UNIVERSITY COLLEGE LONDON
DEPARTMENT OF SPACE AND CLIMATE PHYSICS

Candidate Code: HYXC3

Programme Title: MSc Scientific Computing

Module Code: SPCE0038

Module Title: Machine Learning with Big Data

End Assessment

In submitting this coursework, I assert that the work presented is entirely my own except where properly marked and cited.

Date of Submission:	11/05/20

Question 1

1(a)

With reference to the diagram of the basic *logistic unit* on the following page:

The input vector $\mathbf{x}$ is the input to the *logistic unit*.

Each input x_i has an associated weight θ_i. The weights are set to a random value prior to *training*. The *training* process determines these weights.

The product of each input x_i and weight θ_i is summed to produce a weighted sum z.

The output, $h_\theta(x)$, is the the non-linear *activation function*, h, applied to z.

1(b)

Consider the diagram of Question 1(a).

The weighted sum of the inputs, z, is:

$$z = \sum_{j=1}^{n} \theta_j x_j = \theta^T x \tag{1}$$

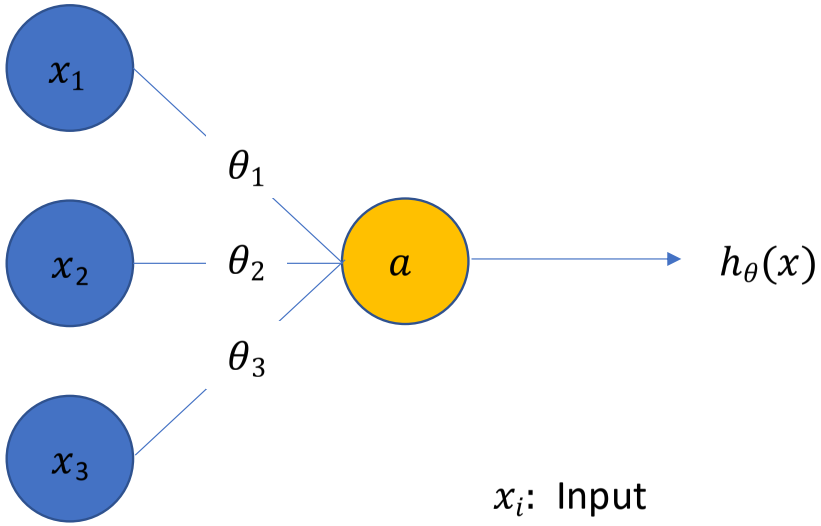
where x_i is the i^{th} element of input vector $\mathbf{x}$ of length n, and θ_i is the associated *weight*.

And, the output from the *logistic unit*, $h_\theta(x)$, is:

$$h_\theta(x) = h(z) \tag{2}$$

where h is a non-linear *activation function*.

Question 1(a) – Basic Logistic Unit



x_i: Input

θ_i: Weight

Weighted Sum: $z = \sum_{j=1}^{n} \theta_j x_j = \theta^T x$

Activation: $a = h(z)$

1(c)

See diagram on following page.

1(d)

Consider the diagram of Question 1(c).

Firstly, consider the data transformation from the input vector $\mathbf{x}$ to the hidden layer. We now need two indices, one for the input vector elements, and one for the *hidden layer* nodes. We will use i for the input vector index, and j for the *hidden layer* nodes.

The weighted sum of the inputs at the j^{th} *hidden layer* node is:

$$z_j = \sum_{i=1}^{n} \theta_{ij} x_i \tag{3}$$

where θ_{ij} is the weight between input element i and *hidden layer* node j, and n is the length of the input vector $\mathbf{x}$.

Secondly, the output from each *hidden layer* node is the non-linear *activation function*, h, applied to each z_j:

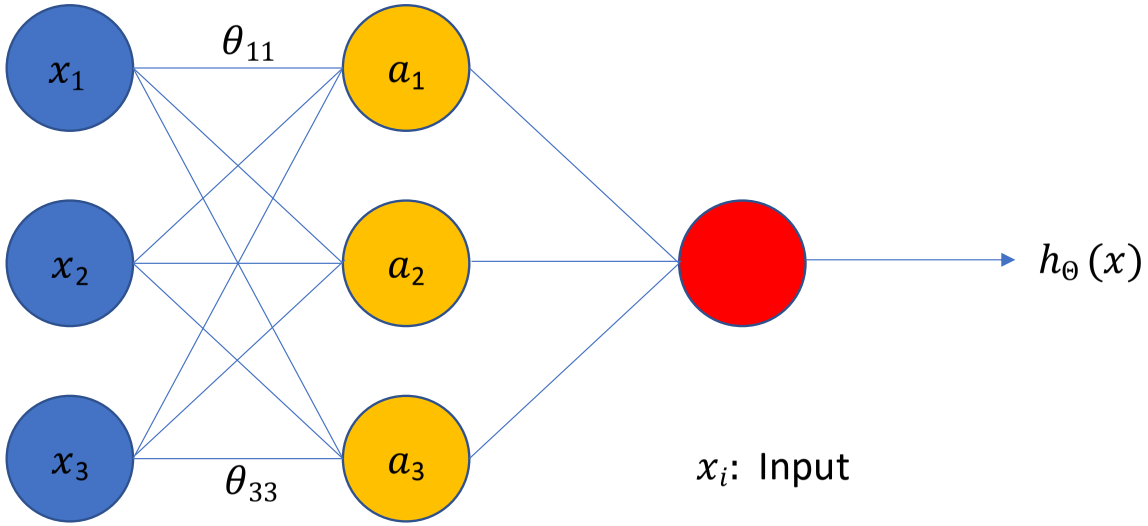
$$h_{\theta j}(x) = h(z_j) \tag{4}$$

And finally, the output from the whole network, $h_\Theta(x)$ is the sum of the *hidden layer* outputs:

$$h_\Theta(x) = \sum_{j=1}^{m} h_{\theta j}(x) \tag{5}$$

where m is the number of *hidden layer* nodes.

Question 1(c) – Fully Connected, Feed Forward, Artificial Neural Network



Input Layer Logistic Units

Hidden Layer Logistic Units

Output Node

x_i: Input

θ_{ij}: Weight, e.g. $\theta_{11}\,\theta_{33}$

Weighted Sums: $z_j = \sum_{i=1}^{n} \theta_{ij} x_i$

Activations: $a_j = h(z_j)$

1(e)

The cost function typically used to train neural networks for regression problems is *mean square error*:

$$MSE(\Theta) = \frac{1}{m} \sum_i \sum_j (p_j^{(i)} - y_j^{(i)})^2 \tag{6}$$

The cost function typically used to train neural networks for classification problems is *cross-entropy*:

$$C(\Theta) = -\frac{1}{m} \sum_i \sum_j y_j^{(i)} \log(p_j^{(i)}) \tag{7}$$

1(f)

Artificial Neural Networks (ANNs) are described as *shallow* or *deep*, and *wide* or *narrow*. *shallow* or *deep* refers to the number of layers in the network, and *wide* or *narrow* refers to the number of nodes in each layer.

The *credit assignment path*, the CAP, of a neural network is a measure of the number of data transformations that occur as data passes through the network. For *feed-forward* networks the CAP is the number of *hidden layers* plus one.

A *deep* neural network is generally considered to be a network with multiple layers and a CAP > 2.

1(g)

The *universal approximation theorem* states that, with appropriate parameters, single hidden layer feed-forward neural networks are *universal approximators*. This means they can represent any continuous function. However, this requires an exponentially larger number of hidden layer nodes. And, training will not necessarily determine the parameters.

Deep networks provide a powerful representational framework because they have the potential to be *universal approximators*, but with a limited width of hidden nodes. This makes the implementation of *universal approximators* more feasible.

Question 2

2(a)

Gradient Descent algorithms attempt to find the parameters θ which minimise the cost function $C(\theta)$ over the *training set* using an iterative process:

$$\theta^{i+1} = \theta^i - \alpha \nabla_\theta C(\theta) \tag{8}$$

where α is the *learning rate*.

Batch Gradient Descent uses the entire *training set* at each iteration to calculate the gradient partial derivatives, $\nabla_\theta C(\theta)$. This produces accurate values for the partial derivatives, but is potentially slow for large *training sets*.

Stochastic Gradient Descent uses a random sub-set of the *training set* at each iteration to calculate the gradient partial derivatives, $\nabla_\theta C(\theta)$. This is faster than *Batch Gradient Descent*, but can produce erratic values for the partial derivatives.

For *convex* cost functions *Batch Gradient Descent* will always converge to the *local minima* which is also the *global minima*. This is not the case for *non-convex* cost functions, where *local minima* are not necessarily *global minima*. A bad choice of θ^0 may result in *Batch Gradient Descent* getting "stuck" in a *local minima*. Because the gradient partial derivatives of *Stochastic Gradient Descent* are erratic, this provides a mechanism of "jumping out of" a *local minima* and improves the probability of finding the *global minima*.

2(b)

When attempting to find the minimum of a cost function using *Stochastic Gradient Descent*, the iterative process "jumps" around the minimum, and it is difficult to determine when a minimum has been reached. For this reason alternative optimisation algorithms are typically considered for training.

2(c)

Consider the iterative minimisation process:

$$\theta^{i+1} = \theta^i - \alpha \nabla_\theta C(\theta) \tag{9}$$

where α is the *learning rate*.

At each step the process "advances" towards the minimum by the step size $-\alpha \nabla_\theta C(\theta)$, which uses the *current* gradient.

The *Momentum Optimisation* algorithm introduces the idea of including *previous* gradients into the step size. At each step the *current* gradient is summed into a momentum term m, which includes *previous* gradients (remember we use the negative gradient):

$$m^{i+1} = \beta m^i - \alpha \nabla_\theta C(\theta) \tag{10}$$

The β parameter is used to prevent the momentum getting too large, and is set between 0 and 1, typically 0.9.

The momentum m is then used to update θ:

$$\theta^{i+1} = \theta^i + m \tag{11}$$

The result is that we now have an *acceleration* towards the minimum.

This algorithm may result in an overshoot and oscillation before stabilising at the minimum.

2(d)

Consider the momentum update equation of the *Momentum Optimisation* algorithm:

$$m^{i+1} = \beta m^i - \alpha \nabla_\theta C(\theta) \tag{12}$$

This uses the gradient at the current value of θ.

However, the momentum m is pointing in the general direction of the cost function minimum, so it makes sense to use a point further along in this direction, $\theta + \beta m^i$, to calculate the gradient, as this will be already closer to the minimum:

$$m^{i+1} = \beta m^i - \alpha \nabla_\theta C(\theta + \beta m^i) \tag{13}$$

This is called the *Nesterov Accelerated Gradient* algorithm.

θ is updated as per the *Momentum Optimisation* algorithm:

$$\theta^{i+1} = \theta^i + m \tag{14}$$

The *Nesterov Accelerated Gradient* algorithm is an enhancement to the *Momentum Optimisation* algorithm, which can result in a significant increase in speed, and can reduce the overshoot and oscillations previously mentioned.

2(e)

For optimisation problems where the *objective function* has circular contours the negative gradient always points towards the *objective function* minimum. This means that for an iterative process such as *Gradient Descent* the steps are always in the direction of the minimum. This produces swift convergence to the minimum.

For an elongated *objective function*, with steep sides, a narrow valley, and a bend in the valley, the negative gradient will not point towards the *objective function* minimum if the current value of θ is at the opposite end of the valley. Large steps will be taken offset from the minimum direction. This will reduce the convergence rate. And where the negative gradient is the steepest is where the largest offset steps will be taken. This phenomena is observed for *objective functions* with non-circular contours.

The *AdaGrad* algorithm addresses this problem by reducing the learning rate where the negative gradient is steepest. This reduces the step size away from the minima. Where the negative gradient is less steep the *learning rate* is

reduced by a lesser amount, preserving the step size towards the minimum. This is achieved by introducing a gradient squared term.

The *AdaGrad* algorithm uses what is called an *adaptive learning rate*. An advantage of this during *training* is that less tuning of the learning rate α is required, as this is now "self-tuning".

There is a problem though that the *learning rate* may be reduced too much, and the minimisation of the *cost function* comes to a stop prior to reaching the minimum. This is the reason why *AdaGrad* is not generally used to train deep *neural networks*.

2(f)

As previously stated, a problem with the *AdaGrad* algorithm is that the *learning rate* may be reduced too much, and the minimisation of the *cost function* comes to a stop prior to reaching the minimum.

The *RMSProp* algorithm address this problem by adding an exponential decay to the gradients. This means that only the most recent gradients are used in the minimisation process, and not all of the gradients.

RMSProp performs better than *AdaGrad* in most *training* scenarios.

2(g)

The *Adam* optimisation algorithm, *adaptive moment estimation*, combines the following ideas from previously discussed algorithms.

From *momentum* optimisation, *Adam* borrows the idea of an exponential decay of previous gradients.

And from *RMSProp*, *Adam* borrows the idea of an exponential decay of previous gradients squared.

Because of its performance, *Adam* is a standard algorithm for training deep neural networks.

2(h)

With reference to the diagram of *Dropout Regularisation* on the following page.

Deep neural networks can be prone to overfitting. *Regularisation* is a method of avoiding overfitting, and *Dropout Regularisation* is one such method.

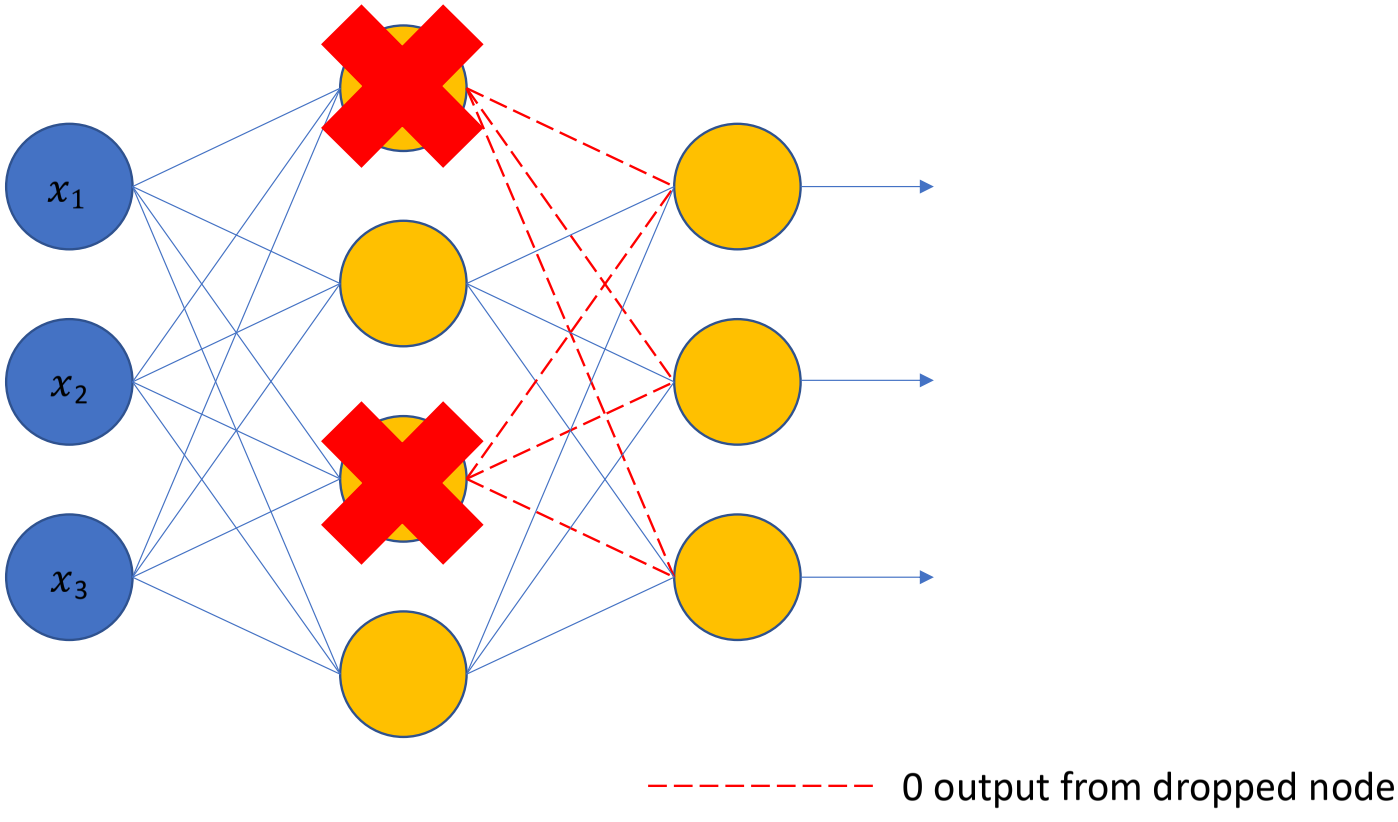
In *Dropout Regularisation*, at every *training* iteration every node, including the input nodes but excluding the output nodes, has a probability p of being *dropped*. When *dropped* the output from the node is zero. The *dropout rate* p is typically 10% to 50%. On completion of training no nodes are *dropped*.

Surprisingly, this method works, even though it would appear that useful information is being destroyed/ignored!

One way of looking at it is that the remaining nodes (the nodes that haven't been *dropped*) have to work harder. And somehow have an enhanced memory or functional ability after training. The reliance on neighbouring nodes is reduced, and each node has to be more effective.

There is one factor that needs to be considered when moving from *training* to *testing*. During *training*, with a *dropout rate* of 50%, say. There will be approximately 50% less inputs nodes. Therefore during *testing*, when no nodes are dropped, the input signal will be twice as strong. To account for this, each node's input weights are factored by $1 - p$.

Question 2(h) – Dropout Regularisation



Question 3

3(a)

The *knowledge* based approach to *artificial intelligence* is the encoding of knowledge as rules, and the subsequent use of logistical inference to make deductions from these rules.

One problem with this approach is that it is difficult to encode all knowledge as rules, for example intuition is difficult to encode as a rule.

Another problem is that the rules need to be added/amended/removed as the knowledge base changes.

3(b)

The *machine learning* approach to *artificial intelligence* is to enable a machine to *learn* from data.

In this approach a machine is not given a set of rules. Instead, a machine is *trained* on data from which patterns and relationships in the data emerge.

This approach is more flexible than the *knowledge* based approach.

3(c)

Features are the elements of data which are relevant to a *machine learning* task.

The *machine learning* approach of *feature engineering* ensures the appropriate *features* are used for a *machine learning* task. This consists *feature selection* and *feature extraction*.

Feature selection is simply selecting the most appropriate features.

Feature extraction is the generation of new features from existing ones. For example, creating a new feature x^2 from the exisiting feature x may make a data set *linearly separable* when the original data set was not.

3(d)

Given *training* data with associated *labels*, e.g. this image (data) is an elephant (*label*), *Supervised Learning* predicts new outputs from new inputs.

Unsupervised Learning is the determining of relationships within unlabelled data, e.g. the clustering of data.

Reinforcement Learning is the learning of a best action based upon being *rewarded* for choosing the best action.

3(e)

In *supervised learning*, a *regression* problem is one which requires the output to be a numerical value, e.g. a share price.

In *supervised learning*, a *classification* problem is one which requires the output to be a *class label*, e.g. this image of a digit is of the number (*class*) 9.

3(f)

TODO

Question 4

4(a)

TensorFlow uses the idea of describing calculation in terms of *computational graphs*. These *computational graphs* are then *executed* to perform the calculation.

The *computational graphs* are created by users and can be broken up into chunks. These chunks can then be *executed* on CPUs and GPUs using highly optimised C++ code. This chunking of calculation also facilitates *parallel* execution.

4(b)

In *TensorFlow* a *tensor* is usually a multi-dimensional array.

A TensorFlow *Constant* type is, as the name would suggest, a *tensor* whose values cannot be changed.

A TensorFlow *Variable* type is a *tensor* whose values can be changed.

Examples of how to create both types of *tensors* are below:

```
import  tensorflow as tf

unchangeable_tensor = tf.constant([0, 1, 2, 3, 4])

changeable_tensor = tf.Variable([0, 1, 2, 3, 4])
```

Any operation that attempts to change the values of *unchangeable_tensor* will result in an *exception* being thrown.

NB "Variable" has a capitalised first letter because *Variable* is a constructor, whereas *constant* is not.

4(c)

A TensorFlow *Placeholder* is a location where a *tensor* will be *fed* at a later time. This is useful in loop constructs where the *tensor(s)* will change with each loop iteration.

For example, in the code of Question 4f (extract below) the tensors *X_batch* and *y_batch* change on each loop iteration, and are *fed* to *sess.run()* via the dictionary *feed_dict*.

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")

...

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    best_theta = theta.eval()
```

4(d)

Although it would possible to calculate the gradients at each node in a *computational graph*, this would be tedious and for large *neural networks* almost impossible. This is where TensorFlow's Autodiff comes in.

For example, *Autodiff* can be used to calculate the gradients of a function with respect to θ, and subsequently used, as follows:

```
gradients = tf.gradients(function, [theta])[0]

training_op = tf.assign(theta, theta - learning_rate * gradients)
```

The advantage of using *Autodiff* is that it makes it possible to determine the gradients of large networks that would be almost impossible to determine otherwise.

4(e)

The machine learning problem the code of Question 4(e) is trying to solve is *Linear Regression*.

The optimisation algorithm being used is *Gradient Descent*.

4(f)

See the Python Jupyter Notebook on following 3 pages.

Question 4f

NB numpy had to be downgraded from 1.17.0 to 1.16.6 to remove an incompatibility with tensorflow 1.14, which manifests itself as a FutureWarning.

In [1]:

```
# Import Numpy and Tensorflow.

import numpy as np
import tensorflow as tf

# Import California housing data and StandardScaler from Scikit-Learn.

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler
```

In [2]:

```
# For reproducibility.

def reset_graph(seed=1):
    np.random.seed(seed)
    tf.reset_default_graph()
    tf.set_random_seed(seed)
```

In [3]:

```
# Load California housing data.

housing = fetch_california_housing()
m, n = housing.data.shape

scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]

housing_data_target = housing.target.reshape(-1, 1)
```

In [4]:

```
# Setup computational graph using placeholders.

reset_graph ()

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")

theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=1), name="th
eta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

In [5]:

```
# Define fetch_batch() for Mini-Batch Gradient Descent.

def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)
    indices = np.random.randint(m, size=batch_size)
    X_batch = scaled_housing_data_plus_bias[indices]
    y_batch = housing_data_target[indices]
    return X_batch, y_batch
```

In [6]:

```
# Execute computational graph using Mini-Batch Gradient Descent.

n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    best_theta = theta.eval()
```

In [7]:

```
# Output theta.

best_theta
```

Out[7]:

```
array([[ 2.0702078 ],
       [ 0.8462986 ],
       [ 0.12034925],
       [-0.27819806],
       [ 0.35567525],
       [ 0.00397728],
       [-0.01187481],
       [-0.86780626],
       [-0.8345916 ]], dtype=float32)
```

Question 5

5(a)

Data sets can have many features, sometimes too many. *Dimensionality reduction* is the process of reducing the dimensions to a manageable size. *Principal Component Analysis*, PCA, is a method to perform *dimensionality reduction*.

PCA uses the idea that the dimensions with the most variance hold the most information.

PCA finds the axis in the data set with the most variance. This is called the first *Principle Component, PC*. PCA then finds subsequent PCs orthogonal to previous ones containing the remaining variance, as many as there are dimensions. This is done using Singular Value Decomposition (SVD) of the co-variance of the data set. To then reduce the data set to n dimensions, the data set is projected on to the hyperplane defined by n PCs.

5(b)

The *Explained Variance Ratio* is the proportion of the variance accounted for in each PC.

5(c)

The *kernel trick* is the observation that a valid (*Mercer Theorem*) *kernel function* $\mathcal{K}(x_i, x_j)$ is capable of finding the dot product of $\phi(x_i)$ and $\phi(x_j)$ without having to evaluate $\phi(x_i)$ or $\phi(x_j)$. This *trick* enables *Support Vector Machines* to efficiently classify non-linear data.

Similarly, *Kernel PCA*, uses the *kernel trick* to perform non-linear projections for *dimensionality reduction*.

5(d)

Locally Linear Embedding, LLE, is a method for *non-linear dimensionality reduction*, NLDR. Rather than use projections, LLE measures the linear relationships between data points, and then seeks a lower dimension space which preserves these relationships the best.

5(e)

Reconstruct x_i as a linear function of it's k nearest neighbours:

$$\mathbf{W} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(x_i - \sum_{j=1}^{m} w_{i,j} x_j \right)^2 \tag{15}$$

where

$$w_{i,j} = 0 \text{ if } x_j \text{ is not one of the } k \text{ nearest neighbours of } x_i \tag{16}$$

and where $\mathbf{W}$ describes the relationships between the *training* data instances.

And normalised such that:

$$\sum_{j=1}^{m} w_{i,j} = 1 \text{ for } i = 1..m \tag{17}$$

NB This first step finds $\mathbf{W}$ for a fixed $\mathbf{x}$.

5(f)

If z_i is the d-space equivalent to x_i, mapping the *training* set into this d-dimensional space, whilst preserving the local relationships as much as possible, requires $\left(z_i - \sum_{j=1}^{m} w_{i,j} z_j \right)^2$ to be minimised:

$$\mathbf{Z} = \underset{\mathbf{z}}{\operatorname{argmin}} \sum_{i=1}^{m} \left(z_i - \sum_{j=1}^{m} w_{i,j} z_j \right)^2 \tag{18}$$

NB This second step finds $\mathbf{z}$ for a fixed $\mathbf{W}$.