Machine Learning: Assessed Coursework 2

Yadu *Bhageria*

1 Question 1: Image Segmentation and Counting

The image provided is a 640×640 pixel image thus there are $N = 409600$ data points, $\mathbf{P}$. The data for each pixel, p_i, is stored in the Red-Green-Blue (RGB) colour format as $p_i = [R_i, G_i, B_i]$. The image is segmented by clustering pixels into a predefined number of groups in the 3 dimensional RGB space. Two method have been considered: the K-means algorithm and then the more general Gaussian mixture model (GMM).

1.1 K-Means

The basic premise of the K-means algorithm is to partition the data points into a given number of clusters, K, to minimise the within cluster distance from the cluster centre/mean. For cluster k this can be defined as

$$\sum_{p_i \in C_k} ||p_i - \mu_k|| = \sum_{i=1}^{N} z_{ki} ||p_i - \mu_k|| \tag{1}$$

$z_{ki} \in \{0, 1\}$ is a binary indicator that assigns each data point, p_i, to a single cluster, k. μ_k is the cluster mean, $\frac{1}{N_k} \sum_{p_i \in C_k} p_i$ i.e. the average values of the data points in the cluster.

The total measure of the K-means algorithm can then be written as

$$\mathcal{E}_K = \sum_{k=1}^{K} \sum_{i=1}^{N} z_{ki} ||p_i - \mu_k|| \tag{2}$$

which is the overall cluster goodness. The K-means algorithm minimizes this by a 2 step algorithm. Given a segmentation of the data, the cluster centres can be found using the formula above. And given cluster centres, $\{\mu_k\}$, it can be found which data points belong to that cluster, i.e. have the smallest distance function to a cluster. This updates the indicator variables z_{ki}. This process can be repeated until the cluster centres stop moving and the solution converges.

The MATLAB code for 1 iteration is as follows:

```
Mu = Mu_new;              % Set Mu to previously computed Cluster Center

dist = zeros(N,K);        % Compute L2 norm
for k = 1:K
    dist(:,k) = sum((X(:,:) - Mu(:,k)).^2,1);
end
[~,Z] = min(dist,[],2); % Get minimum for each N

for k = 1:K               % Set new cluster centers to mean of their elements.
    if isempty(X(:,Z == k)) == 1 % Incase a cluster has no members move it to a random point
        Mu_new(:,k) = rand(M,1);
    else
        Mu_new(:,k) = mean(X(:,Z == k),2);
    end
end
```

Thus for the pixels in the RGB space, each pixel intensity vector can be replaced by cluster mean value corresponding to the cluster of that pixel. Furthermore for storage purposes it means that each picture can be stored using only K cluster mean vectors and a single integer value assigning the data points to one of the cluster means.

1.2 Gaussian Mixture Models

A Guassian mixture model differs from the K-means algorithm in that it assigns a distribution over the indicator variables rather than only assigning binary indicator variables for each cluster-data point pair. This means the model is more complex but allows better segmentation of the image. This is because the K-means algorithm uses the L2 norm and thus is biased towards spherical distributions whereas GMM can have gaussians with covariance between dimensions.

A GMM trys to fit K gaussians to the data. That means the parameters $\boldsymbol{\mu}_k \in \mathbb{R}^3$ and $\boldsymbol{\Sigma}_k \in \mathbb{R}^{3\times 3}$ for each cluster must be found. Let $\boldsymbol{\theta}$ be the parameters of the entire model. Maximising the the log likelihood of the data given the parameters is a method to do so. In lectures the lower bound on the log likelihood was derived as

$$\mathcal{L}_B = logp(\mathbf{P}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{P}) logp(\mathbf{P},\mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{P}) logp(\mathbf{Z}|\mathbf{P}) \tag{3}$$

where $\mathbf{Z}$ is the matrix of indicator variables. This can be maximized using the Expectation-Minimization algorithm. Given estimates for the model parameters, the probability density of the latent variables, $p(\mathbf{Z}|\mathbf{P})$, can be calculated. And given the probabilities of the latent variables, the parameters values, $\boldsymbol{\theta}$, can be estimated by maximizing the log likelihood. These two steps are repeated until convergence.

So the algorithm is set up to start at random initial conditions and then first calculate the posterior distribution over the mixture components

$$p(k|p_i) = \frac{p(p_i|\theta_k)p(k)}{\sum_{k'=1}^{K} p(p_i|\theta_{k'})p(k)} \tag{4}$$

where k is a cluster. Then estimate the parameters, $\boldsymbol{\theta}$,

$$\hat{\boldsymbol{\mu}}_k = \frac{\sum_{i=1}^{N} p(p_i|\theta_k)p_i}{\sum_{j=1}^{N} p(p_j|\theta_k)} \quad \hat{\boldsymbol{\Sigma}}_k = \frac{\sum_{i=1}^{N} p(p_i|\theta_k)(p_i-\hat{\boldsymbol{\mu}}_k)(p_i-\hat{\boldsymbol{\mu}}_k)^T}{\sum_{j=1}^{N} p(p_j|\theta_k)} \tag{5}$$

and finally repeat these two steps until convergence.

1 iteration of this algorithm in MATLAB is written as:

```
% E-Step: Compute expectations
for k = 1:K
    sqrt_Sigma = 1/ sqrt( det( Sigma(:,:,k) ) );
    Xminus =  X - Mu(:,k);
    invSX = -0.5 * (Sigma(:,:,k) \ Xminus);
    PKX(:,k) = Pk(k) * sqrt_Sigma * exp( dot(Xminus,invSX,1));
end
PKX = PKX ./ sum(PKX,2);  % Normalize posterior

for k = 1:K
    Pk(k) = sum(PKX(:,k)) / N;
end

% M-Step: Maximise parameters values based on the likelihood function
```

```
for k = 1:K
    Mu_new(:,k) = 0; % Compute new values for Cluster centers
    PKXX = X;
    for m = 1:M
        PKXX(m,:) = PKX(:,k) .* X(m,:)';
    end
    Mu_new(:,k) = Mu_new(:,k) + sum( PKXX, 2) / sum( PKX(:,k) );

    Sigma_new(:,:,k) = 0; % Compute new values for Cluster variances
    Xminus = X - Mu_new(:,k);
    PKXminus = Xminus;
    for m = 1:M
        PKXminus(m,:) = PKX(:,k) .* Xminus(m,:)';
    end
    Sigma_new(:,:,k) = Sigma_new(:,:,k) + PKXminus * Xminus' / sum( PKX(:,k) );
    % Incase sigma is a non invertible matrix
    if rcond(Sigma_new(:,:,k)) < 10^(-5*M)
        for m = 1:M
            Sigma_new(m,m,k) = 1;
        end
    end
```

Thus for the pixels in the RGB space, each pixel intensity vector can be replaced by a centroid vector. A centroid vector is formed using the probabilities of the indicator variables for each data point along with the K cluster means using the formula $\sum_k P(k|x_i)\mu_k$.

1.3 Results

Image segmentation results using the K-means algorithm for $K = 2, 3, 5, 10$ are shown in the figures. The normalised within cluster sum of squares for $K = 1, \ldots, 10$ clearly has a sharp decrease until $K = 3$ and then plateaus/flat-line a little[1]. Given that we would prefer a simpler model (Occam's Razor) and thus a smaller value of K, this sharp turn indicates that a value of $K = 3$ is a good choice to segment the colour pixels of this image. This result is also reenforced by our intuition as looking at the image, there are clearly 3 dominant colours present: black, green, and blue.

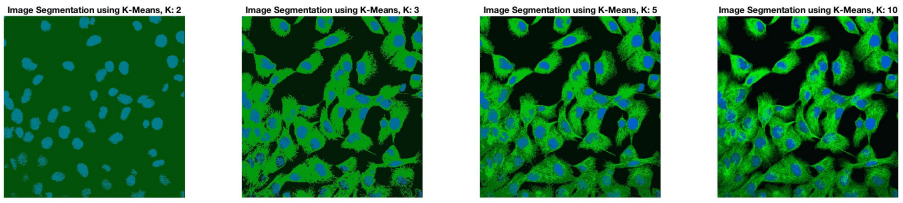

Figure 1: Image segmentation using the K-means algorithm for different values of K

Image segmentation results using a Gaussian Mixture Model algorithm for $K = 2, 3, 5, 10$ are as shown. There is large variation in the likelihood of the model fitted to the data in GMMs due to the random starting positions of the initial cluster centres. This can be seen in the likelihood figures below. There is flatlining of the negative LogLikelihood after a large drop in one of the cases at $K = 4$. Also later using cross validation it can also be seen that the negative LogLikelihood plateaus at $K = 4$ after a large drop. Thus using the idea of the "Elbow method" and Occam's Razor it seems that $K = 4$ is a reasonable choice for a Gaussian Mixture Model.

It should be noted that the inferences in both the cases are subjective.

[1] This method is often called the "Elbow Method"

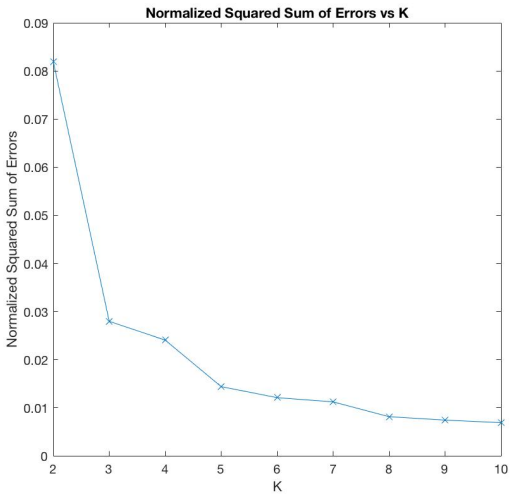

Figure 2: The normalised within cluster sum of squares vs K values using the K-means algorithm

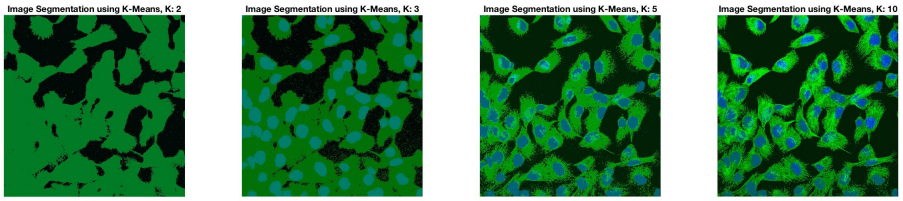

Figure 3: Image segmentation using a Gaussian Mixture Model for different values of K

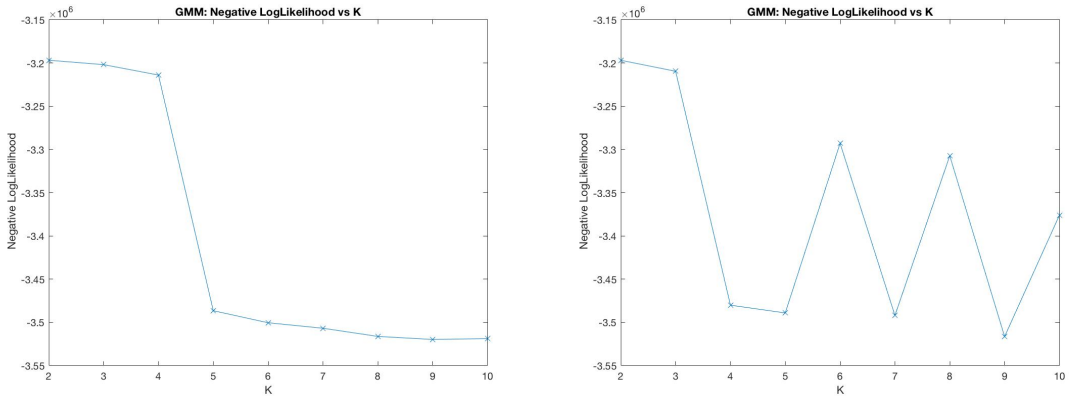

Figure 4: The normalised within cluster sum of squares vs K values using a GMM with K clusters

Another technique that was used to try and see if the a better choice of K could be made was 2 fold cross validation. I took random permutation of the data and split it into test and training sets.

```
Xperm = X(:,randperm(length(X),length(X))); % X is the entire data here of size (MXN)
Xtest = Xperm(:,1:N/2);
Xtrain = Xperm(:, N/2+1:N);
```

Then fit the parameters of the models using the training data and compared the fit to the test data. But the results were mostly inconclusive except for reenforcing the idea that $K = 4$ is a reasonable cluster number choice for a GMM. I believe this because many of pixels lie very close together in the 3D RGB space which can be seen on the scatter figure produced.

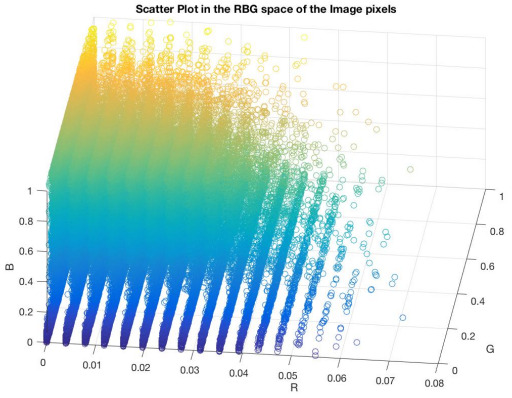

Figure 5: Scatter graph of the pixels in the normalized RGB 3D space. Shows how most of the pixels are concentrated in 1 corner

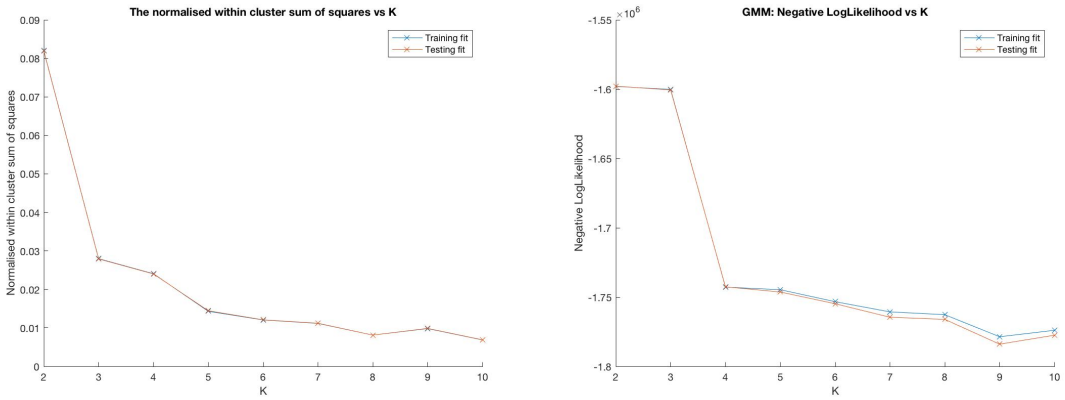

Figure 6: The results for cross validation using K means (left) and GMM (right) fitting. It can clearly be seen that the squared sum or errors and the likelihood for both the test and training data are very closely matched

1.4 Counting the Number of Cells

After an image has been segmented, there is still the issue of counting the number of cells present in the image. In order to do so, I extract the cluster of the images with its centre located closest to the blue colour vector.

This task has been attempted to be automated by using a Gaussian Mixture Model or K-Means algorithm to try and fit it with K clusters. Then model comparison can be utilised to find the value of K. First lets look at the negative LogLikelihood for various values of K fit onto the cell data.

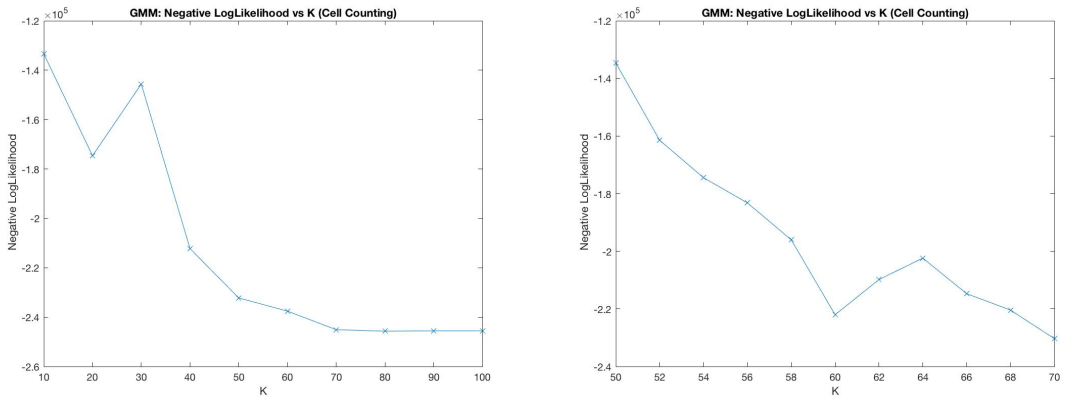

Figure 7: Negative LogLikelihood vs K for fitting a GMM to the cell data. The figure on the right is a higher resolution version of the one on the left

The negative LogLikelihood for the GMM fits for various values of K do not have as sharp as a drop to plateau change. Thus it is hard to find a good cut-off point by inspecting the graph. But the value flatlines after $K = 80$. Running the same test for values between $K = 50$ and $K = 70$ is useful and done on the image on the right. It indicates that $K = 62$ might be a reasonable choice.

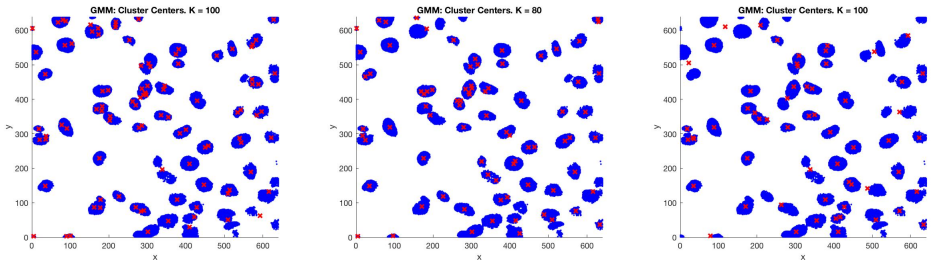

Figure 8: The cluster centres for fitting a GMM to the cell data for various values of K. The effect of some cluster centres getting 'stuck' between other clusters can be observed here.

Due to the random starting points of the GMM algorithm, it does not always produce optimal segmentation. i.e. the result converges but not necessarily to the best solution. Thus looking at the plots of the cells with clusters centres marked it can be seen that some clusters are, in a sense, stuck between two other clusters and thus cannot move towards a visually unoccupied cluster of points

that do not have a cluster near them. Hence simply looking at the likelihood is not a good idea. Although the above plots give an indication that the answer might be $K = 62$.

2 Question 2: Bayesian Linear Regression and Gaussian Proccesses

1. Given data [2] , $\mathcal{D} = \{x_t, y_t\}_{t=1}^N$, linear regression models the data as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} = f(\mathbf{X}) + \boldsymbol{\epsilon} \tag{6}$$

where $\mathbf{y}$ is the vector of regressand values, $\mathbf{X}$ is a matrix whose rows contain the covariates for each data point, and $\boldsymbol{\epsilon}$ is a vector of the noise. $f(\mathbf{X})$ is then the hidden function of time.

This model describes a family of functions and a single realisation is generated by choosing values for the parameter vector $\boldsymbol{\beta}$ along with appropriate parameters for the noise vector. In this question it is assumed that $\epsilon_t \sim N(0, \sigma_\epsilon^2)$.

The likelihood function of the data given the parameter values is

$$P(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}, \sigma_\epsilon) = \prod_{t=1}^{N} p(y_t|x_t, \boldsymbol{\beta}, \sigma_\epsilon) = \prod_{t=1}^{N} \mathcal{N}_{y_t}(\mathbf{x}_t\boldsymbol{\beta}, \sigma_\epsilon) \tag{7}$$

Maximizing this likelihood function gives the most likely set of parameters. But in a Bayesian framework, a probability distribution over the parameters is of interest, $p(\boldsymbol{\beta}, \sigma_\epsilon|\mathbf{y}, \mathbf{X})$ i.e. posterior distribution for the parameters given the data. This posterior can be found using Bayes Rule.

Bayes Rule for a model with parameters Θ and given data $\mathcal{D}$ is

$$P(\Theta|\mathcal{D}) = \frac{P(\mathcal{D}|\Theta)P(\Theta)}{P(\mathcal{D})} \tag{8}$$

where $P(\Theta|\mathcal{D})$ is the posterior probability (the probability of a certain model parameters given the data), $P(\mathcal{D}|\Theta)$ is the probability of observing the data given a model with parameters Θ, $P(\Theta)$ is the probability of certain model parameters, i.e. our prior beliefs about the model, and $P(\mathcal{D})$ is the probability of obtaining the given data over all values of Θ.

Going back to our case of Bayesian Linear regression and using the simplification that σ_ϵ is known this gives that the posterior distribution can be written as

$$\begin{aligned} p(\boldsymbol{\beta}|\sigma_\epsilon, \mathbf{y}, \mathbf{X}) &= \frac{p(\sigma_\epsilon, \mathbf{y}, \mathbf{X}|\boldsymbol{\beta})p(\boldsymbol{\beta})}{p(\sigma_\epsilon, \mathbf{y}, \mathbf{X})} \\ &\propto p(\sigma_\epsilon, \mathbf{y}, \mathbf{X}|\boldsymbol{\beta})p(\boldsymbol{\beta}) \end{aligned} \tag{9}$$

Now assuming an independent Gaussian prior over each of the parameters in the weight vector, β_i, gives

[2] In this question, I constrain my independent variable to be 1 dimensional given the CO2 dataset being analyzed

$$p(\boldsymbol{\beta}) = \mathcal{N}(0, \Sigma_\beta) = \frac{1}{\sqrt{2\pi}|\Sigma_\beta|^{1/2}} \exp -\frac{1}{2}\boldsymbol{\beta}^T \Sigma_\beta^{-1} \boldsymbol{\beta} \tag{10}$$

$$p(\sigma_\epsilon, \mathbf{y}, \mathbf{X}|\boldsymbol{\beta}) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_\epsilon}} \exp -\frac{1}{2\sigma_\epsilon^2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{11}$$

Using the fact that the product of two gaussians is also gaussian this gives another gaussian distribution for the posterior probability. This can be written as

$$p(\boldsymbol{\beta}|\sigma_\epsilon, \mathbf{y}, \mathbf{X}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{12}$$

where $\boldsymbol{\mu} = \frac{1}{\sigma_\epsilon^2}(\boldsymbol{\Sigma}_\beta^{-1} + \frac{1}{\sigma_\epsilon^2}\mathbf{X}^T\mathbf{X})^{-1}$ and $\boldsymbol{\Sigma} = \frac{1}{\sigma_\epsilon^2}(\boldsymbol{\Sigma}_\beta^{-1} + \frac{1}{\sigma_\epsilon^2}\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

So finally, the underlying hidden function of time $f(\mathbf{X})$ in the form $\mathbf{X}\boldsymbol{\beta}$ can be considered as a linear combination between the covariate matrix $\mathbf{X}$ defined by the independent variables and the parameter vector, $\boldsymbol{\beta}$, for which a posterior distribution can be found using the given data.

2. Consider two points x_i and x_j. For linear regression each point can be represented as a linear combination of its basis function (the basis functions form the covariates from the independent variables) as $f(\mathbf{X}_i) = \mathbf{X}_i\boldsymbol{\beta} = \sum_k \beta_k\phi_k(\mathbf{x}_i) = \phi(\mathbf{x}_i)\boldsymbol{\beta}$.

   So the covariance of the function f for the two time points will be

$$\mathrm{Cov}(f(\mathbf{X}_i), f(\mathbf{X}_j)) = \mathbb{E}[f(\mathbf{X}_i)f(\mathbf{X}_j)^T] - \mathbb{E}[f(\mathbf{X}_i)]\mathbb{E}[f(\mathbf{X}_j)] \tag{13}$$

$$\mathbb{E}[f(\mathbf{X}_i)] = \mathbb{E}[\phi(\mathbf{x}_i)\boldsymbol{\beta}] = \phi(\mathbf{x}_i)\mathbb{E}[\boldsymbol{\beta}] = 0 \tag{14}$$

   as $\beta \sim \mathcal{N}(0, \Sigma_\beta)$. Therefore

$$\begin{aligned}\mathrm{Cov}(f(\mathbf{X}_i), f(\mathbf{X}_j)) &= \mathbb{E}[f(\mathbf{X}_i)f(\mathbf{X}_j)^T] = \mathbb{E}[\phi(\mathbf{x}_i)\boldsymbol{\beta}\boldsymbol{\beta}^T\phi(\mathbf{x}_j)^T] = \phi(\mathbf{x}_i)\mathbb{E}[\boldsymbol{\beta}\boldsymbol{\beta}^T]\phi(\mathbf{x}_j)^T \\ &= \phi(\mathbf{x}_i)\Sigma_\beta\phi(\mathbf{x}_i)^T\end{aligned}$$

   Note $\mathbf{x}_i$ is a row of values and thus $\mathbf{x}_i\mathbf{x}_j^T$ defines an inner-product.

   Thus the covariance is a linear combination of Gaussian variables. A Gaussian process is defined by its mean function $\mu(\mathbf{x})$ and Covariance matrix, $K(\mathbf{x}, \mathbf{x}')$. In this case defining $K(\mathbf{x}, \mathbf{x}') = \mathrm{Cov}(\mathbf{x}, \mathbf{x}')$ gives that f is then a gaussian process.

3. In the following section are results produced by fitting a Gaussian Process to the CO2 data recorded in Hawaii. This is done by first finding the hyperparameters that maximise the log likelihood of the data. This is done in `GP.m`. The basic process is as follows [1].

   Firstly assumptions must be made by looking at the data set on what covariances function to utilize. In the figures below I have used 3 different types of Kernels.

   The first is a linear kernel function with added white noise. This has the form.

$$K(x_i, x_j) = \sigma_f(x_i - c)(x_j - c) + \sigma_n\delta_{x_i,x_j} \tag{15}$$

The second is a squared exponential function with added white noise

$$K(x_i, x_j) = \sigma_f e^{-\frac{1}{2l_1^2}(x_i - x_j)^2} + \sigma_n \delta_{x_i, x_j} \tag{16}$$

And finally the third and the one with most hyper parameters was a squared exponential plus squared sin expotential

$$K(x_i, x_j) = \sigma_f e^{-\frac{1}{2l_1^2}(x_i - x_j)^2} + e^{-\frac{1}{2l_2^2} sin(f*(x_i - x_j))^2} + \sigma_n \delta_{x_i, x_j} \tag{17}$$

Each of these Kernels has a different hyper-parameters and for each case these must be optimized. This can be done by using the `fminunc()` function in MATLAB where an input of the Kernel generating function that itself takes the hyper parameters and training data as input and gives the negative LogLikelihood of the Kernel and its derivatives with respect to the various hyper parameters. This is done in `OptimHP.m`.

```
[theta, fval] = fminunc(@(hyperparameters) optimHP(hyperparameters,constants), startVals(s,:), options);
```

An example of a gradient of with respect to the hyper-parameters can be seen here for the squared expotential [2]:

```
% Covariance
K = sigma_f^2 * exp(-(x1-x2)^2/(2*l^2));
% Derivatives
d_l = K * (l^-3) * (x1-x2)^2;
dsigma_f = 2*sigma_f * exp(-(x1-x2)^2/(2*l^2));

```

Once the hyper-parameters have been optimized, then the realized Kernel matrix is computed from the training data. Using this the mean for the test data along with 95% confidence intervals bounds can be found.

```
% Fit test data to the GP with the found hyperparameters
[ytest, bounds, K, kstar] = computeGP (@K_SE, X, y, theta, xtest);
```

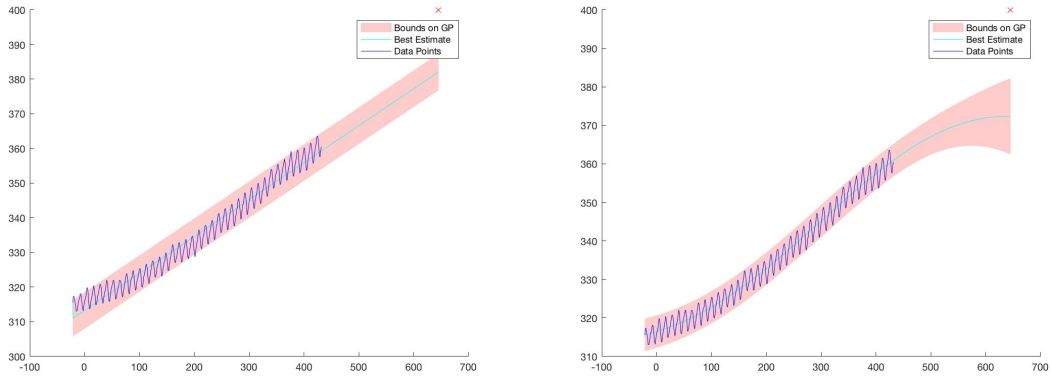

Figure 9: Predictions made by my Gaussian Process models in the 95% confidence interval. The observation of 400pm in 2013 is marked with a red cross. The x-axis here is Months after 1960. The y-axis is the CO2 levels.

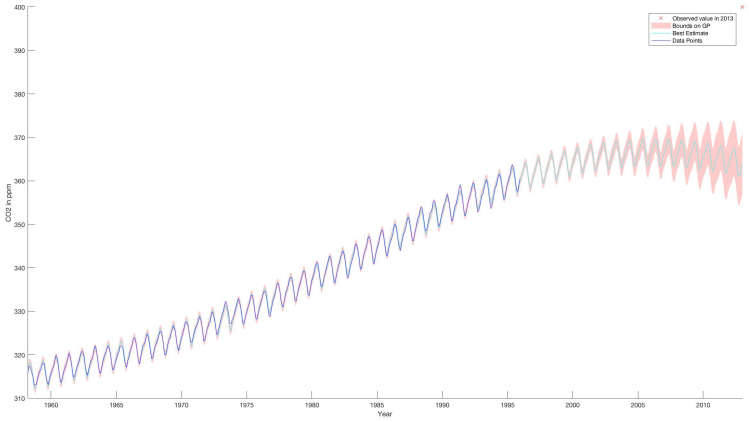

Figure 10: Predictions made by my Gaussian Process models in the 95% confidence interval. The observation of 400pm in 2013 is marked with a red cross

The graphs above have 95% confidence intervals plotted for the data points until 2013. Looking at the various fits of the Gaussian Process for the different Kernel functions, it can be seen none of them predict the dramatic increase in CO2 level ppm recorded in 2013.

The linear Kernel is unable to capture the nature of the exponential rise in the data.

And the squared exponential and periodic kernels are limited due to the training data stopping about 20 years before the prediction point. This point at 2013 is far out from the actual training data and thus the non-noise components start to diminish and the kernel becomes dominated by the noise element leading to the flattening of the curve.

An exponential increase occurred in the CO2 emission level between 1993 and 2013. This is hard to model using a Gaussian Process as seen from the previous part that the underlying generating function is a linear combination of basis functions. And in the case of a Guassian Process, having an exponentially increasing on decreasing basis function is hard to model.

3 Question 3: Social Consequences of AI and Machine Learning

Machine learning, and artificial intelligence (AI[3]), have gained popularity in recent years. The heart of these techniques lies in trying to find optimisations such that a computer can 'learn' how to handle a specific set of problems given enough training data rather than finding explicit solutions. The wide scope of these techniques means that no single researcher can hope to assess all of their possible impacts. In light of this, I shall look at an example that is near the horizon and will quite possibly have a large impact.

My chosen example is that of autonomous cars. Many leading automobile companies, such as Tesla [3], BMW, and more have invested significantly in this field along with technology companies such as Google and Intel as well [4]. Driverless cars use GPS systems and map databases to locate themselves in their environments. Moreover, they utilise machine vision systems, often using lasers, to map out their immediate surroundings. The systems then need to classify various elements on the road such as cars, busses, cyclists and pedestrians. This is done through combinations of PCA basis analysis and classification algorithms. Furthermore, they also need to predict where each element on the road will be moving in the future and utilize algorithms such as Hidden Markov Monte Carlo techniques, as well as classical IF-THEN rules [5] [6].

It is estimated that over 1 hour per day, per person, is spent driving in many European countries [7]. Consider a reality where people no longer need to drive cars at all. This time could instead be utilized for other activities. People could sleep en-route and thus live further away from work leading to less densely populated urban areas. Rented cars might immerse us in advertisements that are personalized using machine learning algorithms, just like for webpage adverts nowadays [8]. Furthermore, there will be a revolution in delivery services that could lead to a large number of jobs becoming automated. A large change in one industry will undoubtedly impact entire economies. Only time will tell whether these impacts will be, on the whole, positive or negative.

A key consequence of autonomous cars will be our requirement of having to deal with new ethical grey-areas. Suppose a car with a passenger runs over a pedestrian, killing them instantly. If the situation is deemed avoidable, is the car company at fault or is it the passenger? Does it matter if the passenger owned the car or rented it? New laws will have to be written and entire systems, such as driving licenses, redone. Already many automobile owners in China, Germany, Korea, and USA have started to use driver-assistance systems [9]. I do not believe the transition to autonomous cars will be sudden, but as their prevalence rises I am certain they will impact us all regardless of whether we choose to adopt them. I only hope that their impact is not physical.

[3]For the sake of clarity I'd like to distinguish between AI, artificial intelligence, and AGI, artificial general intelligence. The former is able to tackle a set of given tasks, e.g. categorising images based on the objects present in them. The latter is able to tackle any sort of problems/tasks and is comparable to creating intelligence that is, in a sense, similar to our own intelligence.

4 Question 4: Bitcoin Transactions

Before choosing a model, I believe it is useful to understand and visualise the data. A key issue with the data set provided is that each transaction does not have a time stamp but only a date stamp along with a transaction ID (assumed to be chronological in time [4]. Hence the data is not linearly spaced time. For example, it is not possible to know whether 1 second passed between 2 transactions or 1 hour. This notion is further reenforced by noting the variation in the number of transactions per day.

```
NumTransactionsPerDay
= 1261  3261 3754 1485 1447 1993 2928 1231 2473 115]
```

The data can then be visualized by looking at the spread of transactions per day. This gives

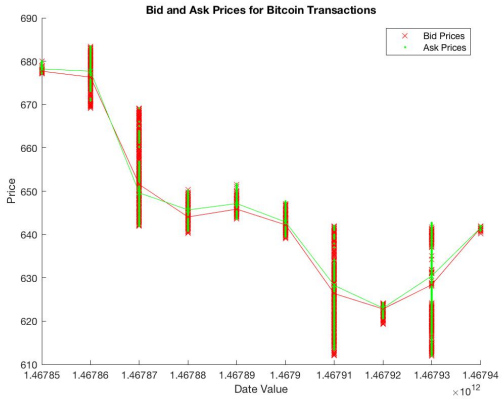

Figure 11: Bid and Ask prices for the data set. Also the lines represent the average bid and ask price traded each day.

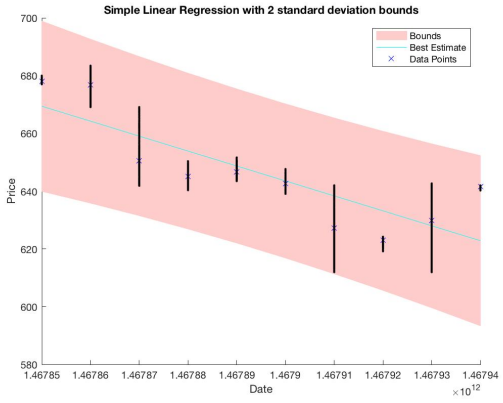

Figure 12: Simple linear regression on the data points that represent the average price of the day

[4]The provided data does have the transaction IDs in order either and thus the data has been sorted to fix this. See Figure 14 in the Appendix for more details

Simple linear regression gives extremely large prediction bounds (Figure 12) that would not be very useful in order to predict the price and make transactions. An example of this can be seen in the figure produced.

Even though the data is not linearly spaced in time. It is helpful to see the impact the bid and ask price have on the trend of the data. It can clearly be seen that often when transactions are repeatedly occurring at bid price, the price is falling and often when the transactions are repeatedly occurring at ask price, the price is increasing. This agrees with our basic market knowledge that if people are willing to pay some incremental amount more than the bid to match the ask price then there is larger demand than supply and thus the price moves up (and vice versa).

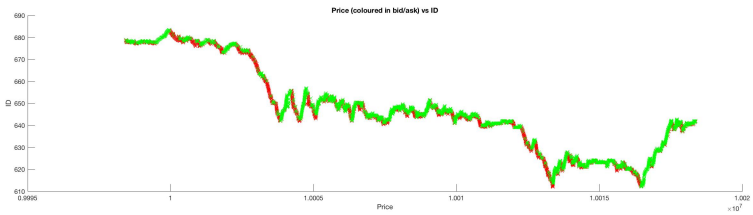

Figure 13: A plot of the prices of Bitcoin in Price vs ID. Coloured red for bid price and green for ask price.

Now as the assumption is made that the transaction IDs are chronological in time, this Bitcoin data must then be treated as a times-series. The price at transaction t appears to be dependent on the prices of the transactions before along with the bid-ask binary information for each transcation. This is an ideal problem for a neural network to solve!

The Neural Network Architecture that I have considered has 3 layers with a sigmoid activation function. The input layer is specified by the size of number of prices plus bid/ask binaries inputted to the system. The hidden layer of a chosen size (mostly through trial and error - more on this later). And finally the output layer, in my chosen model, has 2 nodes representing a binary probability of the price going up or down in the future.

A key element for this model is determining what it means for the price to go up or down in the future. Simple taking the difference between the price between two immediate transactions is not reasonable. Visualising the bit-coin data for small intervals shows that it behaves very much like any other financial instrument in that there is huge amount of fluctuations in the price at small time intervals. But nonetheless there are clear trends in the direction the average price is heading. So to test the data a moving average is considered. It is set up in a simple way

$$\text{movingAvg}_t = \frac{1}{\tau}\sum_{i=1}^{\tau} \text{movingAvg}_{t-i}$$

where τ is the size of the moving average window. So now in our model we wish to input some known values up to the point t and get an indication of whether the average price in the next τ points will be higher or lower than the current average price (also over the same τ time period.

N_I is the number of input layers. N_H is the number of hidden layers. And N_O is the number of output layers.

Then for a 3 layer neural network with input vector $X \in \mathbb{R}^{N_I}$,

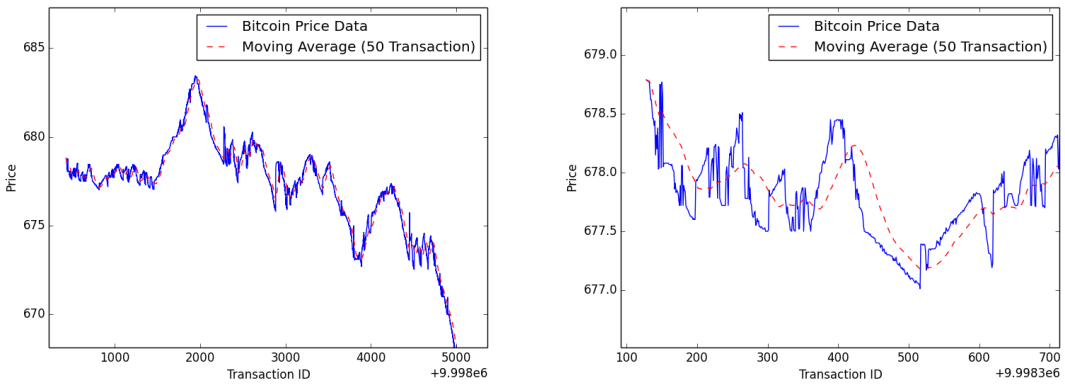

Figure 14: Moving Average over 50 transactions for the Bitcoin Data. The right figure is a zoomed in version of the left. The moving in red is for the previous 50 data points and comparing the moving average between 2 points is more indicative than simply comparing the prices between them.

$$
\begin{aligned}
u_1 &= W_1 X + c_1 \\
v_1 &= \tanh(u_1) \\
u_2 &= W_2 v_2 + c_2 \\
v_2 &= \frac{\exp(u_2)}{\sum_i \exp(u_{2i})}
\end{aligned}
\tag{18}
$$

where $W_1 \in \mathbb{R}^{N_H \times N_I}$, $c_1 \in \mathbb{R}^{N_H}$, $W_2 \in \mathbb{R}^{N_O \times N_H}$, and $c_2 \in \mathbb{R}^{N_O}$. Also $v_1 \in \mathbb{R}^{N_H}$ and $v_2 \in \mathbb{R}^{N_O}$. The model learns but propagating the error backwards by using a test solution vector $y \in \mathbb{R}^{N_O}$ such that

$$
\begin{aligned}
\Delta_1 &= v_2 - y \\
\delta W_2 &= \Delta v_1^T \\
\delta c_2 &= \sum_i \Delta_{1i} \\
\Delta_2 &= W_2^T \Delta_1 (1 - v_1^2) \\
\delta W_1 &= \Delta_2 X^T \\
\delta c_1 &= \sum_i \Delta_{2i} \\
\delta W_2 &= \delta W_2 + \epsilon_2 W_2 \\
\delta W_1 &= \delta W_1 \epsilon_2 W_1 \\
W_1 &= W_1 - \epsilon_1 \delta W_1 \\
c_1 &= c_1 - \epsilon_1 \delta c_1 \\
W_2 &= W_2 - \epsilon_1 \delta W_2 \\
c_2 &= c_2 - \epsilon_1 \delta c_2
\end{aligned}
\tag{19}
$$

4.1 Results

Given that the data is sequential, I believe it is not possible to shuffle the data around, else the intrinsic behaviour of the impact between bid/ask price and the movement of price is lost. Furthermore as the neural network components are initialized to random values it is important to model test multiple times to ensure the results are not generated simply due to chance and rather due to the network learning from the training set.

One method is to use 10-fold cross validation where a new random model is initalized for each set of data to be tested. Using this I see get an averaged prediction performance over all testing sets of 0.547%. The values of individual sections can be seen in Figure 15. Some aspects of interest are that the sections with relatively flat prices and/or that have less autocorrelated data (more up/down spikes) are closer to 50% accuracy as we would reasonably suspect. Sections with strong trends in either direction and/or large autocorrelations perform significantly better.

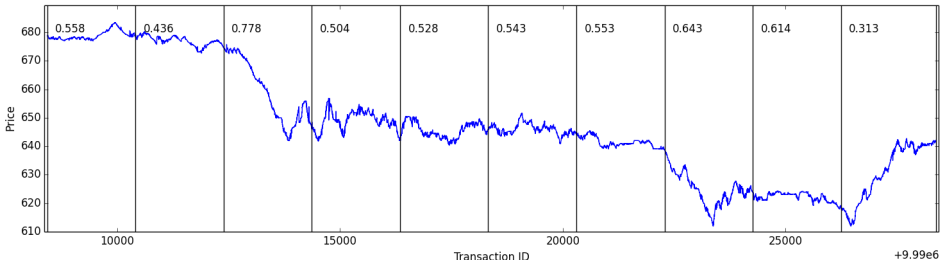

Figure 15: 10-fold Cross Validation. Lines display where the sets are split and the percentage of values correct that specific set was tested using all the other sets for training.

Another method of testing used given the sequential nature of the data is iteratively testing the data using the first X points as training data and then testing on the next X followed by training on the currently tested points and repeating this process multiple times for different random starting conditions of the neural network. It can be seen that this method perhaps with an average performance of 52.9% for batches of size 2000 and of 53.4% for batches of size 4000. The batch size is the value of X defined before. Once again the model predicts better when there are strong underlying trends and worse when the data is mostly flat.

Overall it seems that the neural network is able to get a 52-55% rate of predicting the price correctly over the entire data set. In terms of trading an asset this is quite significant as over time a lot of money could be made with such an advantage. But there are also several assumptions made in the implementation that compromise this result.

1. Firstly, this model does not have an accurate time-stamp and thus it is not possible to know whether all these transactions occurred in a small time interval. Perhaps one even too small for a neural network to train and predict based on past values.

2. Secondly, it is assumed that transactions IDs are chronological in time but this must be verified.

3. Most importantly, our training set is quite small - only 10 days. This means that even though the results have been cross validated, it is hard to assess whether the entire data set is rep-

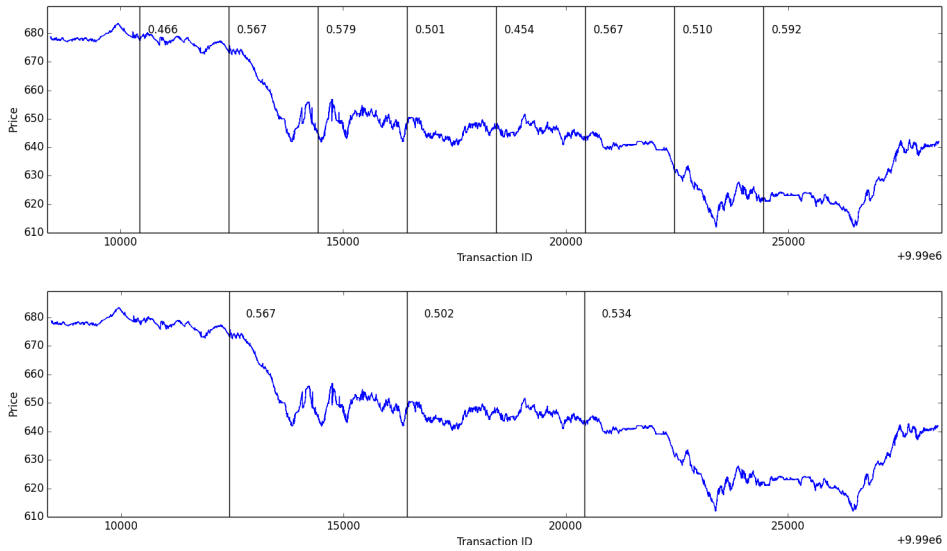

Figure 16: Iterative Batch Cross Validation with 10 Repeats. In the upper figure, the first 2000 values are used to training and then the next 2000 are tested after which they are used for training and this process is iteratively repeated. In the lower figure the same has been done but for a batch size of 4000. The values indicate the performance (percentage of times the model correctly predicted the price trend) in each section

resentative of the general market of Bitcoin transactions. Furthermore, this ties into the idea that neural networks require tremendous amounts of data to train properly.

4. Lastly, the analysis of neural networks is still in its infancy. There are no exact methods for determining what sort of architecture will lead to the best result. This also makes it extremely hard to choose and then justify parameter choices, such as the learning rate.

A similar implementation could be attempted on properly formatted data. Properly formatted data is received regularly, i.e. in set time steps, and sequentially in time. Without regularity, many forms of inference do not make sense such as moving averages and volatility which would be extremely beneficial here. Also, then other simpler and better understood models, for example AR regression models, could be implemented for comparison given the lack of rigour involved in neural networks.

Ultimately, I would not trade based on this model but I think it is a good proof of concept.

Appendix

4.2 Q4: Figure

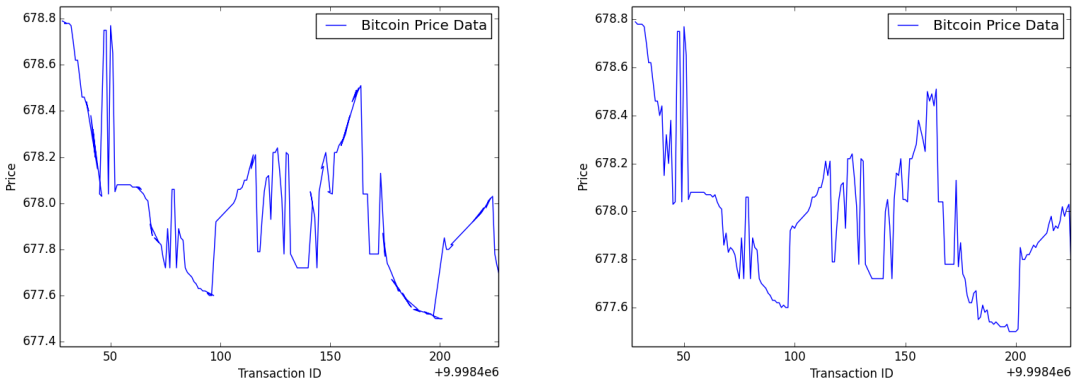

Figure 17: A zoomed in view of small intervals in Transaction IDs with the figure on the left plotted without transaction IDs sorted and the one of the right plotted with the transaction IDs sorted. The locally random ordering of transaction ID data is not visible on a global scale but clearly impacts any implementation that is dependent on assuming the Bitcoin data is time-series data and the underlying function for the price depends on the previous prices

4.3 Code

Note that the bound plotting function that I have used is heavily influenced from [1]. Also the Gaussian Process code is implemented in line with the same tutorial. Furthermore I believe it is worth mentioning that I have used a similar code for investigations in my on-going M4R course as well.

4.3.1 Q1

`Kmeans.m`

```
function [ Z, Mu , EK] = Kmeans( X, K, tol, display )
% Uses the K Means algorithm to segment the inputted data into K clusters
% INPUTS:       X: M x N points of data to be segmented using a gaussians
%               K: Number of clusts. Default set to 3.
% OUTPUTS:      Z: Contains the centroid vector for each data point
%               C: Defines the cluster means for each centroid vector
%              Ek: Normalized Within Cluster Sum Squared of Errors

if (nargin < 2) K = 3; end      % Set default cluster value to 3 if not given
if (nargin < 3) tol = 1e-10; end % Set default value for the break tolerance
if (nargin < 4) display = false; end % Display iteration count/means

[M,N] = size(X);    % Extract the size of K

Z = zeros(N,1);     % Array to store the indicator variables
dist = zeros(K,1);  % Array to store the norm between a point and the cluster means
size_k = zeros(K,1);% Array to store the size of each cluster in each iteration
Mu = zeros(M,K);    % Array to store the cluster means
Mu_new = rand(M,K); % Array to store the new cluster means
iters = 0;          % iteration count

```

```
while norm(Mu - Mu_new) > tol
    if (display) tic; end

    Mu = Mu_new;              % Set Mu to previously computed Cluster Center

    dist = zeros(N,K);        % Compute L2 norm
    for k = 1:K
        dist(:,k) = sum((X(:,:) - Mu(:,k)).^2,1);
    end
    [~,Z] = min(dist,[],2); % Get minimum for each N

    for k = 1:K               % Set new cluster centers to mean of their elements.
        if isempty(X(:,Z == k)) == 1 % Incase a cluster has no members move it to a random point
            Mu_new(:,k) = rand(M,1);
        else
            Mu_new(:,k) = mean(X(:,Z == k),2);
        end
    end
    if display
        iters = iters + 1
        Mu_new = Mu_new
        diff = norm( Mu - Mu_new)
        toc
    end
end

EK = 0; % Compute Normalized Within Cluster Sum of Squares
for k = 1:K
    EK = EK + sum( sum( (X(:,Z == k) - Mu(:,k)).^2));
end
EK = EK / N;

end
```

GaussianMixtureModel.m

```
function [Z, Mu, LK,PKX, Sigma] = GaussianMixtureModel( X, K, tol, display, Mu, Sigma)
% Uses the EM algorithm to use the Gaussian Mixture model to fit to the
% data
% INPUTS:       X: M x N points of data to be segmented using a gaussians
%               K: Number of clusts. Default set to 3.
%              Mu: Optional inital conditions for the starting cluster means
%           Sigma: Optional inital starting cluster variance
% OUTPUTS:      Z: Contains the centroid vector for each data point
%              Mu: Defines the cluster means for each centroid vector
%              Lk: The Likelihood
%             PKX: Matrix (N X K) The probability of point X_n to be in
%             cluster k
%           Sigma: Variance matrices of the K gaussians

[M,N] = size(X); % Get dimensions of data and gaussians to be fit

if (nargin < 2) K = 3; end              % If K is not given
if (nargin < 3) tol = 1e-4; end         % If tol is not given
if (nargin < 4) display = false; end    % If display is not specified
if (nargin < 5) Mu = zeros(M,K); iMu = true; end    % If Mu not given
if (nargin < 6) Sigma = zeros(M,M,K); iSigma = true; end   % If Sigma not given

PKX = zeros(N,K); % Centroid vector for each X point
Pk = ones(K,1); % To store prior probabilities
Z = zeros(N,1);  % Inicator for each X point

% STARTING CONDITIONS: Equally spaced cluster means
if iMu == true
    for m = 1:M
        mmax = max(X(m,:));
        mmin = min(X(m,:));
        mdiff = mmax - mmin;
        Mu(m,:) = mdiff*rand(1,K);
    end
end
if iSigma == true
    for m = 1:M
        Sigma(m,m,:) = 10^(-M)*ones(1,1,K);
    end
end

% Variables to compare if clusters have stopped moving
Mu_new = Mu;
Sigma_new = Sigma;
iters = 0;
n1 = 0;
while (true)
    if (display) tic; end % Start the clock for each iteration if display is on

    % E-Step: Compute expectations
    for k = 1:K
        sqrt_Sigma = 1/ sqrt( det( Sigma(:,:,k) ) );
        Xminus =  X - Mu(:,k);
        invSX = -0.5 * (Sigma(:,:,k) \ Xminus);
        PKX(:,k) = Pk(k) * sqrt_Sigma * exp( dot(Xminus,invSX,1));
    end
    PKX = PKX ./ sum(PKX,2);  % Normalize posterior

```

```
    for k = 1:K
        Pk(k) = sum(PKX(:,k)) / N;
    end

    % M-Step: Maximise parameters values based on the likelihood function
    for k = 1:K
        Mu_new(:,k) = 0; % Compute new values for Cluster centers
        PKXX = X;
        for m = 1:M
            PKXX(m,:) = PKX(:,k) .* X(m,:)';
        end
        Mu_new(:,k) = Mu_new(:,k) + sum( PKXX, 2) / sum( PKX(:,k) );

        Sigma_new(:,:,k) = 0; % Compute new values for Cluster variances
        Xminus = X - Mu_new(:,k);
        PKXminus = Xminus;
        for m = 1:M
            PKXminus(m,:) = PKX(:,k) .* Xminus(m,:)';
        end
        Sigma_new(:,:,k) = Sigma_new(:,:,k) + PKXminus * Xminus' / sum( PKX(:,k) );
        % Incase sigma is a non invertible matrix
        if rcond(Sigma_new(:,:,k)) < 10^(-5*M)
            for m = 1:M
                Sigma_new(m,m,k) = 1;
            end
        end
    end

    n2 = norm(Mu_new - Mu); % norm to check if the solution has converged
    if (n2 <  tol) || (abs(n1 - n2) < tol) % Stop once converged
        Mu = Mu_new;
        Sigma = Sigma_new;
        break
    else
        n1 = n2;
        Mu = Mu_new;
        Sigma = Sigma_new;
    end

    if (display) % Display variables if asked for
        toc
        display(n2)
        iters = iters + 1;
        display(iters)
    end
end


% Get indicator variables
for n = 1:N
    [~,index] = max(PKX(n,:));
    Z(n) = index;
end
% Compute LogLikelihood
LK = 0;
for k = 1:K
    LK = LK - Pk(k) * N * log( det( Sigma(:,:,k) ) ) / 2;
    Xminus =  X(:,:) - Mu(:,k);
    invSX = (Sigma(:,:,k) \ Xminus);
    LK = LK -  (1/2) * sum( PKX(:,k)' .* dot(Xminus,invSX,1));
    LK = LK + 0.5 * Pk(k) * N * log(Pk(k));
%     LPKX = -log(PKX(:,k));
%     LPKX(LPKX == inf) = 0;
%     LK = LK + sum( PKX(:,k) .* (LPKX - (M/2) * log(2*pi) ) );
%     LK = LK + sum( PKX(:,k) .* (LPKX ) );
end

end
```

`test_Kmeans_imageSegmentation.m`.

Note `test_GMM_imageSegmentation.m` is very similar and just calls the GMM function instead of the K means one.

```
img = imread('FluorescentCells.jpg'); % Load the image and format it
img = double(img);
[Nx,Ny,M] = size(img);
X = reshape(img, Nx*Ny,M)';
X = X/255;
% Initalize arrays needed to store data
K_vals = 2:10;
Nk = length(K_vals);
Z = cell(Nk,1);
Mu = cell(Nk,1);
EK = zeros(Nk,1);
TimeTaken = zeros(Nk,1);
% Compute K Means clustering for each value of K
for K = K_vals
    tic;
    [Z{K-1},Mu{K-1},EK(K-1)] = Kmeans(X,K);
    TimeTaken(K-1) = toc
end
```

```
% Plot results
figure();
subplot(1,4,1);
plotImage(Z{2-1},Mu{2-1},2,Nx,Ny);
title(['Image Segmentation using K-Means, K: ',num2str(2)]);
subplot(1,4,2);
plotImage(Z{3-1},Mu{3-1},3,Nx,Ny);
title(['Image Segmentation using K-Means, K: ',num2str(3)]);
subplot(1,4,3);
plotImage(Z{5-1},Mu{5-1},5,Nx,Ny);
title(['Image Segmentation using K-Means, K: ',num2str(5)]);
subplot(1,4,4);
plotImage(Z{10-1},Mu{10-1},10,Nx,Ny);
title(['Image Segmentation using K-Means, K: ',num2str(10)]);

figure();
plot(K_vals,EK,'x-');
title('Normalized Squared Sum of Errors vs K'); xlabel('K'); ylabel('Normalized Squared Sum of Errors')
```

test_GMM_segmentationCrossValidation.m

```
% Load the image
img = imread('FluorescentCells.jpg');
% imshow(img);
img = double(img);
[Nx,Ny,M] = size(img);
N = Nx*Ny;
X = reshape(img, N,M)';
X = X/255;

Xperm = X(:,randperm(length(X),length(X)));
Xtest = Xperm(:,1:N/2);
Xtrain = Xperm(:, N/2+1:N);

K_vals = 2:10;
Nk = length(K_vals);
Z = cell(Nk,1);
Ztest = cell(Nk,1);
Mu = cell(Nk,1);
PKX = cell(Nk,1);
Sigma = cell(Nk,1);
LK = zeros(Nk,1);
LKtest = zeros(Nk,1);
TimeTaken = zeros(Nk,1);

for K = K_vals
    tic;
    [Z{K-1},Mu{K-1},LK(K-1),PKX{K-1}, Sigma{K-1}] = GaussianMixtureModel(Xtrain,K);
    TimeTaken(K-1) = toc
    pkx = PKX{K-1};
    Pk = zeros(K,1);
    for k = 1:K
        Pk(k) = sum(pkx(:,k)) / length(pkx(:,k));
    end
    [ Ztest{K-1} , LKtest(K-1)] = GMMTest( Xtest, K, Mu{K-1}, Sigma{K-1}, Pk);

end

figure(); hold on;
plot(K_vals,-LK,'x-','DisplayName','Training fit');
plot(K_vals,-LKtest,'x-','DisplayName','Testing fit');
title('GMM: Negative LogLikelihood vs K'); xlabel('K'); ylabel('Negative LogLikelihood')
legend('show')
```

test_CellCounting.m

```
% Load the image
img = imread('FluorescentCells.jpg');
% imshow(img);
img = double(img);
[Nx,Ny,M] = size(img);
N = Nx * Ny;
X = reshape(img, N ,M)';
X = X/255;
K = 3;

load('Kmeans_counting.mat'); % Has the data stored for the segmentation with K = 3
tic;
% [Z1,C1] = GaussianMixtureModel(X,K,1e-3); % Data can also be found using
toc
Z = Z1;
C = C1;

Z = reshape(Z, Nx, Ny);

S = [];
Xp = []; Yp = [];
for x = 1:Nx
    for y = 1:Ny
        if Z(x,y) == 2
            S = [S, [x;y]];
            Xp = [Xp, x];
```

```
            Yp = [Yp, y];
        end
    end
end
S(1,:) = S(1,:) / Nx;
S(2,:) = S(2,:) / Ny;

K_vals = 10:10:100;
Nk = length(K_vals);
Z = cell(Nk,1);
Mu = cell(Nk,1);
LK = zeros(Nk,1);
TimeTaken = zeros(Nk,1);

for i = 1:Nk
    K = K_vals(i);
    tic;
    [Z{i},Mu{i},LK(i)] = GaussianMixtureModel(S,K,1e-3);
    timet = toc
    TimeTaken(i) = timet;
end

BIC = N * log(LK / N) + K_vals.* log(N);

figure();
subplot(1,3,1);
plotCellSegmentation(Xp,Yp,Z{i},K,640*Mu{i}, false)
title('GMM: Cluster Centers. K = 100'); xlabel('x'); ylabel('y')
axis([0 Nx 0 Ny])
subplot(1,3,2);
plotCellSegmentation(Xp,Yp,Z{i-2},80,640*Mu{i-2}, false)
title('GMM: Cluster Centers. K = 80'); xlabel('x'); ylabel('y')
axis([0 Nx 0 Ny])
subplot(1,3,3);
plotCellSegmentation(Xp,Yp,Z{i-4},60,640*Mu{i-4}, false)
title('GMM: Cluster Centers. K = 100'); xlabel('x'); ylabel('y')
axis([0 Nx 0 Ny])

figure();
plot(K_vals,-LK,'x-');
title('GMM: Negative LogLikelihood vs K (Cell Counting)'); xlabel('K'); ylabel(' Negative LogLikelihood')
```

`KmeansTest.m`

for segmenting the testing data with given clusters.

```
function [ Z , EK] = KmeansTest( X, K, Mu )
% Segments the inputted test data into K clusters with means Mu
% INPUTS:       X: M x N points of data to be segmented using a gaussians
%               K: Number of clusts. Default set to 3.
%               Mu: Cluster centers
% OUTPUTS:      Z: Contains the centroid vector for each data point
%               EK: Goodness of Clustering

[~,N] = size(X); % Extract the size of K

dist = zeros(N,K); % Assign to cluster with smallest L2 norm
for k = 1:K
    dist(:,k) = sum((X(:,:) - Mu(:,k)).^2,1);
end
[~,Z] = min(dist,[],2);

EK = 0; % Compute Goodness of fit
for k = 1:K
    EK = EK + sum( sum( (X(:,Z == k) - Mu(:,k)).^2));
end
EK = EK/N;

end
```

`GMMTest.m`

for segmenting the testing data with given clusters means and variances.

```
function [ Z , LK, PKX] = GMMTest( X, K, Mu, Sigma, Pk )
% Segments the inputted test data into K clusters with means Mu
% INPUTS:       X: M x N points of data to be segmented using a gaussians
%               K: Number of clusts. Default set to 3.
%               Mu: Cluster means
%              Sigma: Cluster varainces
%                Pk: Prior probabilities of each cluster
% OUTPUTS:      Z: Contains the centroid vector for each data point
%               LK: Negative Log Likelihood
%                Probability vectors

[M,N] = size(X); % Get dimensions of data and gaussians to be fit

PKX = zeros(N,K); % Centroid vector for each X point
Z = zeros(N,1);  % Inicator for each X point

```

```
for k = 1:K % Compute Posterior
    sqrt_Sigma = 1/ sqrt( det(Sigma(:,:,k) ) );
    Xminus =  X - Mu(:,k);
    invSX = -0.5 * (Sigma(:,:,k) \ Xminus);
    PKX(:,k) = Pk(k) * sqrt_Sigma * exp( dot(Xminus,invSX,1));
end
PKX = PKX ./ sum(PKX,2); % Normalize

% Get indicator variables
for n = 1:N
    [~,index] = max(PKX(n,:));
    Z(n) = index;
end

% Compute LogLikelihood
LK = 0;
for k = 1:K
    LK = LK - Pk(k) * N * log( det( Sigma(:,:,k) ) ) / 2;
    Xminus =  X(:,:) - Mu(:,k);
    invSX = (Sigma(:,:,k) \ Xminus);
    LK = LK -  (1/2) * sum( PKX(:,k)' .*  dot(Xminus,invSX,1));
    LK = LK + 0.5 * Pk(k) * N * log(Pk(k));
%     LPKX = -log(PKX(:,k));
%     LPKX(LPKX == inf) = 0;
%     LK = LK + sum( PKX(:,k) .* (LPKX - (M/2) * log(2*pi) ) );
end

end
```

4.3.2 Q2

`testGP.m`

Produces the plot for the periodic gaussian process fit using the functions defined below.

```
data = csvread('../CO2_Mauna_Loa_Data.csv',1);  % load the data in the file

% For ease of reading
X = data(:,1);
X = X/12 + 1960; % Convert to Years
y = data(:,2);
sigma_n = sqrt(var(y));

nx = length(X); % Extract length of data
n_xtest = 1e3;  % Length of GP extrapolation points over the domain
Nstarts = 1;    % Num starting points sampled from a hyperbolic distribution to achieve fmin

ox = 2013;  % Year of observation
% ox = 645; % Num months between 01/01/1960 and 01/01/2013
oy = 400;   % ppmm level

% Create a vector fom the smallest x value to the observation point
xtest = linspace( min(X), ox, n_xtest);
% Compute Hyperparameters
theta = computeHP(X, y, Nstarts);
% Fit test data to the GP with the found hyperparameters
[ytest, bounds, K, kstar] = computeGP (@K_SE, X, y, theta, xtest);
% Plot Data
display(theta)
fig_main = figure(); hold on;
plot(ox, oy, 'rx', 'DisplayName', 'Observed value in 2013');
plotGP( X, y, xtest, ytest, bounds, 'b-' );
ylabel('CO2 in ppm'); xlabel('Year')
% axis([min(X) ox 310 400])
```

Computes the covariance matrix K as well as all the derivatives wrt the hyper-parameters

`K_SE.m`

```
function [K, d_l, dsigma_n, dsigma_f, d_f, d_l2] = K_SE (x1, x2, theta)
% Computes the covariance matrix K using a squared exponential +
% exponential sin squared and white noise kernels as well as all the
% derivatives with respect the inputted theta

if exist('theta','var') == 0 % make sure all parameters have beeb inputted
    theta = [];
end
if length(theta) < 5
    l2 = 1;
else
    l2 = theta(5);
end
if length(theta) < 4
    f = 0;
else
```

```
    f = theta(4);
end
if length(theta) < 3
    sigma_f = 1;
else
    sigma_f = theta(3);
end
if length(theta) < 2
    sigma_n = 0;
else
    sigma_n = theta(2);
end
if length(theta) < 1
    l = 1;
else
    l = theta(1);
end

% Covariance
K = sigma_f^2 * exp( - (x1-x2) ^2 / (2*l^2) );
% Derivatives
d_l = K * (l^-3) * (x1-x2)^2;
dsigma_f = 2 * sigma_f * exp( - (x1-x2) ^2 / (2*l^2) );
% If the Kernel has any periodic behaviour
if f > 0
    K = K + exp( - 2 * sin( (x1-x2)*f)^2 / l2^2);
    d_f =  exp( - 2 * sin( (x1-x2)*f)^2 / l2^2) * -(4 / l2^2) * sin((x1-x2)*f) * cos((x1-x2)*f) * (x1-x2);
    d_l2 = exp( - 2 * sin( (x1-x2)*f)^2  / l2^2) * (4*sin((x1-x2)*f)^2) / l2^3;
else
    d_f = 0;
    d_l2 = 0;
end
% The white noise terms in the Kernel
if x1==x2
    K = K + sigma_n^2;
    dsigma_n = 2*sigma_n;
else
    dsigma_n = 0;
end

end
```

`optimHP.m`

Based on the implementation in [1] but altered to handle more hyperparameters

```
function [nLogLikelihood, nLLDerivatives] = optimHP (logtheta,constants)
% Computes the Negative LogLikelihood for the Kernel Function and its
% various derivatives
theta = exp(logtheta); backupConstants = exp(constants);
toBeComputed = constants(6:10); % indicators
Ntheta = 1;

if toBeComputed(1) == 1
    l1 = theta(1);
    Ntheta = Ntheta + 1;
end
if toBeComputed(2) == 1
    sigma_n = theta(Ntheta);
    Ntheta = Ntheta + 1;
end
if toBeComputed(3) == 1
    sigma_f = theta(Ntheta);
    Ntheta = Ntheta + 1;
end
if toBeComputed(4) == 1
    f = theta(Ntheta);
    Ntheta = Ntheta + 1;
end
if toBeComputed(5) == 1
    l2 = theta(Ntheta);
    Ntheta = Ntheta + 1;
end

% Constants - whatever's left
if exist('l1') == 0
    l1 = backupConstants(1);
end
if exist('sigma_n') == 0
    sigma_n = backupConstants(2);
end
if exist('sigma_f') == 0
    sigma_f = backupConstants(3);
end
if exist('f') == 0
    f = backupConstants(4);
end
if exist('l2') == 0
    f = backupConstants(5);
end

n = constants(11);
X = constants(12:n+11); y = constants(n+12:end); y = y - mean(y);
```

```

% Compute the Kernel matrix and its derivative matrix wrt to the various
% hyper parameters
K = zeros(n); dKdl = zeros(n); dKds = zeros(n); dKdf = zeros(n); dKdw = zeros(n); dKdl2 = zeros(n);
for i = 1:n
    for j = 1:n
        [K(i,j), dKdl(i,j), dKds(i,j), dKdf(i,j), dKdw(i,j), dKdl2(i,j)] = K_SE (X(i),X(j),[l1 sigma_n sigma_f f l2]);
    end
end

% Factorize K using the fact that its positive semi definite
L = chol (K,'lower');
alpha = L'\(L\y);
invK = inv(K);
alpha2 = K\y; % (inv(K)*y) alpha from page 114, not page 19, of Rasmussen and Williams (2006)

% Log marginal likelihood and its gradient
logpyX = -y'*alpha/2 - sum(log(diag(L))) - n*log(2*pi)/2;
dlogp_dl = l1*trace((alpha2*alpha2' - invK)*dKdl)/2;
dlogp_ds = sigma_n*trace((alpha2*alpha2' - invK)*dKds)/2;
dlogp_df = sigma_f*trace((alpha2*alpha2' - invK)*dKdf)/2;
dlogp_dw = f*trace((alpha2*alpha2' - invK)*dKdw)/2;
dlogp_dl2 = l2*trace((alpha2*alpha2' - invK)*dKdl2)/2;

% Output Negative Log Likelihood and any derivatives of interest
nLogLikelihood = -logpyX; nLLDerivatives = [];
if toBeComputed(1) == 1
    nLLDerivatives = [nLLDerivatives -dlogp_dl];
end
if toBeComputed(2) == 1
    nLLDerivatives = [nLLDerivatives -dlogp_ds];
end
if toBeComputed(3) == 1
    nLLDerivatives = [nLLDerivatives -dlogp_df];
end
if toBeComputed(4) == 1
    nLLDerivatives = [nLLDerivatives -dlogp_dw];
end
if toBeComputed(5) == 1
    nLLDerivatives = [nLLDerivatives -dlogp_dl2];
end

end
```

`computeGP.m`

Computes the GP given training data, hyper-parameters and testing x values.

```
function [ytest, bounds, K, ytestvar] = computeGP (k, X, y, theta, xstar)
% Finds the mean vector and variances for the testing values when fitting
% to a Gaussian Process with parameters theta formed from training data
% X and y.
% OUTPUTS: ytest, predicted mean vector of the testing data
%          bounds, 95% confidence values
%               K, The kernel for theta and the training data
%       ytestvar, the variance computed for the testing data.

if nargin < 5, sigma_n = 0; end

% Compute Kernel
ym = mean(y); y = y - ym; % Make data to have zero mean.
N = size(X,1); Nxtest = length(xstar);
K = zeros(N); kstar = zeros(N,1);
for i = 1:N
    for j = 1:N
        K(i,j) = k(X(i),X(j),theta);
    end
end

% Factorize the Kernel for easier computations.
L = chol (K,'lower');
alpha = L'\(L\y);

% Compute bounds and estimate for xstar
ytestmean = zeros(Nxtest,1); ytestvar = zeros(Nxtest,1);
for q = 1:Nxtest
    for i = 1:N % Compute Kernel components
        kstar(i) = k(X(i),xstar(q),theta);
    end
    ytestmean(q) = kstar' * alpha; % Mean values
    v = L\kstar;
    ystar_noise = sigma_n^2;
    ytestvar(q) = k (xstar(q),xstar(q),theta) - v'*v + ystar_noise;
end
% Compute bounds to 2 SDs (i.e. 95%)
bounds = [ytestmean + 1.96 * sqrt(ytestvar) ytestmean - 1.96 * sqrt(ytestvar)] + ym;
ytest = ytestmean + ym;

end
```

`computeHP.m`

Computes the hyper-parameters for a GP given training data.

```
function [ theta ] = computeHP( X, y, Nrepeats)
% Computes the hyperparameters for inputted data X and y from Nrepeats
% OUTPUTS: theta: vector of hyperparameters [l1, sigma_n, sigma_f, f, l2]

if nargin < 3, Nrepeats = 1; end

n = length(X);                  % Set n
sigma_n = sqrt(var(y));         % Set noise variance
toBeComputed = [1 1 1 1 1]'; % 0 = known
% Log to prevents negative values)
constants = [log([1 sigma_n 1 1 1])'; toBeComputed; n; X; y];

options = optimset('GradObj','on');
% Several varied starting points for minimization function
l1_samp = hypSample ([30 40], Nrepeats); % for l
sf_samp = hypSample ([0.1 2], Nrepeats); % for sigma_f
sigmaN_samp = hypSample ([0.1*sigma_n 0.2*sigma_n], Nrepeats); % for sigma_n
f_samp = hypSample ([2 4], Nrepeats); % for frequency
l2_samp = hypSample ([0.9 1], Nrepeats); % for l
startVals = log([l1_samp sigmaN_samp sf_samp f_samp l2_samp]);

% Loop over the random starting values to find minimum negative log
% likelihood.
thetaVec = [];
for s = 1:Nrepeats
    [theta, fval] = fminunc(@(hyperparameters) optimHP(hyperparameters,constants), startVals(s,:), options);
    thetaVec = [thetaVec; fval startVals(s,:) theta];
end
thetaVec(:,5:end) = exp(thetaVec(:,5:end));             % Return data to non log format
thetaVec(:,1) = thetaVec(:,1)/max(abs(thetaVec(:,1))); % Normalize
thetaVec = sortrows(thetaVec);                          % Sort so smallest is first
theta = thetaVec(1,size(startVals,2)+2:end);            % Choose smallest

end
```

`plotGP.m`

creates plots of the computed Gaussian Process

```
function [ ] = plotGP( X, y, xtest, ytest, bounds, linestyle )
% Plots the data for a Gaussian Process.
% INPUTS:          X, y: Training Data
%         xtest, ytest: Fit of the Testing Data
%               bounds: Upper and lower 2SD bounds on the testing data
%            linestyle: Can decide the linestyle of the training data.
% OUTPUTS: Plot.

if nargin < 6
    linestyle = 'b+';
end

hold on

stdRegion(xtest,bounds);
plot (xtest,ytest,'c','DisplayName','Best Estimate');
plot (X,y,linestyle,'DisplayName','Data Points');
legend('show');
end
```

`stdRegion.m`

From [1]. Creates shaded bounds around the mean

```
% Fill a graph with standard deviations; t is nx1 and Range is nx2
% sC is the colour to use, in RGB vector format
% reaxisVar = 0 to leave the axes to be overdrawn, or 1 to redraw them
% Mark Ebden, 2008

function theRange = stdRegion (t, theRange, sC, reaxisVar)
if exist('sC') == 0
    sC = [.6 .6 .6];
end
if exist('reaxisVar') == 0
    reaxisVar = 0;
end
t = t(:);
% Create enclosed shape using bounds
fill ([t; flipud(t)], [theRange(:,1); flipud(theRange(:,2))], sC, 'EdgeColor', sC,'DisplayName','Bounds on GP');

if reaxisVar == 1
    v = axis;
    line (v([1 1]), v([3 4]), 'Color', 'k');
    line (v([1 2]), v([3 3]), 'Color', 'k');
end
```

`HypSample.m`

From [10]. Samples a hyperbola N times for a parameter with given bounds

```
% Sample a hyperbola N times for a parameter bounded by the two components of 'bounds'.
% Example usage: if p(x) = k/x from 0.1 to 4, and p(x) = 0 otherwise:
%                x = hypSample ([.1 4], 1e5); hist(x,100)
% Bounds can be calculated as per Section 5.5.1 in 'Data Analysis: A Bayesian Tutorial',
% 2nd edition, D.S. Sivia and J. Skilling, 2006.

function x = hypSample (bounds, N)

xmin = bounds(1); xmax = bounds(2);
F = rand(N,1);
x = xmin.^(1-F) .* xmax.^F;

end
```

4.3.3 Q4

`testingGeneral.m`

for generating the data visualisations at the start of Q4.

```
% Load and format data
load('bitcoinData.mat');
bid = string(bid);
bid = (bid == 'TRUE');
symbol = string(symbol);
exchange = string(exchange);

N = length(price);  % get length of data

% Plot the prices the transactions at bid and ask price occurred
fig_prices = figure(); hold on;
plot(date1(bid == 1),price(bid == 1),'rx', 'DisplayName', 'Bid Prices');
plot(date1(bid == 0),price(bid == 0),'g.', 'DisplayName', 'Ask Prices');
xlabel('Date Value'); ylabel('Price'); legend('show');
title('Bid and Ask Prices for Bitcoin Transactions')

U = unique(date1);      % Unique Days
NU = length(U);         % # Unique Days
NpDay = zeros(NU,1);    % N per Day
for day = 1:NU
    NpDay(day) = length(date1(date1 == U(day) ));
end

Bdate = date1(bid == 1);    % Dates of Bid transactions
Bprice = price(bid == 1);   % Price of Bid transactions
Bamount = amount(bid == 1); % Amount of Ask transactions
Adate = date1(bid == 0);    % Dates of Ask transactions
Aprice = price(bid == 0);   % Price of Ask transactions
Aamount = amount(bid == 0); % Amount of Ask transactions

% fig_next = figure(); hold on;
% subplot(1,2,1); plot(Bprice);
% subplot(1,2,2); plot(Aprice);

VpDay = zeros(NU,2);
AvgpDay = zeros(NU,2);

for day = 1:NU
    dayAmount = Bamount(Bdate == U(day));
    dayPrice = Bprice(Bdate == U(day));
    VpDay(day,1) = sum(dayAmount);
    AvgpDay(day,1) = sum(dayAmount.*dayPrice) / VpDay(day,1);

    dayAmount = Aamount(Adate == U(day));
    dayPrice = Aprice(Adate == U(day));
    VpDay(day,2) = sum(dayAmount);
    AvgpDay(day,2) = sum(dayAmount.*dayPrice) / VpDay(day,2);
end

% fig_next = figure(); hold on;
plot(U,AvgpDay(:,1),'x-r');
plot(U,AvgpDay(:,2),'x-g');

figure(); hold on;
plot(id(bid == 1),price(bid == 1),'rx');
plot(id(bid == 0),price(bid == 0),'gx');
xlabel('Price'); ylabel('ID'); title('Price (coloured in bid/ask) vs ID');
```

`testingRegression.m`

For generating the data regression plot visualisations

```
% Load Data
```

```
load('bitcoinData.mat');
bid = string(bid);
bid = (bid == 'TRUE');
symbol = string(symbol);
exchange = string(exchange);

X = date1;
y = price;

% Extract length of data
nx = length(X);
% Confidence interval size: 2SD = 95% for normal distribution
alpha = 0.05;

% Fit data
[ y_OLS, bounds, ~ ] = OLS(X, y, alpha);

% Plot data
figure(); hold on;
% plotRegression(X,y,X,y_OLS,bounds,'bx');
xlabel('Date'); ylabel('Price');
title('Simple Linear Regression with 2 standard deviation bounds');

U = unique(date1);      % Unique Days
NU = length(U);         % # Unique Days
NpDay = zeros(NU,1);    % N per Day
for day = 1:NU
    NpDay(day) = length(date1(date1 == U(day) ));
end

Bdate = date1(bid == 1);    % Dates of Bid transactions
Bprice = price(bid == 1);   % Price of Bid transactions
Bamount = amount(bid == 1); % Amount of Ask transactions
Adate = date1(bid == 0);    % Dates of Ask transactions
Aprice = price(bid == 0);   % Price of Ask transactions
Aamount = amount(bid == 0); % Amount of Ask transactions

% fig_next = figure(); hold on;
% subplot(1,2,1); plot(Bprice);
% subplot(1,2,2); plot(Aprice);

VpDay = zeros(NU,2);
AvgpDay = zeros(NU,2);

for day = 1:NU
    dayAmount = Bamount(Bdate == U(day));
    dayPrice = Bprice(Bdate == U(day));
    VpDay(day,1) = sum(dayAmount);
    AvgpDay(day,1) = sum(dayAmount.*dayPrice) / VpDay(day,1);

    dayAmount = Aamount(Adate == U(day));
    dayPrice = Aprice(Adate == U(day));
    VpDay(day,2) = sum(dayAmount);
    AvgpDay(day,2) = sum(dayAmount.*dayPrice) / VpDay(day,2);
end

AvgpDayT = sum(VpDay.*AvgpDay,2)./sum(VpDay')';

[ y_OLS_dayAVG, bounds, ~ ] = OLS(U, AvgpDayT, alpha);

% Plot data
plotRegression( U, AvgpDayT, U, y_OLS_dayAVG, bounds, 'bx');
plot (X,y,'k.');
```

OLS

For generating the data regression plot visualisations

```
function [ fitted_data, bounds, beta_coeffs, sigma ] = OLS(x, y, alpha)
%OLS - Ordinary Least Squares
%   Detailed explanation goes here

if nargin < 3, alpha = 0.05; end

% Extract length of the data of interest
N = length(x);

xm = mean(x);
ym = mean(y);

beta1 = sum((x - xm).*(y-ym));
beta1 = beta1 / sum( (x - xm).^2 );
beta0 = ym - beta1 * xm;
beta_coeffs = [beta0, beta1];
% Find the estimated values
fitted_data = beta_coeffs(2) * x + beta_coeffs(1);

r = fitted_data-y;
SSe = sum(r.^2);          % Squared sum of errors
var_e = SSe/(N-2);        % Variance of the error
sigma = sqrt(var_e);
tval = tinv(1 - alpha/2, N-2); % Student t value with given degrees of freedom and (1-alpha) Confidence interval
c = sqrt( 1 + 1/N + (( x - xm ).^2) / sum( ( x - xm ).^2 )  );
```

```
bound = tval * sigma * c;

bounds = [fitted_data +  bound, fitted_data - bound ];

end
```

`NeuralNetwork.py`

This file contains some of the initial python plots and some basic analysis of the system using iterative validation

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

def buildModel(NIL,NHL,NOL):

    # Initialize matrices and arrays to random values
    np.random.seed(0) # For testing purposes
    W1 = np.random.randn(NHL, NIL) / np.sqrt(NIL)
    c1 = np.zeros((NHL,1))
    W2 = np.random.randn(NOL, NHL) / np.sqrt(NHL)
    c2 = np.zeros((NOL,1))

    return { 'W1': W1, 'c1': c1, 'W2': W2, 'c2': c2}
#########################################################################

def trainModel(model, X, y, repeats = 1):
    W1, c1, W2, c2 = model['W1'], model['c1'], model['W2'], model['c2']
    # Compute solution and then propogate error back
    for i in range(repeats):
        # Forward propagation
        u1 = W1.dot(X) + c1
        v1 = np.tanh(u1)
        u2 = W2.dot(v1) + c2
        exp_scores = np.exp(u2)
        probs = exp_scores / np.sum(exp_scores)
        # Backpropagation
        delta3 = probs - y
        dW2 = delta3.dot(v1.T)
        dc2 = np.sum(delta3)
        delta2 = (W2.T).dot(delta3) * (1 - np.power(v1, 2))
        dW1 = np.dot(delta2,X.T)
        dc1 = np.sum(delta2)
        # Compute level of error to add
        dW2 += eps2 * W2
        dW1 += eps2 * W1
        # Add Error to the model matrices/arrays
        W1 += -eps1 * dW1
        c1 += -eps1 * dc1
        W2 += -eps1 * dW2
        c2 += -eps1 * dc2
    return { 'W1': W1, 'c1': c1, 'W2': W2, 'c2': c2}
#########################################################################

# Helper function to predict an output (0 or 1)
def predict(model, X):
    W1, c1, W2, c2 = model['W1'], model['c1'], model['W2'], model['c2']
    # Forward propagation
    u1 = W1.dot(X) + c1
    v1 = np.tanh(u1)
    u2 = W2.dot(v1) + c2
    exp_scores = np.exp(u2)
    probs = exp_scores / np.sum(exp_scores)
    return probs
#########################################################################
def moving_average(x, n) :
    ret = np.cumsum(x, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    for i in range(n-1):
        ret[i] /= (i+1)
    ret[n - 1:] /= n
    return ret
# Cite: http://stackoverflow.com/users/110026/jaime
#########################################################################

def testResult(model, Xtest, ytest):
# Specefic to this situation. Can be changed for different models
    probs = predict(model, Xtest)
    if np.argmax(probs) == np.argmax(ytest):
        return 1
    else:
        return 0
#########################################################################
def trainVals(model, start, num):
    for i in range(start, start+num):
        vals = np.arange(i-Ntau+1,i+1) # indices of prices of interest
        Xt = np.array([price[vals].T, bid[vals].T])
        Xt = Xt.reshape(NIL,1) # Make into a vector
        yt = np.array([priceUp[i], 1 - priceUp[i]]) # Vector of [up, down] booleans
        yt = yt.reshape((2,1))
        # Train the model of this value
```

```
        model = trainModel(model, Xt, yt)

    return model

#####################################################################
def testVals(model, start, num):
    correct = 0
    for i in range(start, start+num):
        vals = np.arange(i-Ntau+1,i+1)  # indices of prices of interest
        Xt = np.array([price[vals].T, bid[vals].T])
        Xt = Xt.reshape(NIL,1)  # Make into a vector
        yt = np.array([priceUp[i], 1 - priceUp[i]]) # Vector of [up, down] booleans
        yt = yt.reshape((2,1))
        # Compute if the result is correct
        correct += testResult(model, Xt, yt)
    return correct / float(num) # Return % of correct values

#####################################################################
if __name__ == '__main__':
    # Load and format the data
    my_data = np.genfromtxt('data.csv', delimiter = ',', dtype = None)

    ID = my_data[1:,0]
    ID = ID.astype(np.float)
    ID = ID[np.argsort(ID)]

    date = my_data[1:,3]
    date = date.astype(np.float)
    date = date[np.argsort(ID)]

    price = my_data[1:,4]
    price = price.astype(np.float)
    price = price[np.argsort(ID)]

    amount = my_data[1:,5]
    amount = amount.astype(np.float)
    amount = amount[np.argsort(ID)]

    bid = my_data[1:,6]
    bid[bid == 'TRUE'] = 1
    bid[bid == 'FALSE'] = 0
    bid = bid.astype(np.float)
    bid = bid[np.argsort(ID)]

    # Comment out the statements with np.argsort to see the impact
    # plt.plot(ID,price,  label = 'Bitcoin Price Data')
    # plt.legend(loc='upper right')
    # plt.xlabel('Transaction ID')
    # plt.ylabel('Price')
    # plt.show()

    # Different definition of if the price went up.
    # 1. Firstly if the next transaction value is higher on lower
    # Not very useful in terms of understanding anything because of financial fluctuations
    N = np.size(price)
    D1 = price[1:N] - price[:N-1] # Level 1 different in price
    priceUp1 = np.ones(N-1)
    priceUp1[D1 < 0] = 0 # I.e. if the price went up or down
    # 2. See if the average price in the next avgSize transactions is higher or lower
    avgSize = 50 # Chosen arbitarily
    movingAvg = moving_average(price, avgSize)
    ND = avgSize
    D = movingAvg[ND:] - movingAvg[:-ND]
    priceUp = np.ones(N-ND)
    priceUp[D < 0] = 0
    # Plot to visualize
    # plt.plot(ID,price, label = 'Bitcoin Price Data')
    # plt.plot(ID,movingAvg, '--r', label='Moving Average (50 Transaction)')
    # plt.legend(loc='upper right')
    # plt.xlabel('Transaction ID')
    # plt.ylabel('Price')
    # plt.show()

    Ntau = 100
    dim_X = 2
    NX = Ntau * dim_X

    # Parameters for Neural Network
    eps1 = 1e-6 # learning rate (chosen)
    eps2 = 1e-6 # regularization strength (chosen)
    NIL = NX # input layer size
    NHL = NIL
    NOL = 2 # output layer size

    # Initialize matrices and vectors for the model
    model = buildModel(NIL,NHL,NOL)

    # Basic testing if trained on roughly half the values is prediction on the
    # rest of the data set
    model = trainVals(model, Ntau,10000)
    print testVals(model, Ntau+10000, N-2*Ntau-10000)

    # # Iterative cross validation but without randomness dealt with.
    N_samples = 4000
    N_runs = N / N_samples
```

```
    # Train model on intial data set
    model = trainVals(model, Ntau, N_samples)
    # Store model performance in each validation set
    performance = np.array([])
    # Test model on the next N_sample values then train and repeat
    for run in range(1,N_runs):
        if run == N_runs: # Make sure all values are tested in the last batch
            N_samples += N - N_runs * N_samples
        # Compute perforamance
        performance = np.append(performance, testVals(model, Ntau + run*N_samples , N_samples) )
        if run != N_runs: # If not the last run then train the model further
            model = trainVals(model, Ntau + run*N_samples, N_samples)
    print performance
    print np.average(performance)

    plt.figure()
    plt.plot(ID,price)
    for run in range(1,N_runs):
        midval = run*N_samples+ N_samples/10
        plt.annotate( str( performance[run-1]) , ( ID[midval], 680))
        plt.plot((ID[run*N_samples], ID[run*N_samples]), (610, 690), 'k-')
    plt.xlabel('Transaction ID')
    plt.ylabel('Price')
    plt.show()
```

`NeuralNetwork.py`

Python file for K-folds cross validation

```
    perfs = np.array([])
    K_fold = 10
    # Set number of values in each fold
    N_samples = (N - Ntau - avgSize) / K_fold
    for K in range(K_fold):
        # Initialize matrices and vectors for the model
        model = buildModel(NIL,NHL,NOL)

        # Train model on 9 of the 10 data sets
        for run in np.random.permutation(K_fold):
            if run != K:
                model = trainVals(model, Ntau + run * N_samples, N_samples)
        # Test on the Kth data set
        performance = testVals(model, Ntau + K * N_samples , N_samples)
        perfs = np.append(perfs, performance)
    print perfs
    print np.average(perfs)

    plt.figure()
    plt.plot(ID,price)
    for K in range(K_fold):
        midval = K*N_samples+ N_samples / 10
        plt.annotate( str.format('{0:.3f}', perfs[K] ) , ( ID[midval], 680))
        plt.plot((ID[K*N_samples], ID[K*N_samples]), (610, 690), 'k-')
    plt.plot((ID[N-1], ID[N-1]), (610, 690), 'k-')
    plt.xlabel('Transaction ID')
    plt.ylabel('Price')
    plt.show()
```

`NeuralNetwork.py`

Python file for Iterative Batch Cross Validation with multiple runs

```
    perfs = np.array([])
    N_random = 10
    for rand in range(N_random):
        # Initialize matrices and vectors for the model
        model = buildModel(NIL,NHL,NOL)

        # Iterative cross validation:
        N_samples = 2000
        N_runs = N / N_samples
        # Train model on intial data set
        model = trainVals(model, Ntau, N_samples)
        # Store model performance in each validation set
        performance = np.array([])
        # Test model on the next N_sample values then train and repeat
        for run in range(1,N_runs):
            if run == N_runs: # Make sure all values are tested in the last batch
                N_samples += N - N_runs * N_samples
            # Compute perforamance
            performance = np.append(performance, testVals(model, Ntau + run*N_samples , N_samples) )
            if run != N_runs: # If not the last run then train the model further
                model = trainVals(model, Ntau + run*N_samples, N_samples)
        perfs = np.append(perfs, performance)

    perfs = np.reshape(perfs, (N_runs-1,N_random))
    avgPerfs = np.average(perfs, axis = 1)
    TotalPerf = np.average(avgPerfs)
    print TotalPerf

```

```
    N_samples = 2000
    plt.figure()
    plt.plot(ID,price)
    for run in range(N_runs-1):
        midval = (run+1)*N_samples+ N_samples/10
        plt.annotate( str.format('{0:.3f}', avgPerfs[run]) , ( ID[midval], 680))
        plt.plot((ID[(run+1)*N_samples], ID[(run+1)*N_samples]), (610, 690), 'k-')
    plt.xlabel('Transaction ID')
    plt.ylabel('Price')
    plt.show()
```

References

[1] M. Ebden. Gaussian Processes: A Quick Introduction. *ArXiv e-prints*, May 2015.

[2] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. Adaptative computation and machine learning series. University Press Group Limited, 2006. ISBN 9780262182539.

[3] Tesla self-driving demonstration. URL `https://www.tesla.com/en_GB/videos/autopilot-self-driving-hardware-neighborhood-short`. Accessed: March 2017.

[4] Google waymo. URL `https://waymo.com/press/`. Accessed: March 2017.

[5] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23(9):661–692, 2006. ISSN 1556-4967. doi: 10.1002/rob.20147. URL `http://dx.doi.org/10.1002/rob.20147`.

[6] Alberto Broggi, Claudio Caraffi, Pier Paolo Porta, and Paolo Zani. The single frame stereo vision system for reliable obstacle detection used during the 2005 darpa grand challenge on terramax. In *Intelligent Transportation Systems Conference, 2006. ITSC'06. IEEE*, pages 745–752. IEEE, 2006.

[7] G Pasaoglu, D Fiorello, A Martino, G Scarcella, A Alemanno, A Zubaryeva, and C Thiel. Driving and parking patterns of european car drivers-a mobility survey. 2012.

[8] Aurangzeb Khan, Baharum Baharudin, Lam Hong Lee, and Khairullah Khan. A review of machine learning algorithms for text-documents classification. *Journal of advances in information technology*, 1(1):4–20, 2010.

[9] Mckinsey: Capturing the advanced driver-assistance systems opportunity. URL `http://www.mckinsey.com/industries/automotive-and-assembly/our-insights/capturing-the-advanced` Accessed: March 2017.

[10] D. Sivia and J. Skilling. *Data Analysis: A Bayesian Tutorial*. Oxford science publications. OUP Oxford, 2006. ISBN 9780198568315.