School of Electronics and Computer Science
University of Southampton

COMP6245(2020/21): Foundations of Machine Learning (Assessed work 20%**)**

Issue	25 November 2020
Deadline	9 December 2020 (16:00)

This work forms part of the assessment of this module. It is important you work independently. For the purposes of *learning*, you are required to complete **all** the four tasks below. However, for the purpose of *assessment* you only need to submit a short report (a) confirming the completion of the first task (the five labs) and (b) **any one** of the three defined below. The report should be no longer than **four** pages (this is an absolute maximum, *i.e.* no coverpages / appendices etc.).

Tasks

10 Marks To ensure all previous Labs (1-5) have been completed and to confirm that any feedback provided in Labs 1-2 has been taken on board.

10 Marks To implement a multiplicative update algorithm for non-negative matrix factorization and learn some aspects of its use.

10 Marks To implement and study some aspects of the $K-$ means clustering algorithm.

10 Marks To demonstrate how a multi-layer perceptron (MLP) classifier approximates posterior probabilities of a Bayesian classifier.

1 Labs 1 - 5

State in your report that you have completed all the five labs and uploaded reports. Also confirm that where ‘‘`Incomplete`’’ has been noted in the feedback provided (Labs 1 - 2), you have re-visited the tasks and completed them.

2 Non-Negative Matrix Factoriztion

Non negative matrix factorization is a powerful dimensioanlity reduction method, applicable when we have data that is all positive and we have reason to represent it as a product of two matrices with all positive values. Unlike Principal Component analysis which is variance preserving, constraining the factors to be positive encourages sparsity in the representation. Hence NMF is seen as a method to achieve a *parts-based representation*, as you have seen in the background reading required (D. Lee and S. Seung's paper in *Nature*).
We will implement a multiplicative update algorithm (as described in the above paper). Skeleton code to set up a low rank matrix obtained by multiplying two positive valued matrices and the iterative loop to compute the factors is given in the Appendix.

1. Complete the code and verify that the factorization is correct. For synthetic data with the rank correctly specified, this is likely to be the case. Correct convergence of the algorithm might result in the reduction of error similar to what is shown in 1. Try factorising with different ranks (r) and check if the algorithm fails.

2. Compare the factoriation results with the `nmf` package in `sklearn.decomposition`.

3. The file `equities.xlsx` consists of the daily FTSE100 stock index values and the values of 95 of its constituent assets taken during the period August 2011 to February 2019. A passive investor

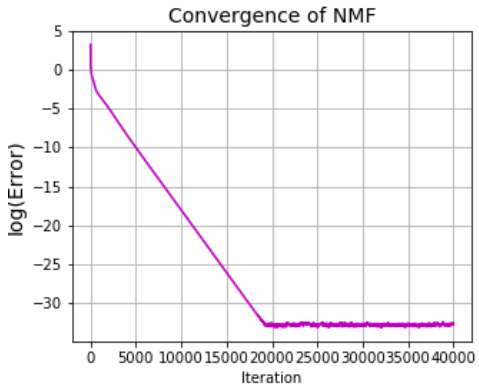


Figure 1: Convergence of the NMF algorithm on a synthetic dataset

is someone who invests in the stock market by attempting to gain returns comparable to the index, rather than actively selecting a small number of stocks to invest in. However, investing in all 100 components of the FTSE is difficult due to transaction costs. Hence *factor trading* – an approach in which one invests in a combination of assets that might best track the index by being dominant determinants of the market.

In this task you are asked to verify if non-negative matrix factorization may be a way of achieving such trading by following the steps below:

- Select a convenient rank (say $r = 10$) and factorize the equities data ($T \times N$ data matrix, of number of time points T and number of assets N). Columns of the $T \times r$ matrix are the factors of interest.
- Rank these factors by their correlation with the FTSE index and select the one that correlates most.
- Compare the cummulative returns you will get by investing in the FTSE and in the selected factor.
- Does the factor perform better than an equally weighted basket of the same number, r, of assets chosen at random?

Note the factorization and selection should be done on a training set and the evaluation of returns on an independent test set. It is suggested you use the first half of the time series for training and the second half for evaluation.

3 K-Means Clustering

A mixture of Gaussians probability density model of K components of a multi-variate vector $\boldsymbol{x} \in \mathcal{R}^p$ is defined by

$$p(\boldsymbol{x}) = \sum_{j=1}^{K} \lambda_j \mathcal{N}(\boldsymbol{m}_j, C_j),$$

where $\mathcal{N}(\boldsymbol{m}_j, C_j)$ is a multivariate Gaussian density we are already familiar with, and λ_j are positive numbers such that $\sum_{j=1}^{K} \lambda_j = 1$. It is a weighted sum of multi-variate Gaussians, with the constraints on the weights ensuring it is a proper density.

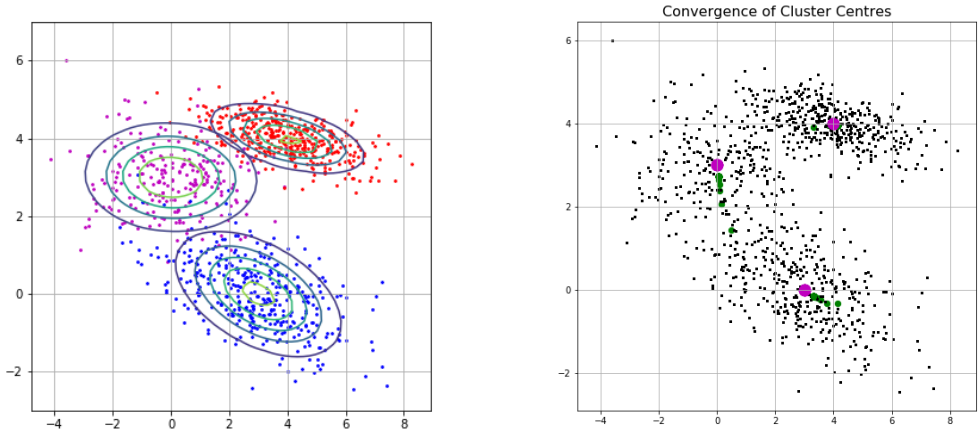
1. Sample data from a mixture Gaussian density and implement $K-$ means clustering algorithm. Snippet of code is provided in Appendix to randomly set means, covariances and proportions to define a model, and to sample from it (for $K = 3$). You have to write code for the $K-$means iterations yourself. The implementation of your algorithm should produce results similar to that in Fig. 2.

2. Draw contours on the probability density you have used and compare with regions associated with each cluster.

3. Compare your results with the $K-$means clustering algorithm in `sklearn`.

4. It is said that the $K-$means algorithm is sensitive to the initial guess of the cluster centers and the choice of K. Is this the case in your implementation? Show an example of the algorithm failing.

5. Select a $K-$class classification dataset from the `UCI` repository, cluster the input data using $K-$means clustering and check how well the clusters relate to the targets defined in the dataset.



(a) **(b)**

Figure 2: Data from a mixture Gaussian density (a), and cluster centres from $K-$means clustering (b). Initial guess of cluster centres and their different estimates during iterations are marked in green and the converged answer in magenta.

4 Multi-Layer Perceptron

The Multi-Layer Perceptron (MLP) is a widely used architecture for pattern classification and non-linear regression. It is far more flexible than the linear and RBF models we have seen in Labs 3-5, but comes with difficulties in training. In this task, you are asked to study how well an MLP approximates posterior probabilities of a classification problem.
Snippets of code to generate a multi-class classification problem in two dimensions is given in the Appendix.

1. Set up two classification problems, one relatively easy to learn (i.e.) the classes are far apart and another in which the classe overlap (say, approximately 20% of the data overlap) and difficult to learn. Each class may be either a Gaussian or a Mixture of Gaussians.

2. Split the data into training and test sets and implement a Bayesian classifier and an MLP classifier and compare their performances. Your answer should be in the form of two boxplots obtained by ten-fold cross validation.

3. For one of the partitions (of training - test split), plot the class boundaries from the Gaussian and MLP classifiers. Compare a very simple MLP (with a very small number of hidden nodes) and a complex one.

4. In the piece of code given, the MLP classifier is being called with default settings. Running this will hint at all the possible parameters one is free to set and attempt to improve the model being built as follows:

```
MLPClassifier(activation='relu',
    alpha=0.0001,
    batch_size='auto',
    beta_1=0.9,
    beta_2=0.999,
    early_stopping=False,
    epsilon=1e-08,
    hidden_layer_sizes=(100,),
    learning_rate='constant',
    learning_rate_init=0.001,
    max_iter=200,
    momentum=0.9,
    nesterovs_momentum=True,
    power_t=0.5,
    random_state=None,
    shuffle=True,
    solver='adam',
    tol=0.0001,
    validation_fraction=0.1,
    verbose=False,
    warm_start=False)
```

Read in the documentation what the role of each of the parameters is, and show the effect of changing any two of them in your report (e.g. How does convergence change of you choose a different value for the `learning_rate_init` parameter?).

Appendix: Snippets of Code

To sample from a mixture of Gaussians

```
def genGaussianSamples(N, m, C):
    A = np.linalg.cholesky(C)
    U = np.random.randn(N,2)

    return(U @ A.T + m)

# Define three means
#
Means = np.array([[0, 3], [3, 0], [4,4]])

# Define three covariance matrices ensuring
# they are positive definite
#
from sklearn.datasets import make_spd_matrix
CovMatrices = np.zeros((3,2,2))
for j in range(3):
    CovMatrices[j,:,:] = make_spd_matrix(2)

# Priors
#
w = np.random.rand(3)
w = w / np.sum(w)

# How many data in each component (1000 in total)
#
nData = np.floor(w * 1000).astype(int)

# Draw samples from each component
#
X0 = genGaussianSamples(nData[0], Means[0,:], CovMatrices[0,:,:])
X1 = genGaussianSamples(nData[1], Means[1,:], CovMatrices[1,:,:])
X2 = genGaussianSamples(nData[2], Means[2,:], CovMatrices[2,:,:])

# Append into an array for the data we need
#
X = np.append(np.append(X0, X1, axis=0), X2, axis=0)
```

Non negative matrix factorization

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

# Specify the rank
#
r = 6;

# Construct a low rank matrix
#
Wtrue = np.random.rand(40,5)
Htrue = np.random.rand(5,10)
V0 = Wtrue @ Htrue
print(V0.shape)

# Dimensions of data
#
m, n = V0.shape

# Normalise columnwise
#
V = np.zeros((m,n))
for i in range(n):
    V[:,i] = V0[:,i] / np.max(V0[:,i])

# Initialize
#
W = np.random.rand(m,r);
H = np.random.rand(r,n);

MaxIter = 40000
f = np.zeros((MaxIter,1))
# Initial error
#
f[0] = np.linalg.norm(V - W @ H, ord='fro');

for iter in range(MaxIter-1):
    # Update W
    #
      ++ your code here ++
    # Update H
    #
      ++ your code here ++

    # Measure Error
    #
    f[iter+1] = np.linalg.norm(V - W @ H, ord='fro')

fig, ax =  plt.subplots(figsize=(5,4))
ax.plot(np.arange(MaxIter), np.log(f), c='m')
ax.grid(True)
print(np.linalg.norm(V - W @ H, ord='fro'))
```

Approximating Bayes Posterior: Data

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

def genGaussianSamples(N, m, C):
    A = np.linalg.cholesky(C)
    U = np.random.randn(N,2)

    return(U @ A.T + m)

NClasses = 3

# Priors
#
w = np.random.rand(NClasses)
w = w / np.sum(w)
N = 1000  # total data (Training = Test)
NPrior = np.floor(w * N).astype(int)

Scale = 10
Means = Scale*np.random.rand(NClasses, 2)

from sklearn.datasets import make_spd_matrix
CovMatrices = np.zeros((NClasses,2,2))
for j in range(NClasses):
    CovMatrices[j,:,:] = make_spd_matrix(2)

AllData_train = list()
for j in range(NClasses):
    AllData_train.append(genGaussianSamples(NPrior[j], Means[j,:], CovMatrices[j,:,:]))

X_train = AllData_train[0]
y_train = np.ones((NPrior[0], 1))
for j in range(NClasses-1):
    Xj = genGaussianSamples(NPrior[j+1], Means[j+1,:], CovMatrices[j+1,:,:])
    X_train = np.append(X_train, Xj, axis=0)
    yj = (j+2)*np.ones((NPrior[j+1], 1))
    y_train = np.append(y_train, yj)

AllData_test = list()
for j in range(NClasses):
    AllData_test.append(genGaussianSamples(NPrior[j], Means[j,:], CovMatrices[j,:,:]))

X_test = AllData_test[0]
y_test = np.ones((NPrior[0], 1))
for j in range(NClasses-1):
    Xj = genGaussianSamples(NPrior[j+1], Means[j+1,:], CovMatrices[j+1,:,:])
    X_test = np.append(X_test, Xj, axis=0)
    yj = (j+2)*np.ones((NPrior[j+1], 1))
    y_test = np.append(y_test, yj)


fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(12,4))
plt.subplots_adjust(wspace=0.3)
for j in range(NClasses):
    Xplt = AllData_train[j]
    ax[0].scatter(Xplt[:,0], Xplt[:,1], s=3)
ax[0].grid(True)
ax[0].set_title("Training Data Distributions")

ax[1].plot(y_train)
ax[1].set_title("Training Targets")

for j in range(NClasses):
    Xplt = AllData_test[j]
    ax[2].scatter(Xplt[:,0], Xplt[:,1], s=3)
ax[2].grid(True)
ax[2].set_title("Test Data Distributions")
```

Approximating Bayes Posterior: MLP Training

```
# Encoding the output
#
from sklearn.preprocessing import OneHotEncoder

onehot_encoder = OneHotEncoder(sparse=False)
y_onehot_train = onehot_encoder.fit_transform(y_train.reshape(-1, 1))

# Trainign a neural network
#
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier()
clf.fit(X_train, y_onehot_train)

# Predictions, accuracy and confusion matrix
#
from sklearn.metrics import accuracy_score
y_pred_train = clf.predict(X_train)
print(accuracy_score(y_onehot_train, y_pred_train))

N_train = X_train.shape[0]
predicted_class_train = np.zeros((N_train,1))
for j in range(N_train):
    predicted_class_train[j] = (1+np.argmax(y_pred_train[j,:])).astype(int)

from sklearn.metrics import confusion_matrix
print("Confusion Matrix: ")
print(confusion_matrix(y_train, predicted_class_train))
```