorithm
acc = []
n_folds = 5
L = 20
for num_neighbors in range(L):
    print("K =", num_neighbors + 1)
    scores = evaluate_algorithm(dataset, k_nearest_neighbors, n_folds, num_neighbors + 1
)
    print('Scores: %s' % scores)
    print('Mean Accuracy: %.3f%%' % (sum(scores) / float(len(scores))))
    acc.append((sum(scores) / float(len(scores))))

x = np.arange(1, L + 1, 1)
fig, ax = plt.subplots()
ax.plot(x, acc)
ax.set(xlabel='k', ylabel='acc/percent',
       title='knn_max')
ax.set_xticks([i + 1 for i in range(L)])

fig.savefig("acc_knn_max.png")
```

The result is,



Accuracy of knn with max

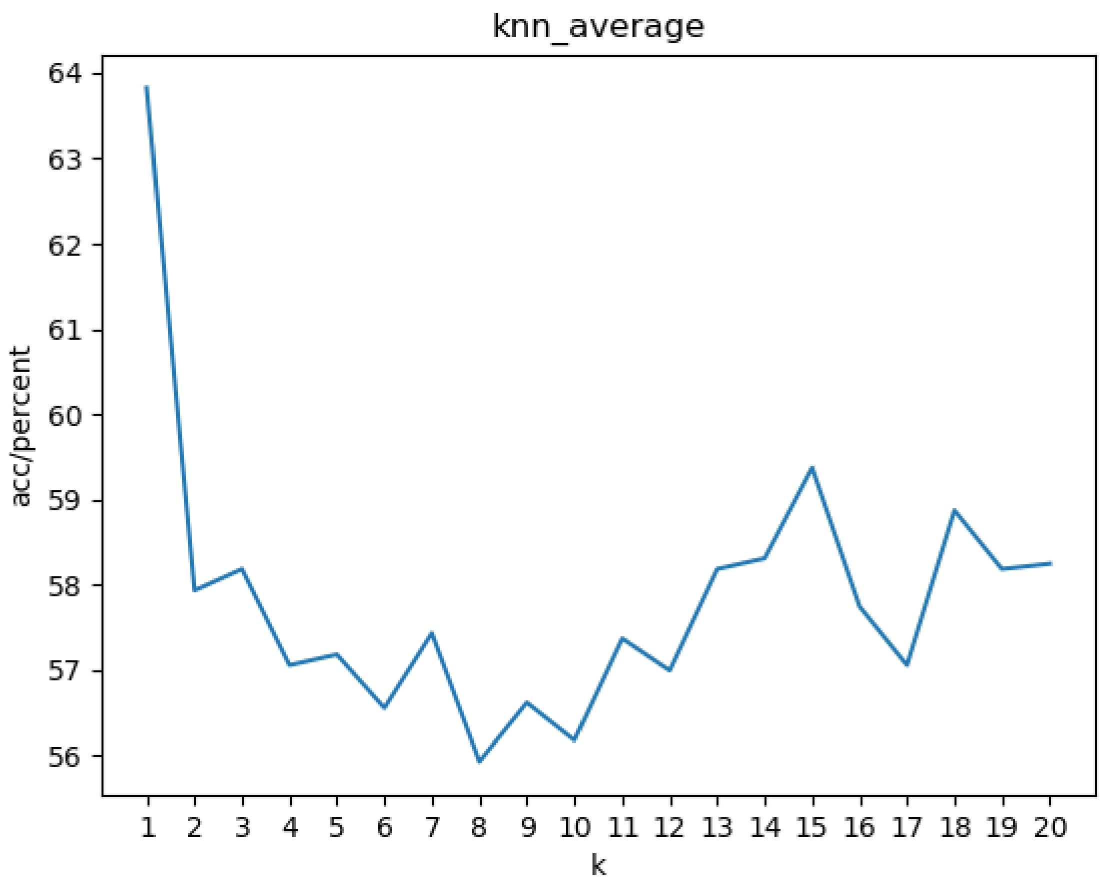
So from the plot, we can see that at k=1, accuracy is at maximum, but it is kind of over-fitting. When k=1 you estimate your probability based on a single sample: your closest neighbor. This is very sensitive to all sort of distortions like noise, outliers, mislabelling of data, and so on. By using a higher value for k, you tend to be more robust against those distortions. So the optimal k value should be at 4, which is at the local maximum.

3.2

Use kNN regression by taking the average ofthe levels of the nearest neighbors and rounding it to the nearest level, just change the predict_classification_max() functions to predict_classification_average(), other functions are the same,

```python
# Make a prediction with neighbors by average
def predict_classification_average(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = round(sum(output_values) / len(output_values))
    return prediction
```

The result is,



Accuracy of knn with average

After using the average, we can see that the local maximum is at k = 15. Compared to the previous one k = 4, this k is larger, so the model should be more robust than the previous.