COMP7703 – Homework Task 5
Joel Thomas 44793203

1.

Since the student number is 44793203 and the last 4 digits are 3203, the initial 3 has been replaced with next most significant digit, so the four chosen classes as 9, 2, 0 and 3. The equation $z = (W^T(x - \boldsymbol{m})^T)^T = (x - \boldsymbol{m})W$ was used for linear projection with $\dim(W) = 3072 \times 2$ (using the first two principal components only), $\dim(x) = dim(\mathcal{X}) = 4034 \times 3072$ (performing PCA on entire train X) and $\dim(\boldsymbol{m}) = 1 \times 3072$. Note that is slightly different to Alpaydin which assumes $\boldsymbol{x} \in \mathbb{R}^d \rightarrow \dim(\boldsymbol{x}) = d \times 1$ i.e. the linear projection is on a single sample rather than the whole dataset and that each $\boldsymbol{x}$ is stored as a column vector rather than row vector. Hence, we use an additional transpose on $(x - \boldsymbol{m})^T$ to ensure the dimensions match up for matrix multiplication. Finally, take the transpose again (or use the simplified formula) to get $\dim(z) = 4034 \times 2$.

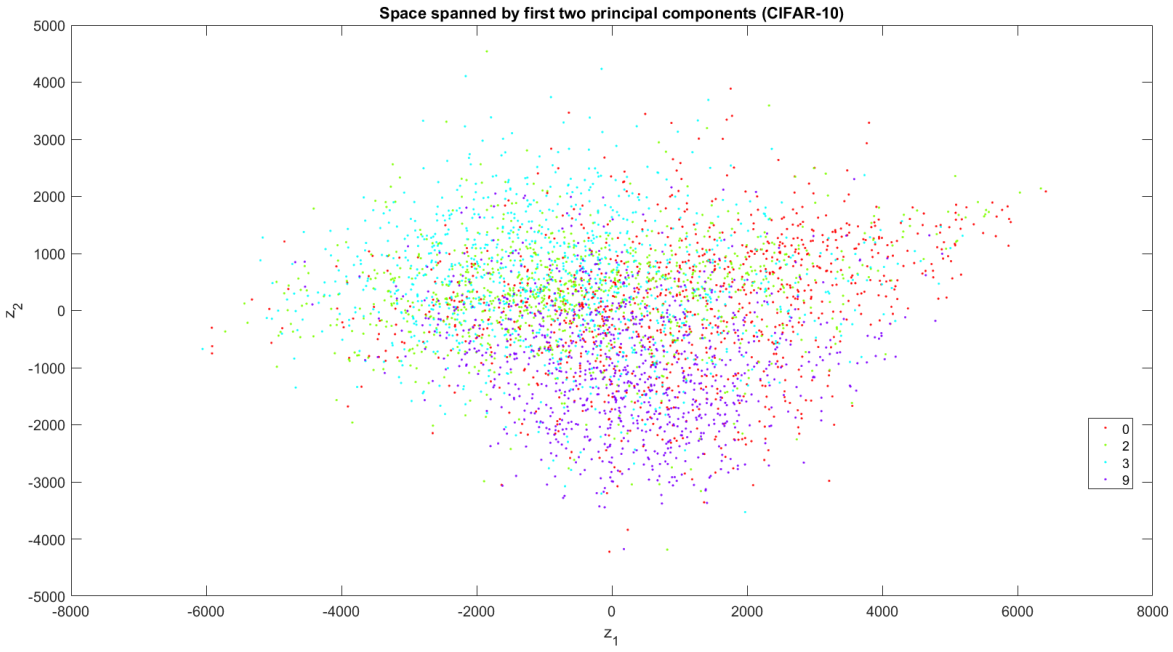

Figure 1: Plot of the data projected onto the first two principal components

A self-implemented function for PCA was used and the code for $pca.m$ is given below:

```
function [z, W, lambdas] = pca(x)
    % Function that implements Principal Component Analysis (PCA)
    % Inputs:
        % x = dataset to perform PCA on
    % Outputs:
        % z = linear projection of x using z = W^T * (x - m)
        % W = principal components (eigenvectors of sample covariance
            % matrix) ordered according to lambdas (see below)
```

```
        % lambdas = eigenvalues of sample covariance matrix in descending
            % order

    % Calculating eigenvectors and eigenvalues
    % dim(W) = d x d, if choosing k columns --> d x k
    [W, lambdas] = eig(cov(x - mean(x)));

    % Sorting eigenvalues from largest to smallest:
    % Decomposing diagonal matrix into array of diagonals
    lambdas = diag(lambdas);
    [lambdas, idx] = sort(lambdas, 'descend');

    % Sorting eigenvectors (columns) according to index sequence from
    % sorting eigenvalues
    W = W(:, idx);

    % z = (W^T * (x - m)^T)^T = (x - m) * W modified for linear projection
    % applied to whole dataset rather than single samples.
    z = (x - mean(x))*W;
end
```

In addition, the code $Q1.m$ was used to perform PCA and create the required plot:

```
clear all;

data = importdata("cifar10_data_batch_1.mat");
train_x = double(data.data);
train_y = double(data.labels);

% Student number = 44793203 --> choosing 9, 2, 0, 3 as the classes to
% perform PCA on. Function ismember finds row indices in train_y matching
% these values, then using that to find corresponding rows of train_x.
train_x = train_x(ismember(train_y, [9, 2, 0, 3]), :);
train_y = train_y(ismember(train_y, [9, 2, 0, 3]));

% Running PCA on the dataset
[z, W, lambdas] = pca(train_x);
```

```
% a) Plot of the data in the space spanned by the first two principal
% components
figure
gscatter(z(:, 1), z(:, 2), train_y)
xlabel("z_1")
ylabel("z_2")
title("Space spanned by first two principal components (CIFAR-10)")
set(gca, 'FontSize', 15)

% b) Percentage of data variance accounted for by the first two principal
% components
pct_var = sum(lambdas(1:2))/sum(lambdas)

% c) Scree graph
figure
subplot(2, 1, 1)
plot(1:length(W), lambdas, "--xk")
xlabel("Number of eigenvectors")
ylabel("Eigenvalues")
title("Scree graph (CIFAR-10)")
set(gca, 'FontSize', 15)

subplot(2, 1, 2)
plot(1:length(W), cumsum(lambdas)./sum(lambdas), "--xk")
xlabel("Number of eigenvectors")
ylabel("Proportion")
title("Proportion of variance explained (CIFAR-10)")
set(gca, 'FontSize', 15)
```

2.

The between-class scatter matrix before projection for $K > 2$ classes is calculated as:

$$S_B = \sum_{i=1}^{K} N_i(\boldsymbol{m}_i - \boldsymbol{m})(\boldsymbol{m}_i - \boldsymbol{m})^T$$

$$N_i = \sum_t r_i^t, \qquad r_i^t = \begin{cases} 1, \text{ if } x^t \in C_i \\ 0, \text{ otherwise} \end{cases}$$

$\boldsymbol{m}_i = \text{mean of the samples from } C_i \text{ before projection}$

$$\boldsymbol{m} = \frac{1}{K}\sum_{i=1}^{K} \boldsymbol{m}_i$$

We know that $N_i = 50 \; \forall \; i = 1,2,3$ because there are exactly 50 samples of each class (Iris-setosa, Iris-versicolor and Iris-virginica) in the provided Iris dataset. The rest of the expressions are relatively easily calculated and we obtain S_B:

$$S_B = \begin{bmatrix} 63.0533 & -19.2316 \\ -19.2316 & 10.6861 \end{bmatrix}$$

The code used $Q2.m$ in finding S_B is given below:

```
clear all;


data = readtable("iris.txt");
train_x = data{:, 1:2};
train_y = data{:, 5};


% Each m_i is d-dim --> d x 1
% Mean of class 1 = Iris-setosa
m1 = mean(train_x(1:50, :))';
% Mean of class 2 = Iris-versicolor
m2 = mean(train_x(50:100, :))';
% Mean of class 3 = Iris-virginica
m3 = mean(train_x(101:end, :))';


% Store in array mi representing m_i from textbook, i = 1:3
mi = [m1, m2, m3];
```

```
% Scatter of the means, taking sum along horizontal instead of vertical
% axis
m = sum(mi, 2)/3;

% N1 = N2 = N3 = N = 50 (recall r^t_i = 1 if x^t is an element of C_i, 0
% otherwise and there are 50 samples of each class in Iris dataset)
N = 50;

SB = zeros(2, 2);
% Between-class scatter matrix BEFORE projection
for i = 1:3
    SB = SB + N * (mi(:, i) - m) * (mi(:, i) - m)';
end

% Display S_B
SB
```

3.

The general type of optimisation algorithm used for training t-SNE is known as Gradient Descent and works as follows. Let θ_j be the j-th parameter of the cost/objective function $J(\boldsymbol{\theta})$ which we are trying to optimise noting that $\boldsymbol{\theta}$ is a vector containing n parameters, $j = 1, \dots, n$. First, we randomly initialise every parameter $\theta_1, \dots, \theta_n$. Next, at each step of the iteration, we attempt to optimise $\theta_j \; \forall \; j = 1, \dots, n$ by decreasing its value if the gradient of the cost function $J(\boldsymbol{\theta})$ with respect to θ_j is positive or increasing its value otherwise. Now we keep repeating this previous step with the hope being to descend into a global optimum but this is not guaranteed and this often minimises $J(\boldsymbol{\theta})$ to a local minimum instead (hence called Gradient **Descent** because we are trying to "descend" the cost function to the most optimal optimum). It is very popular and can be combined with most machine/deep learning models including polynomial regression which already has a closed-form analytical solution. The full algorithm is summarised below:

1) Randomly initialise the parameter vector $\boldsymbol{\theta} = (\theta_1, \dots, \theta_n)$.
2) Evaluate and assign $\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) \; \forall \; j = 1, \dots, n$ noting that $\alpha =$ learning rate (a hyperparameter).
3) Repeat 2) until convergence i.e. new $\theta_j -$ old $\theta_j < \epsilon \; \forall \; j = 1, \dots, n$ where ϵ is a pre-defined convergence threshold.

4.

Wrappers use the machine learning model of interest as a “black box” to score subsets of the features in terms of their predictive power with scoring dictated by performance according to a given metric (e.g. decreasing cost function, increasing validation set accuracy, etc.). Conversely, filters choose subsets of the features as a pre-processing step independently of the chosen target by only relying on the characteristics of these variables (and not a learning model) usually via traditional statistical measures (e.g. correlation, chi-square tests, etc.). Wrappers tend to be greedy (attempting to find the best subset of features given by the best performing model) and are potentially computationally expensive given repeated learning and cross-validation whereas filters are computationally efficient but may not be the best subset for a desired learning model.

5.

The main idea here is regardless of whether two variables are highly but not perfectly correlated (in absolute value), it is important to distinguish between the covariance between these two variables and the covariance given by their class conditional distributions. Although these distributions will generally not be known in advance in the real world, it is important to verify that any two features genuinely do not have a good class separation by manually plotting the features on a 2-D space and observing them (or using an algorithm) even if they are highly positively or negatively correlated. Furthermore, correlation is a measure of **linear** dependence between two features, which may be a poor choice of selecting features if we wish to keep non-linear dependencies (e.g. quadratic, exponential, etc.).