Worksheet 2: Machine Learning Basics

INFO7001 Advanced Machine Learning

In this lab, we will investigate how to use python to investigate data using machine learning methods. Many of the functions we will be using are from the scikit-learn module.

To get the most out of this lab, run the script interactively (e.g. using ipython) and answer the questions as you go.

```
import sklearn # we will use the sklearn package
import sklearn.datasets
```

Data Creation

To begin, we will generate data. By generating data, we are able to set the underlying distribution and investigate the utility of the machine learning methods for this lab.

```
import numpy as np
import matplotlib.pyplot as plt

## generate training data
x1 = np.random.multivariate_normal(mean = [1, 1], cov = [[1,0],[0,1]], size = 100)
x2 = np.random.multivariate_normal(mean = [3, 1], cov = [[1,0],[0,1]], size = 100)
x3 = np.random.multivariate_normal(mean = [2, 2], cov = [[1,0],[0,1]], size = 100)

X = np.vstack([x1, x2, x3])
y = np.concatenate([np.repeat(0,200), np.repeat(1,100)])

plt.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=25, edgecolor='k')
plt.show()

## generate testing data, using the same distribution as the training data
x1 = np.random.multivariate_normal(mean = [1, 1], cov = [[1,0],[0,1]], size = 100)
x2 = np.random.multivariate_normal(mean = [3, 1], cov = [[1,0],[0,1]], size = 100)
x3 = np.random.multivariate_normal(mean = [2, 2], cov = [[1,0],[0,1]], size = 100)

Xtest = np.vstack([x1, x2, x3])
ytest = np.concatenate([np.repeat(0,200), np.repeat(1,100)])
```

Training, Testing, Regularisation and Hyper-parameters

We can train models to fit training data so that we obtain the best accuracy for the training data, but what we really want is the best accuracy for any new data that we deal with in the future (we want the model to be useful in practice, no only on our training data).

We saw in the lecture that a model's hyper-parameters are not fitted to data, but must be set before training. We can hand tune these parameters to provide the best accuracy on held-out testing data.

To begin, let's fit an SVM to our data. We will use a polynomial kernel

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j + o)^d$$

where o is the parameter `coef0` and d is `degree`. We set the hyper-parameters `C` to 1, `coef0` to 1, and `degree` to 1.

```
# http://scikit-learn.org/stable/modules/svm.html
from sklearn import svm

m = svm.SVC(kernel='poly', C=1, degree = 1, coef0 = 1)
m.fit(X, y)
```

We can now compute the accuracy (proportion of correct predictions) of the fitted model to the training and testing data.

```
m.score(X, y) # training accuracy
m.score(Xtest, ytest) # test accuracy
```

Note that parameters `C` and `degree` and `coef0` can be used for regularisation (we can tune them to provide generalisation to held out data).

1. Adjust the hyper-parameters `C` and `degree` and examine their effect on the training accuracy.
2. Adjust the hyper-parameters `C` and `degree` to try to find greater accuracy on the testing data for a model fitted to the training data.
3. Does the increasing the testing accuracy also increase the training accuracy?

```
## Answer
# Adjust C and d to examine the effect on training and testing accuracy
C = 2 # must be positive
d = 3 # must be a positive integer
m = svm.SVC(kernel='poly', C=C, degree = d, coef0 = 1)
m.fit(X, y)
print("C =", C, ", degree =", d, ", Training:", m.score(X, y),
", Testing:", m.score(Xtest, ytest))
```

The parameter `degree` is the degree of the kernel function, therefore increasing its value increases the capacity of the SVM (just as increasing the degree of a regression polynomial increases the regression capacity). Increasing the degree is likely to provide a improve training accuracy, but not always improve test accuracy.

Plotting Difference

We will plot the training and test accuracy for a range of values of `C`, while keeping `degree` and `coef0` constant.

First define function to compute the train and test error for a given value of `C`.

```
def trainAccuracy(C = 1):
    m = svm.SVC(kernel='poly', C=C, degree = 2, coef0 = 1)
    m.fit(X, y)
    return(m.score(X, y))

def testAccuracy(C = 1):
    m = svm.SVC(kernel='poly', C=C, degree = 2, coef0 = 1)
    m.fit(X, y)
    return(m.score(Xtest, ytest))
```

The create a vector containing the set of values of to be examined.

```
Cset = 2.0**np.arange(-5,5)
```

1. compute the accuracy for each C in `Cset` and store them in `trainAcc` and `testAcc`.

```
## Answer
trainAcc = [trainAccuracy(C = C) for C in Cset]
testAcc = [testAccuracy(C = C) for C in Cset]
```

View the results using a scatter plot.

```
## view the change in train and test error with respect to C
plt.xscale('log')
plt.scatter(Cset, trainAcc, c = 'k', label = "Train")
plt.scatter(Cset, testAcc, c = 'r', label = "Test")
plt.legend()
plt.show()
```

What does this tell us about the relationship between the training and testing accuracy?

Cross Validation

Cross validation (CV) is the process of splitting the training data into equal sized partitions, so that we test using one and train on the rest, then cycle through so that all partitions are used as the testing set. This allows us to examine the effect of the parameter setting on data that is not included in the training set, without obtaining more data.

In the below code we compute the CV error using the parameters $C = 1$ and (degree) $d = 1$, leaving `coef0` constant.

```
# http://scikit-learn.org/stable/modules/cross_validation.html

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

## compute the set of CV splits, here we set the number of partitions to 10
kf = KFold(n_splits=10)
## compute the accuracy for each CV split
m = svm.SVC(kernel='poly', C=1, degree = 1, coef0 = 1)
score = cross_val_score(m, X, y, cv=kf)
## compute the mean accuracy over all CV splits
cvScore = score.mean()
print(cvScore)

## compute the test accuracy
m.fit(X, y)
m.score(Xtest, ytest)
```

To make full use of cross validation, we want to compute the CV accuracy over a set of parameters. This will allow us to identify which set of parameters provide the best generalisation accuracy.

```
def cvAccuracy(C = 1):
    kf = KFold(n_splits=10)
    m = svm.SVC(kernel='poly', C=C, degree = 2, coef0 = 1)
    score = cross_val_score(m, X, y, cv=kf)
    cvScore = score.mean()
    return(cvScore)

cvTrainAcc = [cvAccuracy(C = C) for C in Cset]
```

1. Plot the train, test and CV accuracy. Remember that in practice we will not have the test accuracy, so we want to be able to estimate it from either the train or CV accuracy. Which is closer to the test accuracy?

```
## Answer
plt.xscale('log')
plt.scatter(Cset, trainAcc, c = 'k', label = "Train")
plt.scatter(Cset, testAcc, c = 'r', label = "Test")
plt.scatter(Cset, cvTrainAcc, c = 'b', label = "CV")
plt.legend()
plt.show()
```

We should notice that the CV accuracy is a better approximation of the test accuracy (when CV accuracy is high, test accuracy is likely to be high). Meaning that we should use cross validation to choose model parameters.

Note that we can perform a grid search of the parameter space to choose the best parameters.

```
# http://scikit-learn.org/stable/modules/grid_search.html
# http://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html

from sklearn.model_selection import GridSearchCV
```

```
## set the search to examine gamma from 2^-5 to 2^4 and C from 2^-5 to 2^4
parameterSet = [{'kernel': ['poly'], 'degree': np.arange(1,5),
'coef0' : [1],  'C': 2.0**np.arange(-5,5)}]
m = svm.SVC()
gsm = GridSearchCV(m, parameterSet, cv=5)
gsm.fit(X, y)
## print the parameters that provide the greatest accuracy
gsm.best_params_
```

Examine the above Web references to work out how to print out the results for all parameter sets.

Bayes Error

The Bayes Error is a measure of the classification error, given that we know the underlying data generating process.

Remember that we generated the data from three Gaussians, where two belonged to one class and the third belonged to the remaining class. Since we generated the data, we know the generating process. Note that this is usually not possible in practice; we are using machine learning to discover details of the generating process.

For each point, we ask, what is the probability of class 1, given the point vector ($P(y = 1|x)$)? We know the probabilities $P(x|y = 0)$ from the first to Gaussians, $P(x|y = 1)$ from the third Gaussian, and we know $P(y = 0) = 2/3$ since we generated 2/3 of the data as class 0 and 1/3 as class 1. Therefore, we can use Bayes rule:

$$P(y = 1|x) = \frac{P(x|y = 1)P(y = 1)}{P(x|y = 1)P(y = 1) + P(x|y = 0)P(y = 0)}$$

which we can reduce to:

$$P(y = 1|x) = \frac{P(x|y = 1)/3}{P(x|y = 1)/3 + P(x|y = 0)2/3}$$

and also:

$$P(y = 0|x) = \frac{P(x|y = 0)2/3}{P(x|y = 1)/3 + P(x|y = 0)2/3}$$

where $P(x|y = 1)$ is Normal with mean $\mu = [2\ 2]$ and the identity matrix as its covariance matrix (we set this at the start of the worksheet). $P(x|y = 0)$ is a mixture of two Normal distributions with means $\mu = [1\ 1]$ and $[3\ 1]$ and the identity matrix as their covariance matrix. Therefore:

$$P(x|y = 0) = P(x|z_1)P(z_1) + P(x|z_2)P(z_2)$$

where $P(x|z_1)$ is Normal with mean $\mu = [1\ 1]$ and the identity matrix as its covariance matrix, and $P(x|z_2)$ is Normal with mean $\mu = [3\ 1]$ and the identity matrix as its covariance matrix. Since we sampled equally from z_1 and z_2, $P(z_1) = P(z_2) = 0.5$.

Note that since the first class is a mixture of two Normal distributions, we compute the probability of each point as $0.5p_1 + 0.5p_2$, where p_1 and p_2 are the two normal distributions.

Below is an example of how to compute the probability (we should really say density) from a 2d Normal distribution:

```
from scipy.stats import multivariate_normal

d1 = multivariate_normal(mean=[1,1], cov=[[1,0],[0,1]])
p1 = d1.pdf(X) # compute the density of each point in X
```

1. Compute the probability of each point in X belonging to each class.
2. Compute the most likely class for each point in X (the class with the greatest probability).
3. Compute the Bayes accuracy by comparing the computed classes to the true classes.
4. How does this compare to the SVM accuracy?

```
## Answer
from scipy.stats import multivariate_normal

d1 = multivariate_normal(mean=[1,1], cov=[[1,0],[0,1]])
d2 = multivariate_normal(mean=[3,1], cov=[[1,0],[0,1]])
d3 = multivariate_normal(mean=[2,2], cov=[[1,0],[0,1]])

p1 = d1.pdf(X)
p2 = d2.pdf(X)
p3 = d3.pdf(X)

pxC0 = 0.5*p1 + 0.5*p2
pxC1 = p3

## compute the probabilty of each point being class 1
pY1 = (pxC1/3)/(pxC0*2/3 + pxC1/3)
## compute the probabilty of each point being class 0
pY0 = (pxC0*2/3)/(pxC0*2/3 + pxC1/3)

## the log of the ratio is positive is y = 1 is more likely, or
## negative if y = 0 is more likely.
log(pY1/pY0)

## Or we can round the probability of y = 1, to get the more likely class
## since pY0 = 1 - pY1
np.round(pY1)

## Examine the accuracy to the true class values
np.mean(np.round(pY1) == y)

# We can see that even if we know the underlying generating distribution
# of the data, there is still error in our prediction, due to regions of
#uncertainty in the data space (where the two classes overlap).
```