School of Electronics and Computer Science
University of Southampton

Foundations of Machine Learning Exercises

1 Multivarite Gaussians

To study two uses of properties of multi-variate Gaussian densities (sampling and projection):

$$\boldsymbol{x} \sim \mathcal{N}\left(\boldsymbol{m}, C\right), \; \boldsymbol{y} = A\,\boldsymbol{x} \implies \boldsymbol{y} \sim \mathcal{N}(A\boldsymbol{x}, A\,C\,A^T)$$

1.1 Getting Started

```
%matplotlib online
import matplotlib.pyplot as plt
import numpy as np
print("Hello World!")
```

Here is a quick refresher on some manipulations on vectors and matrices we will need in this module along with some simple commands in Python:

1. Scalar product of two vectors: $a = \boldsymbol{x}^T\,\boldsymbol{y}$
2. Vector norm: $b = \sqrt{\sum_{i=1}^{d} x^2(i)}$
3. Angle between two vectors
4. Symmetric matrix: $B^T \;=, B$
5. Rank of a matrix as the number of linearly independent rows/columns
6. Matrix vector multiplication: $A\,\boldsymbol{x}$
7. Quadratic form: $\boldsymbol{x}^T\,A\,\boldsymbol{x}$
8. Trace of a matrix B: $\sum_i B_{ii}$
9. Determinant of a matrix, denoted $\texttt{det}B$ or $|B|$
10. Eigenvalues and eigenvectors: $B\,\boldsymbol{u} = \lambda\boldsymbol{u}$
11. Advanced topic: Singular Value Decomposition (SVD)
12. Please try the following to get started. At each step, you should ask yourself if you notice any specific property of matrices, vector etc. you recall from previous phase of your study.

```
x = np.array([1, 2])
y = np.array([-2, 1])
a = np.dot(x, y)
print(a)

b = np.linalg.norm(x)
c = np.sqrt(x[0]**2 + x[1]**2)
print(b, c)

theta = np.arccos(np.dot(x,y) / (np.linalg.norm(x) * np.linalg.norm(x)))

B = np.array([[3,2,1], [2,6,5], [1,5,9]], dtype=float)
print(B)
print(B - B.T)

z = np.random.rand(3)
v = B @ z
print(v.shape)

print(z.T @ B @ z)

print(np.trace(B))
print(np.linalg.det(B))

D, U = np.linalg.eig(B)
print(D)
print(U)

print(np.dot(U[:,0], U[:,1]))
print(U @ U.T)
```

What do you observe for the last command above (i.e. print(np.dot(U[:,0], U[:,1])))? Can you formally prove that this is the result you would expect for the specific structure in the matrix B?

13. Now, for some advanced material and fun, find the following two items:

    - A document with title `The Matrix Cookbook` written by K.B.Petersen and M.S.Pedersen. This is a neat resource with all the basics we need and much more. A very useful reference material to have around.
    - "*It had to be U - the SVD song*" on `youtube`, which is a nice piece of art that tells you what Singular Value Decomposition is and gives an example of where it is used.

1.2 Random Numbers and Uni-variate Densities

Generate 1000 uniform random numbers and plot a histogram.

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

x = np.random.rand(1000,1)
x = np.random.rand(1000,1)

fig, ax = plt.subplots(nrows=1, ncols=2,figsize=(10,4))
n1bins, n2bins = 4, 40
ax[0].hist(x, bins=n1bins)
ax[0].set_ylim(0,250)
ax[0].set_xlabel("Bins", fontsize=16)
ax[0].set_ylabel("Count", fontsize=16)
ax[0].tick_params(axis='both', which='major', labelsize=14)
ax[0].set_title("Histogram: bins=%4d"%(n1bins), fontsize=16)

ax[1].hist(x, bins=n2bins)
ax[1].set_ylim(0,250)
ax[1].set_xlabel("Bins", fontsize=16)
ax[1].set_ylabel("Count", fontsize=16)
ax[1].tick_params(axis='both', which='major', labelsize=14)
ax[1].set_title("Histogram: bins=%4d"%(n2bins), fontsize=16)

plt.savefig("histograms_uniform.png")
plt.tight_layout()
```

Think through the following:

- Though the data is from a uniform distribution, the histogram does not appear flat. Why?
- Every time you run it, the histogram looks slightly different? Why?
- How do the above observations change (if so how) if you had started with more data?

Let us now add and subtract some uniform random numbers:

```
N = 1000
x1 = np.zeros(N)
for n in range(N):
    x1[n] = np.sum(np.random.rand(12,1)) - np.sum(np.random.rand(12,1))
fig, ax = plt.subplots(figsize=(5,5))
ax.hist(x1, 20)
```

What do you observe? How does the resulting histogram change when you change the number of uniform random numbers you add and subtract? Is there a theory that explains your observation?

1.3 Uncertainty in Estimation

Much of what we study in machine learning has to do with estimating parameters of models from a finite set of data. Consider estimating the variance of a uni-variate Gaussian density using samples drawn from it. When we estimate the variance from different sets of samples ("different realizations of a process"), the answer we get each time will be slightly different. But if we had more data, we would expect this variation to be small. Let's see if this is true:

```
MaxTrial = 2000
sampleSizeRange = np.linspace(100, 200, 40)
plotVar = np.zeros(len(sampleSizeRange))
for sSize in range(len(sampleSizeRange)):
    numSamples = int(sampleSizeRange[sSize])
vStrial = np.zeros(MaxTrial)
for trial in range(MaxTrial):
    xx = np.random.randn(numSamples,1)
vStrial[trial] = np.var(xx)
plotVar[sSize] = np.var(vStrial)
fig, ax = plt.subplots(figsize=(4,4))
ax.plot((plotVar))
```

1.4 Bi-variate Gaussian Distribution

We will come across the multi-variate Gaussian distribution, defined in some $d-$ dimensional space quite a lot in this module. We can study some properties of this with $d = 2$, bi-variate, because in two dimensions we can visualize some of these.

```
def gauss2D(x, m, C):
    Ci = np.linalg.inv(C)
    dC = np.linalg.det(C1)
    num = np.exp(-0.5 * np.dot((x-m).T, np.dot(Ci, (x-m))))
    den = 2 * np.pi * dC

    return num/den

def twoDGaussianPlot (nx, ny, m, C):
    x = np.linspace(-5, 5, nx)
    y = np.linspace(-5, 5, ny)
    X, Y = np.meshgrid(x, y, indexing='ij')

    Z = np.zeros([nx, ny])
    for i in range(nx):
        for j in range(ny):
            xvec = np.array([X[i,j], Y[i,j]])
            Z[i,j] = gauss2D(xvec, m, C)

    return X, Y, Z
```

This is a function of two variables. You can plot contours on this function or visualize it as a three dimensional surface plot.

```
# Plot contours
#
nx, ny = 50, 40
m1 = np.array([0,2])
C1 = np.array([[2,1], [1,2]], np.float32)
Xp, Yp, Zp = twoDGaussianPlot (nx, ny, m1, C1)

plt.contour(Xp, Yp, Zp, 5)
```

Draw contours of the following distributions: $\mathcal{N}\left(\begin{bmatrix} 2.4 \\ 3.2 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\right)$, $\mathcal{N}\left(\begin{bmatrix} 1.2 \\ 0.2 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}\right)$

and $\mathcal{N}\left(\begin{bmatrix} 2.4 \\ 3.2 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}\right)$

1.5 Sampling from a multi-variate Gaussian

Suppose we are tasked with drawing several samples from a multi-variate Gaussian density with mean $\boldsymbol{m} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and covariance matrix $C = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. Here is a way to do this using the properties of multi-variate Guassians we have learnt:

- Factorize the covariance matrix into a lower triangular matrix and its transpose: $C = A\,A^T$:

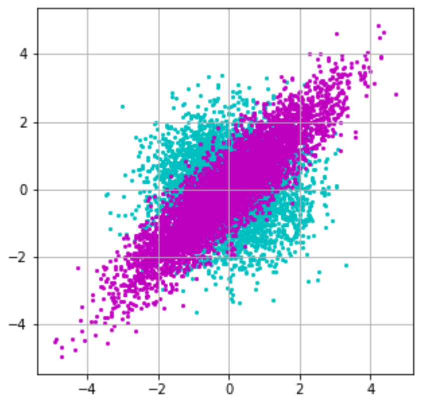
```
C = np.array([[2.0,1.0], [1.0,2]])
A = np.linalg.cholesky(C)
print(A @ A.T)
```

- Generate 5000 bivariate Gaussian random data by `X = np.random.randn(5000,2)` and transform each of the data (rows of X) by $Y = X\,A$

```
X = np.random.randn(1000,2)
Y = X @ A
print(X.shape)
print(Y.shape)
```

- Draw scatter plots of X and Y

```
fig, ax = plt.subplots(figsize=(5,5))
ax.scatter(X[:,0], X[:,1], c="c", s=4)
ax.scatter(Y[:,0], Y[:,1], c="m", s=4)
ax.set_xlim(-6, 6)
ax.set_ylim(-6, 6)
```



1.6 Distribution of Projections

- Construct a vector $\boldsymbol{u} = [\sin\theta \;\; \cos\theta]^T$, parameterized by the variable θ.

```
theta = np.pi/3
u = [np.sin(theta), np.cos(theta)]
print("The vector: ", u)
print("Magnitude : ", np.sqrt(u[0]**2 + u[1]**2))
print("Angle     : ", theta*180/np.pi)
```

- Compute the variance of projected data along this direction

```
yp = Y @ u
print(yp.shape)
print("Projected variance: ", np.var(yp))
```

- Now, using the above write a program that plots the variance of the projected data as you change θ over the range 0 to $2\,\pi$.

```
# Store projected variances in pVars & plot
#
nPoints = 50
pVars = np.zeros(nPoints)
thRange = np.linspace(0, 2*np.pi, nPoints)
for n in range(nPoints):
    theta = thRange[n]
    u = [np.sin(theta), np.cos(theta)]
    pVars[n] = np.var(Y @ u)

fig, ax = plt.subplots(figsize=(5,3))
ax.plot(pVars)
```

  What are the maxima and minima of the resulting plot?

- Compute the eigenvalues an eigenvectors of the covariance matrix C
- Can you see a relationship between the eignevalues and eigenvectors and the maxima and minima of the way the projected variance changes?
- The shape of the graph might have looked sinusoidal for this two dimensional problem. Can you analytically confirm if this might be true?

2 Pereceptron Learning

2.1 Objective

Implementing a linear classifier using the Perceptron algorithm.
The perceptron algorithm computes a linear classifier using a stochastic error correcting learning algorithm. It is simple and has much historic relevance to this subject and makes a good starting point to learn more sophisticated models and algorithms.

2.2 Implementation

Generate 100 samples each from two bi-variate Gaussian densities with distinct means $\boldsymbol{m}_1 = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$ and $\boldsymbol{m}_2 = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, and identical covariance matrix $\boldsymbol{C} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. (Hint: Use material developed in Lab One).

```
NumDataPerClass = 200

# Two-class problem, distinct means, equal covariance matrices
#
m1 = [[0, 5]]
m2 = [[5, 0]]
C  = [[2, 1], [1, 2]]

# Set up the data by generating isotropic Guassians and
# rotating them accordingly
#
A = np.linalg.cholesky(C)

U1 = np.random.randn(NumDataPerClass,2)
X1 = U1 @ A.T + m1

U2 = np.random.randn(NumDataPerClass,2)
X2 = U2 @ A.T + m2
```

The distribution of your data should look like what is shown in Fig. 1. Note the data are linearly separable (*i.e.* a linear class boundary will classify the data correctly).

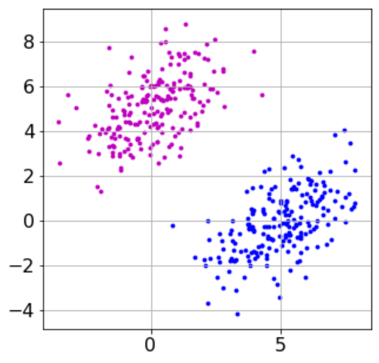


Figure 1: Training Data, sampled from two Bivariate Gaussian Densities

We will now train a perceptron algorithm to classify this data. A perceptron is a linear classifier whose training is done by error correction. If the weights of the perceptron are denoted $\boldsymbol{w}$ and the input features are in vector $\boldsymbol{x}$, a perceptron decision function assigns the data to one class or the other depending on whether $\boldsymbol{w}^T \boldsymbol{x} \lesseqgtr 0$.
The algorithm is as follows:

```
Inputs $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, $\boldsymbol{x}_n \in \mathcal{R}^d$, $y_n \in (-1, +1)$
Initialize weights $\boldsymbol{w}$
Generate index $0 \leq \tau \leq N$ at random
If $(\boldsymbol{w}^T \boldsymbol{x}^{(\tau)} \leq 0)$
.       $\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \alpha\, y^(\tau)\, \boldsymbol{x}^{(\tau)}$
```

Note the scalar product $\boldsymbol{w}^T \boldsymbol{x}^{(\tau)}$ times the target $y^{(\tau)}$ being positive over all the data is our goal. Upon seeing a random data, we only update if this data is misclassified. This is why we refer to this algorithm as a *stochastic error correcting* algorithm. Stochastic because we are lookign at random presentations of data and error correcting because we only update when the current example is misclassified.
We will derive this in a formal setting after we have studied regression, by setting up an error function and optimising it (minimising it) by gradient descent.
Here are snippets of code to help you do this.

1. For simplicity, I have assumed the following (some of which you are free to change and study the effect):
   - There is an equal number of data `NumDataPerClass` in each class
   - We use an equal partition of the data into training and test sets, taken at random.
2. Concatenate data from two classes into one array. (Fig. 1).

```
X = np.concatenate((X1, X2), axis=0)
```

3. Setting up targets (labels): we set $+1$ and -1 as labels to indicate the two classes.

```
labelPos = np.ones(NumDataPerClass)
labelNeg = -1.0 * np.ones(NumDataPerClass)
y = np.concatenate((labelPos, labelNeg))
```

4. Partitioning the data into training and test sets

```
rIndex = np.random.permutation(2*NumDataPerClass)
Xr = X[rIndex,]
yr = y[rIndex]

# Training and test sets (half half)
#
X_train = Xr[0:NumDataPerClass]
y_train = yr[0:NumDataPerClass]
X_test  = Xr[NumDataPerClass:2*NumDataPerClass]
y_test  = yr[NumDataPerClass:2*NumDataPerClass]
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)

Ntrain = NumDataPerClass;
Ntest  = NumDataPerClass;
```

5. Calculating the percentage of correctly classified examples

```
def PercentCorrect(Inputs, targets, weights):
    N = len(targets)
    nCorrect = 0
    for n in range(N):
        OneInput = Inputs[n,:]
        if (targets[n] * np.dot(OneInput, weights) > 0):
            nCorrect +=1
    return 100*nCorrect/N
```

6. Iterative error correcting learning

```
# Perceptron learning loop
#

# Random initialization of weights
#
w = np.random.randn(2)
print(w)

# What is the performance with the initial random weights?
#
print('Initial Percentage Correct: %6.2f' %(PercentCorrect(X_train, y_train, w)))

# Fixed number of iterations (think of better stopping criterion)
#
MaxIter=1000

# Learning rate (change this to see convergence changing)
#
alpha = 0.002

# Space to save answers for plotting
#
P_train = np.zeros(MaxIter)
P_test  = np.zeros(MaxIter)

# Main Loop
#
for iter in range(MaxIter):

    # Select a data item at random
    #
    r = np.floor(np.random.rand()*Ntrain).astype(int)
    x = X_train[r,:]

    # If it is misclassified, update weights
    #
    if (y_train[r] * np.dot(x, w) < 0):
        w += alpha * y_train[r] * x

    # Evaluate trainign and test performances for plotting
    #
    P_train[iter] = PercentCorrect(X_train, y_train, w);
    P_test[iter]  = PercentCorrect(X_test, y_test, w);

print('Percentage Correct After Training: %6.2f  %6.2f'
      %(PercentCorrect(X_train, y_train, w), PercentCorrect(X_test, y_test, w)))
```

7. Plot learning curves

```
fig, ax = plt.subplots(figsize=(6,4))
ax.plot(range(MaxIter), P_train, 'b', label = "Training")
ax.plot(range(MaxIter), P_test, 'r', label = "Test")
ax.grid(True)
ax.legend()
ax.set_title('Perceptron Learning')
ax.set_ylabel('Training and Test Accuracies', fontsize=14)
ax.set_xlabel('Iteration', fontsize=14)
plt.savefig('learningCurves.png')
```

The expected results of training a perceptron might look similar to Fig. 2

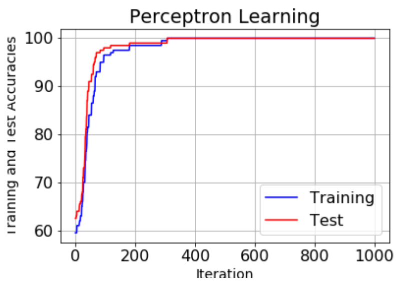


Figure 2: Learning Curves for Classifying two Gaussian Distributed Data

8. The `scikitlearn` package is an excellent source of machine learning algorithms in `Python`. Compare the performance of your perceptron algorithm on the two-class Gaussian dataset with that of the perceptron tool in the `scikitlearn` package. Here is a snippet of code to help you get started:

```
# Scikitlearn can do it for us
#
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
model = Perceptron()
model.fit(X_train, y_train)
yh_train = model.predict(X_train)
print("Accuracy on training set: %6.2f" %(accuracy_score(yh_train, y_train)))

yh_test = model.predict(X_test)
print("Accuracy on test set: %6.2f" %(accuracy_score(yh_test, y_test)))

if (accuracy_score(yh_test, y_test) > 0.99):
    print("Wow, Perfect Classification on Separable dataset!")
```

9. Consider the problem with means at $\boldsymbol{m}_1 = \begin{bmatrix} 2.5 \\ 2.5 \end{bmatrix}$ and $\boldsymbol{m}_2 = \begin{bmatrix} 10.0 \\ 10.0 \end{bmatrix}$ with the covariance matrices equal and the same as before. Does the perceptron as implemented solve this problem? If not what modification is needed to help solve this problem? Hint:

```
O = np.ones((2*NumDataPerClass, 1))
X = np.append(X, O, axis=1)

w = np.random.randn(3)
```

10. Download a two class classification problem from the UCI machine Learning Repository of benchmark datasets `https://archive.ics.uci.edu/ml/index.php` and classify using your own perceptron algorithm. How does the performance compare to any quoted results on this dataset by other researchers? You may need more python tools to read and manipulate downloaded data (*e.g.* `Pandas`).

3 Linear Regression & Regularization

3.1 Objectives

- Implementing linear regression
- Regularization using quadratic and sparsity-inducing penalties
- Implementing sparse regression on a realistic problem in chemoinformatics

3.2 Linear Least Squares Regression:

We will work with the `Diabetes` dataset from the `UCI Machine Learning` repository [1] taken from the package `sklearn`. Load the data and inspect the features and targets. It is usually a good idea to plot a few histograms of the targets and pair-wise scatters of the features in any new problem you are tasked to solve.

- Implement a linear predictor that is solved by the pseudo-inverse method:

$$\boldsymbol{w} = \left(X^t X\right)^{-1} X^t \boldsymbol{t},$$

  where X is the $N \times d$ input matrix and $\boldsymbol{t}$ is the $N \times 1$ vector of responses (or targets).

- Solve the same problem using the linear model from `sklearn` and compare the results.

3.3 Regularization

Tikhonov regularization (or L_2) minimizes the mean squared error with a quadratic penalty on the weights:

$$\min_{\boldsymbol{w}} ||\boldsymbol{t} - X\boldsymbol{w}||_2^2 + \gamma\, ||\boldsymbol{w}||_2^2$$

Derive and implement a regularized regression. Show, using two bar graphs of the weights side by side to the same scale, how the two solutions differ.

3.4 Sparse Regression

L_1 regularization is a method for achieving *sparse* solutions [2]. It minimizes:

$$\min_{\boldsymbol{w}} ||\boldsymbol{t} - X\boldsymbol{w}||_2^2 + \gamma\, ||\boldsymbol{w}||_1$$

We will use the `sklearn` package for implementing its solution. For the Diabetes problem considered above, solve the `lasso` problem and plot the resulting weights as a bar graph. Observe how the number of non-zero weights change with the regularization parameter γ. Your comparisons should look similar to Fig. 3. In each of these cases, compare the prediction errors. In the case of the sparse regression, would you say the features with nonzero weights are more meaningful (to answer, you have to find the source of the data and look at the variables)?

Regularization Path

When implementing the *lasso* it is convenient to study the regularization path (Fig. 4, Image taken from `https://scikit-learn.org/stable/auto_examples/linear_model/plot_lasso_lars.html`) Implement and study the regularization path for the six-variable illustrative example considered in [2].

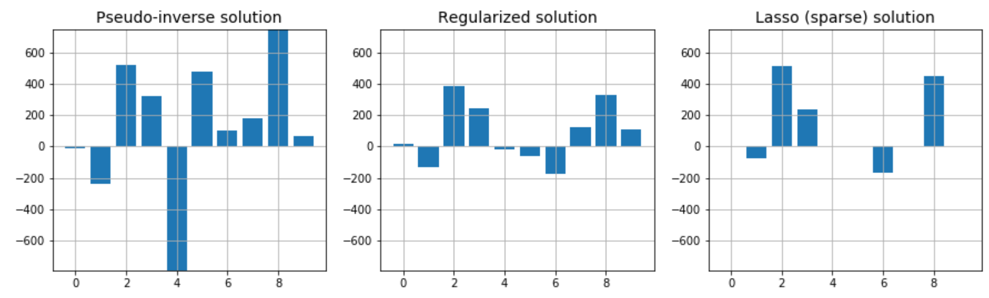


Figure 3: Solutions of Linear and Regularized Regressions

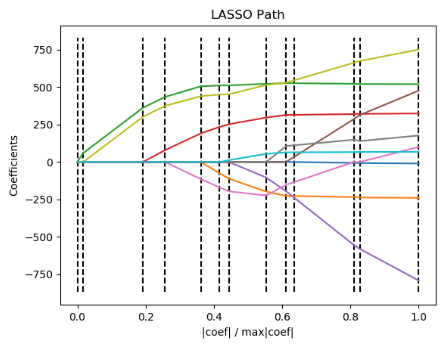


Figure 4: Regularization Path: How regression coefficients change with hyperparameter.

3.5 Solubility Prediction:

We will now look at a large problem of predicting solubility of chemical compounds from features derived from their molecular structure. Predicting function from structural variables is an important problem because it is easy to define and synthesize small chemical compounds, but very expensive to test them experimentally. Hence the step known as *in silico* screening is increasingly popular. The dataset we will use is from Huuskonen *et al.* [3] and the problem has also been considered recently in Pirashvili *et al.* [4] using more sophisticated machinery. Have a skim-read through the introductory and results sections of these papers.
Data used in [3] with several additional features and more compounds is available in the excel spread sheet `Husskonen_Solubility_Features.xlsx`.

- Load the data, split into training and test sets, implement a linear regression and plot the predicted solubilities against the true solubilities on the training and test sets. To facilitate comparison, draw the two scatter plots side by side to the same scale on both axes.
- Implement a `lasso` regularized solution and plot graphs of how the prediction error (on the test data) and the corresponding number of non-zero coefficients change with increasing regularization.
- If you were to select the top ten features to predict solubility, what would they be[1]? How good is the prediction accuracy with these slected features when compared to using all the features and a quadratic regularizer?

[1]Of course, we will not know enough chemistry to interpret these, but in a real-world setting, we will be working with a friend in the School of Chemistry!

- Are you able to make any comment comparing your results to those claimed in [3] or [4]?

4 Radial Basis Functions (RBF)

4.1 Objectives

- To implement a Radial Basis Functions model on a regression problem and compare its performance with linear regression.
- To use ten-fold cross validation to quote uncertainty in empirical results when comparing the percormances of two machine learning approaches.

4.2 Background

The RBF model is given by

$$g(\boldsymbol{x}) = \sum_{j=1}^{M} \lambda_j \, \phi(||\boldsymbol{x} - \boldsymbol{m}_j|| \, / \, \sigma).$$

The model has a nonlinear part (with $\boldsymbol{m}_j$ and σ as parameters within a basis function $\phi(.)$) and a linear part with parameters λ_j. In problems where we can make sensible choices of the nonlinear part, the learning problem reduces to a linear problem in the λ_js.
Constructing an $N \times M$ *design matrix* U with terms $u_{ij} = \phi(||\boldsymbol{x}_i - \boldsymbol{x}_j|| \, / \, \sigma$, and the targets in an $N \times 1$ vector $\boldsymbol{f}$, the least squares solution to estimate λs is similar to linear regression: $\boldsymbol{l} = \left(U^t \, U\right)^{-1} U^t \, \boldsymbol{f}$, where $\boldsymbol{l}$ is an $M \times 1$ vector containing the unknown λs.

4.3 Tasks

1. Study the skeleton implementation of a Gaussian RBF model given in the Appendix.
2. Make the following improvements to the given implementation:
   - Normalize each feature of the input data to have a mean of 0 and standard deviation of 1.
   - The width parameter of the basis functions σ is set to be the distance between two randomly chosen points. Could this sometimes cause an error? Change it to be the average of several pairwise distances.
   - The locations of the M basis functions $\boldsymbol{m}_j$ are set at random points in the input space. Cluster the data using K-means clustering (with $K = M$) and set the basis function locations to the cluster centres.
   - Split the data into training and test sets, estimate the model on the training set and note the test set performance.
3. Implement ten-fold cross validation, where you split the data into ten parts, train on nine tenths of the data and test on the held out tenth set, repeating the process ten times.
4. Display the distributions of test set results for the RBF and Linear Regression Models as boxplots side by side.
5. Compare your implementations with `sklearn`'s inbuilt RBF model.

5 Discriminant Functions

5.1 Objectives

1. Class boundaries and posterior probabilities of Gaussian classifiers
2. Fisher Linear Discriminant Analysis
3. Receiver Operating Characteristic (ROC) Curve
4. Mahalanobis distance

5.2 Class Boundaries and Posterior Probabilities

Consider two-class classification problems in two dimensions in which features of the two classes are Gaussian distributed:

- $\boldsymbol{m}_1 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ and $\boldsymbol{m}_1 = \begin{pmatrix} 3 \\ 2.5 \end{pmatrix}$, $\boldsymbol{C}_1 = \boldsymbol{C}_2 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, $P_1 = P2 = 0.5$;
- $\boldsymbol{m}_1 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ and $\boldsymbol{m}_2 = \begin{pmatrix} 3 \\ 2.5 \end{pmatrix}$, $\boldsymbol{C}_1 = \boldsymbol{C}_2 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, $P_1 = 0.7$, $P_2 = 0.3$;
- $\boldsymbol{m}_1 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ and $\boldsymbol{m}_2 = \begin{pmatrix} 3 \\ 2.5 \end{pmatrix}$, $\boldsymbol{C}_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$, $\boldsymbol{C}_2 = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$, $P_1 = P2 = 0.5$.

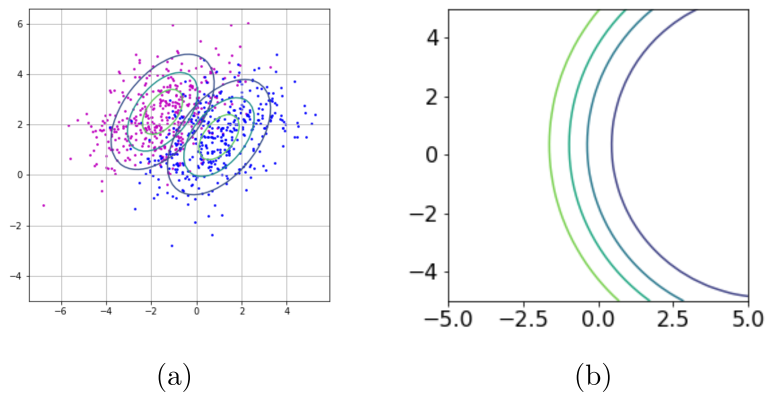


Figure 5: (a) Example of probability densities and data; (b) Example of contours on the posterior probability $P[\omega_1 \,|\, \boldsymbol{x}]$. Note these are illustrations and (b) is not the posterior probability for the problem in (a)!

In each of the above cases, plot contours on the likelihoods of the two classes, a scatter of 200 data sampled from each of the classes and contours on the posterior probability of one of the classes. Discuss (in your report) if what you plot is consistent with your expectation from analytical derivation of the class boundaries. Note you are free to change the means and covariance matrices given above to illustrate the differences we are learning about. Examples of the graphs to aim for are given in Fig. 5

5.3 Fisher LDA and ROC Curve

Define a two class pattern classification problem in two dimensions, in which the two classes are Gaussian distributed with means $\boldsymbol{m}_1 = [0 \;\; 3]^t$ and $\boldsymbol{m}_1 = [3 \;\; 2.5]^t$, and have a common covariance matrix

$$\boldsymbol{C}_1 = \boldsymbol{C}_2 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

1. Plot contours on the two densities.
2. Draw 200 samples from each of the two distributions and plot them on top of the contours.
3. Compute the Fisher Linear Discriminant direction using the means and covariance matrices of the problem, and plot the discriminant direction: $\boldsymbol{w}_F = (\boldsymbol{C}_1 + \boldsymbol{C}_2)^{-1} (\boldsymbol{m}_1 - \boldsymbol{m}_2)$
4. Project the data onto the Fisher discriminant directions and plot histograms of the distribution of projections (an example of this is in Fig. 6(a);
5. Compute and plot the Receiver Operating Characteristic (ROC) curve, by sliding a decision threshold, and computing the True Positive and False Positive rates (see code snippet in Appendix and example of an ROC curve in Fig. 6(b).
6. Compute the area under the ROC curve (Hint: try `numpy.trapz`)
7. For a suitable choice of decision threshold, compute the classification accuracy.
8. Plot the ROC curve (on the same scale) for
   - A random direction (instead of the Fisher discriminant direction).
   - Projections onto the direction connecting the means of the two classes.

   Compute the area under the ROC curve (AUC) for these two cases. Your report should explain what the precise statistical interpretation of AUC is and when it is used.

5.4 Mahalanobis Distance

Using a suitable classification problem (two-class in two dimensions, adapted from one of the above examples), illustrate the difference between a distance-to-mean classifier and a Mahalanobis distance-to-mean classifier.

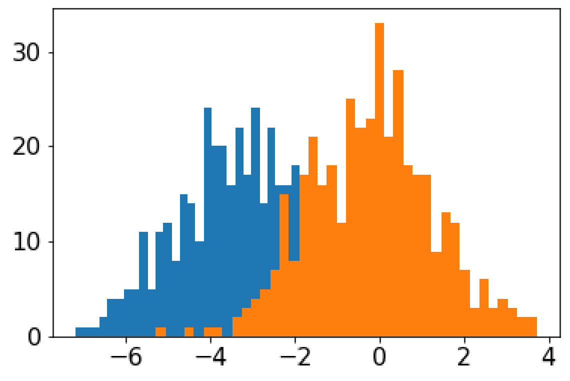
References

[1] K. Bache and M. Lichman, "UCI machine learning repository." `http://archive.ics.uci.edu/ml`, 2013.

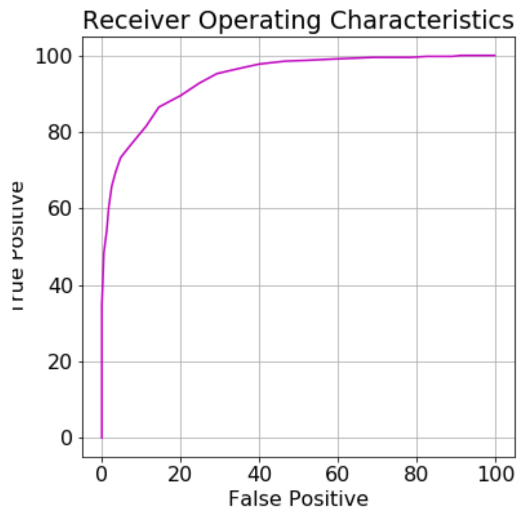
[2] T. Hastie, R. Tibshirani, and M. Wainwright, *Statistical Learning with Sparsity: The Lasso and Generalizations.* Chapman & Hall/CRC, 2015.

[3] J. Huuskonen, M. Salo, and J. Taskinen, "Aqueous solubility prediction of drugs based on molecular topology and neural network modeling," *Journal of Chemical Information and Computer Sciences*, vol. 38, no. 3, pp. 450–456, 1998.

[4] M. Pirashvili, L. Steinberg, B. G. F., M. Niranjan, J. G. Frey, and J. Brodzki, "Improved understanding of aqueous solubility modeling through topological data analysis," *Journal of Cheminformatics*, vol. 10, no. 1, p. 54, 2018.



(a)



(b)

Figure 6: (a) Histograms of projections of the two classes onto the Fisher discriminant direction; (b) An ROC Curve

Appendix: Snippets of Code

1. Linear regression on diabetes dataset

```
from sklearn import datasets
from sklearn.linear_model import LinearRegression

# Load data, inspect and do exploratory plots
#
diabetes = datasets.load_diabetes()

X = diabetes.data
t = diabetes.target

# Inspect sizes
#
NumData, NumFeatures = X.shape
print(NumData, NumFeatures)     # 442 X 10
print(t.shape)                  # 442

# Plot and save
#
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 4))
ax[0].hist(t, bins=40)
ax[1].scatter(X[:,6], X[:,7], c='m', s=3)
ax[1].grid(True)
plt.tight_layout()
plt.savefig("DiabetesTargetAndTwoInputs.jpg")
```

2. Comparing pseudo-inverse solution to `sklearn` output

```
# Linear regression using sklearn
#
lin = LinearRegression()
lin.fit(X, t)
th1 = lin.predict(X)

# Pseudo-incerse solution to linear regression
#
w = np.linalg.inv(X.T @ X) @ X.T @ t
th2 = X @ w

# Plot predictions to check if they look the same!
#
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,5))
ax[0].scatter(t, th1, c='c', s=3)
```

3. Tikhanov (quadratic) Regularizer

```
gamma = 0.5
wR = np.linalg.inv(X.T @ X + gamma*np.identity(NumFeatures)) @ X.T @ t

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8,4))
ax[0].bar(np.arange(len(w)), w)

plt.savefig("LeastSquaresAndRegularizedWeights.jpg")
```

4. Sparsity inducing (lasso) regularizer

```
from sklearn.linear_model import Lasso
ll = Lasso(alpha=0.2)
ll.fit(X, t)
th_lasso = ll.predict(X)

fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(15,4))
ax[0].bar(np.arange(len(w)), w)
#
#...
#
plt.savefig("solutions.png")
```

5. Lasso Regularization path on a synthetic example (Set up data):

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import lasso_path
from sklearn import datasets

# Synthetic data:
# Problem taken from Hastie, et al., Statistical Learning with Sparsity
# Z1, Z2 ~ N(0,1)
# Y = 3*Z1 -1.5*Z2 + 10*N(0,1) Noisy response
# Noisy inputs (the six are in two groups of three each)
# Xj= Z1 + 0.2*N(0,1) for j = 1,2,3, and
# Xj= Z2 + 0.2*N(0,1) for j = 4,5,6.

N = 100
y = np.empty(0)
X = np.empty([0,6])
for i in range(N):
    Z1= np.random.randn()
    Z2= np.random.randn()
    y = np.append(y, 3*Z1 - 1.5*Z2 + 2*np.random.randn())
    Xarr = np.array([Z1,Z1,Z1,Z2,Z2,Z2])+ np.random.randn(6)/5
    X = np.vstack ((X, Xarr.tolist()))
```

6. Lasso Regularization path on a synthetic example (Regression and paths):

```
# Compute regressions with Lasso and return paths
#
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, fit_intercept=False)

# Plot each coefficient
#
fig, ax = plt.subplots(figsize = (8,4))
for i in range(6):
    ax.plot(alphas_lasso, coefs_lasso[i,:])

ax.grid(True)
ax.set_xlabel("Regularization")
ax.set_ylabel("Regression Coefficients")
```

7. Predicting Solubility of Chemical Compounds

```
sol = pd.read_excel("Husskonen_Solubility_Features.xlsx", verbose=False)
print(sol.shape)
print(sol.columns)

t = sol["LogS.M."].values
fig, ax = plt.subplots(figsize=(4,4))
ax.hist(t, bins=40, facecolor='m')
ax.set_title("Histogram of Log Solubility", fontsize=14)
```

```
X = sol[colnames[5:len(colnames)]]
N, p = X.shape

print(X.shape)
print(t.shape)

# Split data into training and test sets
#
from sklearn.model_selection import train_test_split
X_train, X_test, t_train, t_test = train_test_split(X, t, test_size=0.3)

# Regularized regression
#
gamma = 2.3
w = np.linalg.inv(X_train.T @ X_train + gamma*np.identity(p)) @ X_train.T @ t_train
th_train = X_train @ w.to_numpy()
th_test  = X_test @ w.to_numpy()

# Plot training and test predictions
#
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,4))
ax[0].scatter(t_train, th_train, c='m', s=3)

# ... plots
#
# Over to you for implementing Lasso
```

Skeleton Implementation of RBF

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

from sklearn import datasets
from sklearn.linear_model import LinearRegression

diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

def gaussian(x, u, sigma):
    return(np.exp(-0.5 * np.linalg.norm(x-u) / sigma))

N, p = X.shape
print(N, p)

# Space for design matrix
#
M = 200
U = np.zeros((N,M))

# Basis function locations at random
#
C = np.random.randn(M,p)

# Basis function range as distance between two random data
#
x1 = X[np.floor(np.random.rand()*N).astype(int),:]
x2 = X[np.floor(np.random.rand()*N).astype(int),:]
sigma = np.linalg.norm(x1-x2)

# Construct the design matrix
#
for i in range(N):
    for j in range(M):
        U[i,j] = gaussian(X[i,:], C[j,:], sigma)

# Pseudo inverse solution for linear part
#
l = np.linalg.inv(U.T @ U) @ U.T @ y

# Predicted values on training data
#
yh = U @ l
fig, ax = plt.subplots(figsize=(3,3))
ax.scatter(y, yh, c='m', s=3)
ax.grid(True)
ax.set_title("Training Set", fontsize=14)
ax.set_xlabel("True Target", fontsize=12)
ax.set_ylabel("Prediction", fontsize=12)
```

Appendix: Snippets of code

1. To compute the posterior probability (Note: `gauss2D` was used in previous Labs.)

```
def posteriorPlot(nx, ny, m1, C1, m2, C2, P1, P2):
    x = np.linspace(-5, 5, nx)
    y = np.linspace(-5, 5, ny)
    X, Y = np.meshgrid(x, y, indexing='ij')

    Z = np.zeros([nx, ny])
    for i in range(nx):
        for j in range(ny):
            xvec = np.array([X[i,j], Y[i,j]])
            num = P1*gauss2D(xvec, m1, C1)
            den = P1*gauss2D(xvec, m1, C1) + P2*gauss2D(xvec, m2, C2)
            Z[i,j] = num / den
    return X, Y, Z
```

2. To compute the Fisher discriminant direction, project data and plot histograms of the two projected classes:

```
Ci = np.linalg.inv(2*C)
uF = Ci @ (m2-m1)

yp1 = Y1 @ uF
yp2 = Y2 @ uF

matplotlib.rcParams.update({'font.size': 16})
plt.hist(yp1, bins=40)
plt.hist(yp2, bins=40)
plt.savefig('histogramprojections.png')
```

3. To compute and plot a ROC curve:

```
# Define a range over which to slide a threshold
#
pmin = np.min( np.array( (np.min(yp1), np.min(yp2) )))
pmax = np.max( np.array( (np.max(yp1), np.max(yp2) )))
print(pmin, pmax)

# Set up an array of thresholds
#
nRocPoints = 50;
thRange = np.linspace(pmin, pmax, nRocPoints)
ROC = np.zeros( (nRocPoints, 2) )

# Compute True Positives and False positives at each threshold
#
for i in range(len(thRange)):
    thresh = thRange[i]
    TP = len(yp2[yp2 > thresh]) * 100 / len(yp2)
    FP = len(yp1[yp1 > thresh]) * 100 / len(yp1)
    ROC[i,:] = [TP, FP]

# Plot ROC curve
#
fig, ax = plt.subplots(figsize=(6,6))
ax.plot(ROC[:,1], ROC[:,0], c='m')
ax.set_xlabel('False Positive')
ax.set_ylabel('True Positive')
ax.set_title('Receiver Operating Characteristics')
ax.grid(True)
plt.savefig('rocCurve.png')
```