PRML Assignment 1

Dataset

https://github.com/fields1631/fields-s-repository-for-FDU-PRML2020

Description and analysis of model

The linear discriminative model is defined by Python class ***LinearDiscriminativeModel*** in ***source.py***. On the linear classification of n dimensional C classes, an instance of the model has C $n \times 1$ weight vectors, i.e. an $n \times C$ weight matrix. The weight matrix is specified on calling the method ***train*** to train the model. The method *train* uses **gradient descent strategy** and you can specify the training options. Once the model is trained, you can classify samples by calling the ***classify*** method, who uses **softmax** function to generate the probability that samples belong to each class.

The linear generative model is defined by Python class ***LinearGenerativeModel*** in ***source.py***. On the linear classification of n dimensional C classes, an instance of the model has C $n \times 1$ weight vectors, C biases and one covariance matrix, i.e. an $n \times C$ weight matrix, a $C \times 1$ bias vector and an $n \times n$ covariance matrix. The weight matrix, bias vector and covariance matrix are specified on calling the method ***train*** to train the model. The method *train* calculates mathematical expectations and covariance matrices of each class using **maximum likelihood estimation**. Once the model is trained, you can classify samples by calling the ***classify*** method, who uses **softmax** function to generate the probability that samples belong to each class.

The **major differences** between these two models are the **training method** and **classification method**. The **discriminative model** uses gradient descent strategy, i.e. an **optimization method**, to approach the ideal weight matrix, while the **generative model** tries to generalize the statistical properties of given samples using **statistical method**. On the classification method, the **discriminative model** assumes the probability that samples belong to each class has the form of softmax($w^T x$), which comes from the **linear property** of the classifier. However, the **generative model** adopts softmax($w^T x + b$) as the form of probability that samples belong to each class because it assumes that **samples are compiled to gaussian distributions**. Apart from these differences, discriminative model needs certain amount of time to train the model, and the classification accuracy rises first and then declines in the training process. The performance of two models is similar.

Demo

This demo is also at the beginning of ***source.py***.

```
>>> from source import *
>>> create_dataset()
>>> samples, labels = load_dataset()
>>> set_of_samples, set_of_labels = split_dataset(samples, labels, [0.8])
>>> training_samples, testing_samples = set_of_samples
>>> training_labels, testing_labels = set_of_labels
>>> model1 = LinearDiscriminativeModel()
>>> model1.train(training_samples, training_labels, max_epochs=20)
Epoch   Accuracy
1       0.9590163934426229 #random
2       0.9639344262295082
3       0.9655737704918033
4       0.9672131147540983
5       0.9688524590163935
6       0.9688524590163935
7       0.9672131147540983
8       0.9704918032786886
9       0.9721311475409836
10      0.9721311475409836
11      0.9721311475409836
12      0.9737704918032787
13      0.9737704918032787
14      0.9737704918032787
15      0.9737704918032787
16      0.9737704918032787
17      0.9737704918032787
18      0.9737704918032787
19      0.9737704918032787
20      0.9737704918032787
>>> labels1 = model1.classify(testing_samples)
>>> acc = sum(testing_labels == labels1) / testing_labels.size
>>> print(model1.w)
[[ 2.62622022 -1.31718143 -1.30903878]  #random
 [ 0.72619941 -3.81306407  3.08686466]]
>>> print(acc[0, 0])
0.9794871794871794  #random
>>> model2 = LinearGenerativeModel()
>>> model2.train(training_samples, training_labels)
>>> labels2 = model2.classify(testing_samples)
>>> acc = sum(testing_labels == labels2) / testing_labels.size
>>> print(model2.w)
[[ 9.18231285 -0.0287646   0.49775292]  #random
 [ 0.27302244 -9.82742259  9.82056657]]
>>> print(acc[0, 0])
0.9794871794871794 #random
```