COMP4360 Homework #2:

Jacob Diaz (#7863183)

Answer to question 1:

Q1)a)

We can assume that the data has a noise contaminating it of a Gaussian nature.

The Gaussian Density Function:

$$p(x) = N(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} e^{(-\frac{(x-\mu)^2}{2\sigma^2})}$$

So, for every arbitrary point in our regression function, we assume the real data to fall around that point within a Gaussian distribution. The question is that how do we work this in reverse, using this type of error assumption to calculate an error function that reaches a more accurate fit?

Given data $D = (\mathbf{x}_n, t_n), n = 1, 2, ..., N$ and model parameters $\mathbf{w}$, each $\mathbf{x}_n$ is a piece of data, and each t_n is the model's expected output. we wish to ask: What are the model parameters that give us the highest possible probability that our dataset could have been generated by our model?

We can start with basic probability. The probability that a single matching piece is generated by our data is $p(\mathbf{x}_n, t_n|\mathbf{w})$. Simple probability laws tell us that the probability of all data points must be the product of all their associated probabilities. Let us create a likelihood function $L(\mathbf{w})$ such that:

$$L(\mathbf{w}) = \prod_{n=1}^{N} p(\mathbf{x}_n, t_n|\mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w}) p(\mathbf{x}_n|\mathbf{w})$$

The last equality is derived from probability rules. In an intuitive sense: the probability that we will get an appropriate t_n and $\mathbf{x}_n$ given $\mathbf{w}$ is the same as the probability we get appropriate $\mathbf{x}_n$ given $\mathbf{w}$ and appropriate t_n given $\mathbf{x}_n, \mathbf{w}$. We can further note that the values of $\mathbf{x}_n$ do not depend on $\mathbf{w}$ like t_n does(because t_n is a function of our model subject to the model weights), so we can drop it from the probability:

$$L(\mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w})p(\mathbf{x}_n)$$

We now move onto log likelihood. We first choose the natural log, because of it's relation to the Gaussian density function. We can immediately see some very important facts:

1) ln is a monotonically increasing function on it's domain.

2) ln turns multiplication into addition.

Since ln is monotonically increasing, we can conclude that any determined maximal value in $ln(L(\mathbf{w}))$ must be the same as the maximal value in $L(\mathbf{w})$. We can also reverse this and note that it also applies for the minimal value of $-ln$. So we define an error function to be $E(\mathbf{w}) = -ln(L(\mathbf{w}))$. Observe:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} ln(p(t_n|\mathbf{x}_n, \mathbf{w})) - \sum_{n=1}^{N} ln(p(\mathbf{x}_n)) = -\sum_{n=1}^{N} ln(p(t_n|\mathbf{x}_n, \mathbf{w}))$$

Notice how all multiplication turned to addition/subtraction and in the final equality we removed a term because the probability $p(\mathbf{x}_n)$ clearly does not depend on $\mathbf{w}$.

So when we construct our model with the assumption of a Normal error on the desired model output: $t_n = F(\mathbf{x}_n, \mathbf{w}) + \epsilon_n$ where $\epsilon_n \sim N(0, \sigma^2)$, and we have our set of corresponding model predictions $y_n = F(\mathbf{x}_n, \mathbf{w})$, we can re-frame the minimization of our error function, as the minimization of the distance between our predicted and desired function outputs by using the original Gaussian distribution formula we gave above:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} ln(p(t_n|\mathbf{x}_n, \mathbf{w})) = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(t_n - y_n(\mathbf{w}))^2$$

(noting that we can factor out constants, apply the natural logarithm to the e value, and reformat y_n as a function of $\mathbf{w}$)

This gives us a clear and direct path from the assumption of Gaussian distributed noise, to the utilization of the sum-of-squares error function.

Q1)b)

The "kernel trick" is a very interesting workaround that, with proper care/consideration, allows us to work with linear classifiers on some non-linear problems. It allows use a positive semi-definite kernel to calculate a higher dimensional dot product of two values within a lower dimensional space. It allows us to take advantage of the easy separability of data in higher dimensions, while also keeping the calculations within the lower dimensional feature space and therefore saving on computation time.

This core notion of higher dimensional separability is from Cover's Theorem. It follows from Cover's theorem that we can use high dimensional hyper planes to separate data that isn't linearly separable within it's low dimensional space. Some examples of these Kernels:

The Radial Basis Function: $K(\mathbf{x}, \mathbf{y}) = e^{-\gamma||\mathbf{x}-\mathbf{y}||^2}$

The Sigmoid Function: $K(\mathbf{x}, \mathbf{y}) = tanh(\kappa\mathbf{x} \cdot \mathbf{y} + c)$

Q1)c)

It is important to note that correlation is a linear measure. In a sense, covariance is a measure of the direction of linear relation between two variables, and correlation is a standardized function of covariance that also measures of the strength of which two variables are linearly related. This relationship between correlation and covariance can be clearly seen by the fact that we directly calculate the correlation matrix from the covariance matrix.

With the supplied covariance matrix we can calculate the correlation matrix R:

$$S = \begin{bmatrix} 10.2^2 & -45.6 \\ -45.6 & 8.1^2 \end{bmatrix} \tag{1}$$

$$R_{2,1} = R_{1,2} = \frac{S_{1,2}^2}{S_1 S_2} = \frac{(-45.6)^2}{(10.2)^2(8.1)^2} \approx 0.3046$$

$$R = \begin{bmatrix} 1 & 0.3046 \\ 0.3046 & 1 \end{bmatrix} \tag{2}$$

So from this we can see that the two variables are both positively correlated, but not to a very strong degree (.3046). So there is somewhat of a sense of these variables increasing together and decreasing together, but it's not a strong correlation.

Two variable can be uncorrelated by using the correlation matrix to sphere the data, a process that mutually uncorrelates all variables. But while this process can and often does make data much more clean and uncorrelated, we can not extend this and go on to state that sphering our data makes it mutually independent. That would be a false claim. This is because, like stated above, correlation is a linear measure and therefore it cannot account for more complex non-linear relationships between variables. It is only when we can prove that two variables are not linear and non-linearly related that we can say they are independent of each other.

Q1)d)

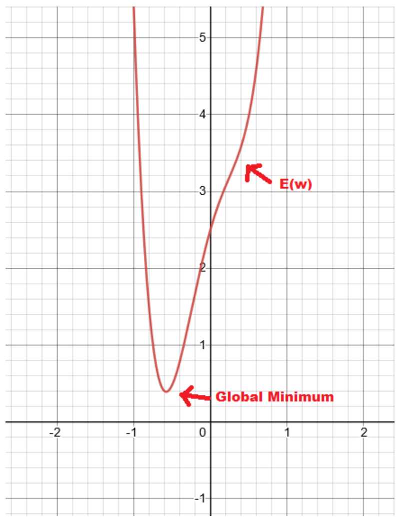
When we train an AI model for a particular set of data, we aim to see the model output an expected set of values corresponding to the inputs. When our model is trained our true test of validity is to see how the model interprets data it has never seen before, data that it was never tested on. We aim to see how well our model can generalize to new data, and the *generalization* *error* is our measure of how far off from the expected value our model performs on this new data.

When creating a model there is a sweet spot of both training methodology and hyperparameter settings that need to be reached to achieve a high level of generalization. If we do not train well, or our training data is skewed, biased, or incomplete, we will run into issues. And hyperparameters can also make or break a model. One notable issue that can arise is *overfitting*. Overfitting is when a model becomes too adjusted to it's test data, so much so that it only really works for it's test data, giving very high generalization error.

The most common way to combat overfitting is to make your model simpler. The intuition is that if we have a very large model with a massive amount of weights, we're in a sense imposing more constraints onto the output as a function of it's input and it's weights, leading to a much narrower area where the model can generalize past it's own test data. If we reduce the hyperparameters controlling model complexity (number of weights, number of hidden layers) we can aim to simplify how the model "looks" at it's test data in the hopes that it will act on new data in a similar way, i.e: much better generalization.

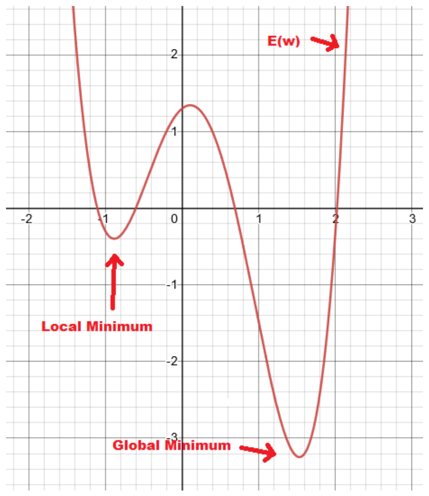
Q1)e)

In a formal sense, a function f is convex if it's second derivative is always positive. i.e: $\dfrac{df}{dx} > 0$. What this gives us in the graph is a lack of "dents" and concave features. The following function is an example of a convex function:



A very important fact about convex functions that make it extremely desirable as an error function is that convex functions only have one minimum, and this minimum is necessarily the global minimum. So if we were to train a model with a convex error function, then minimizing the error is a much clearer problem to solve.

If our error function was non-convex, we could very easily risk settling into a local minimum instead of a global minimum when using techniques like gradient descent. This is an example of a non-convex function with distinct local minimums:



Here we can clearly see that descending opposing sides of this function could lead us to settle into two different local minimums, only one being the global minimum. These graphs are within the $\mathbf{R}^2$ plane, but the analogy holds for any higher dimensional function.

Answer to question 2:

Q2)a)

I will describe the original perceptron model. Let $m \in \mathbf{N}$ be the number of input values

$\mathbf{x} = [x_1, x_2, ..., x_m]^T \in \mathbf{R}^m$, an input vector

$\mathbf{w} = [w_1, w_2, ..., w_m]^T \in \mathbf{R}^m$, the synaptic weights

$b \in \mathbf{R}$, a bias value

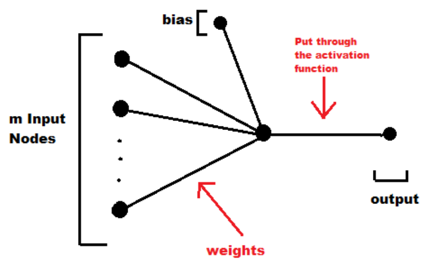
$\phi : \mathbf{R} \longrightarrow \mathbf{R}$ the activation function

We input an input vector of values, this is the vector the perceptron will act on. Each input value is multiplied by it's corresponding synaptic weight value, and all weighted inputs are summed. Then a bias value is added to adjust the final value to a more suitable range. This is captured in this formula:

$$v = (\mathbf{x} \cdot \mathbf{w}) + b = \sum_{n=1}^{m} w_i x_i + b$$

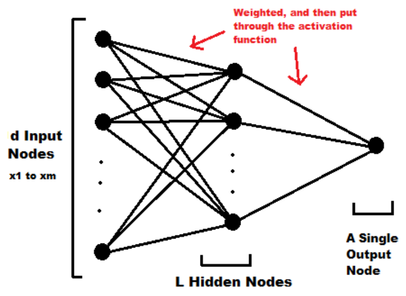
From here we need to send our v value through the activation function $\phi : \mathbf{R} \longrightarrow \mathbf{R}$ that intakes this calculated value and determines if the output should "activate". Because the perceptron model seeks to imitate the human brain, this activation function is almost a filter for the raw output of the neurons, possibly discarding unsure answers. Our final evaluated function is $\phi(v)$. The Sigmoid function on is a very common activation function, and this is understandable because of how it pushes values over 0.5 up towards 1, and pushes values under 0.5 down towards zero. This behaviour could make the output tend towards a more decisive conclusion.

Diagram:



Q2)b)

The three layer perceptron model is very similar to the two layer model. In this case we have an extra hidden layer, and a more complex usage of weights and biases. In general, we can call this Multi-Layer Perceptron (MLP) a function mapping the input space to a single output node $f : \mathbf{R}^d \longrightarrow \mathbf{R}$, where d is the size of the input vector. But the multilayer aspect of this gives a much more complex picture than the function could lead us to imagine. we have one "hidden" layer,



Now we almost have a full simple perceptron model leading into each individual hidden node. From this diagram we can lead into a more formal definition of the structure:

Let $d \in \mathbf{N}$ be the size of the input vector and let $L \in \mathbf{N}$ be the number of hidden nodes.

$\mathbf{x} = [x_1, x_2, ..., x_d]^T \in \mathbf{R}^d$, an input vector

$\mathbf{w} \in \mathbf{R}^{d \times L}$ is now an weight matrix from input to hidden layer

$\mathbf{a} = [a_1, a_2, ..., a_L]^T \in \mathbf{R}^L$, is a weight vector from hidden layer to output

$\phi : \mathbf{R} \longrightarrow \mathbf{R}$ the activation function

As for the bias values, it can vary. The class notes do not touch on bias values for MLP, but there can be associated bias values for each hidden node, as well as the output node. Sometimes it can be convention to add the bias value as the 0-th index node, with no corresponding input node. This is done to preserve clean formulae.

The evaluation of this MLP is as follows:

$$y = \phi\left(\sum_{i=0}^{L} a_i \phi\left(\sum_{j=1}^{d} w_{ji} x_j\right)\right)$$

The weight matrix serves to modify the input within the constraints of the activation function, almost like a large set of linear combinations. From the hidden nodes the values get another set of weights applied and a final activation function to make an output.

Q2)c)

We can first describe the error of a single sample like so:

$$e(k) = d(k) - \mathbf{x}(k) \cdot \hat{\mathbf{w}}(k)$$

Where $d(k)$ is the true value for the k-th sample, and $(\mathbf{x}(k) \cdot \hat{\mathbf{w}}(k))$ is what the model calculates for the k-th sample. We can see that this is a simple distance metric.

We can note that with a gradient descent training model we update our weights based on our error function:

$$\hat{\mathbf{w}}(k+1) = \hat{\mathbf{w}} - \mu \nabla J(\hat{\mathbf{w}}(k))$$

This says that our new weights are equal to our old weights with a small nudge in the direction of the negative gradient, or the steepest descent of the error at the certain point. This descent is controlled by a learning rate coefficient $\mu > 0$. This learning rate value modifies the size of our gradient vector being applied to our weights.

But what is our J function? We can construct it in a way that conveys our error function in a useful way like so:

$$J(\hat{\mathbf{w}}) = \frac{1}{2} e^2(k)$$

So when we differentiate it, we get:

$$\nabla J(\hat{\mathbf{w}}) = e'(k)e(k) = (d(k) - \mathbf{x}(k) \cdot \hat{\mathbf{w}}(k))' e(k) = -\mathbf{x}(k)e(k)$$

This is determined by the chain rule for multivariable functions. (applied in the first equality) If we feed this into our original weight update function, we can reach a more explicit definition for our method of updating all weights for a given sample.

$$\hat{\mathbf{w}}(k+1) = \hat{\mathbf{w}} + \mu \mathbf{x}(k)e(k)$$

And this is our Least-Mean-Squares algorithm for updating the weights in our model, derived from the notion of "stepping" down the steepest descent of our error function with a strength μ.

Answer to question 3:

Q3)a)

Please refer to question Q1)a where a very closely related topic was explained. Some of those ideas are the same here, and they'll be restated in this new context.

Assume we're given a set of pairs $\{(\mathbf{x}_n, t_n)\}$ of sample points $\mathbf{x}_n$ and their expected outcomes t_n. Where $1 \leq n \leq N, N \in \mathbf{N}$ being the number of pairs.

We can start with basic probability. The probability that a single matching piece is generated by our data is $p(\mathbf{x}_n, t_n|\mathbf{w})$. The probability of all data points being valid must be the product of all their associated individual probabilities. Let us create a likelihood function $L(\mathbf{w})$ such that:

$$L(\mathbf{w}) = \prod_{n=1}^{N} p(\mathbf{x}_n, t_n|\mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w})p(\mathbf{x}_n|\mathbf{w})$$

The last equality is derived from probability rules. In an intuitive sense: the probability that we will get an appropriate t_n and $\mathbf{x}_n$ given $\mathbf{w}$ is the same as the probability we get appropriate $\mathbf{x}_n$ given $\mathbf{w}$ and appropriate t_n given $\mathbf{x}_n, \mathbf{w}$. We can further note that the values of $\mathbf{x}_n$ do not depend on $\mathbf{w}$ like t_n does (because t_n is a function of our model subject to the model weights), so we can drop it from the probability:

$$L(\mathbf{w}) = \prod_{n=1}^{N} p(t_n|\mathbf{x}_n, \mathbf{w})p(\mathbf{x}_n)$$

We now move onto log likelihood. We choose the log function because of some very important properties:

1) *log* is a monotonically increasing function on it's domain.

2) *log* turns multiplication into addition in our formula.

Since *log* is monotonically increasing, we can conclude that any weight value $\mathbf{w}_0$ that maximizes $log(L(\mathbf{w}))$ must also maximize $L(\mathbf{w})$. We can also reverse this and note that it also applies for the minimal value of $-log$. So we define an error function to be $E(\mathbf{w}) = -log(L(\mathbf{w}))$ and we wish to minimize this error function. The principle of maximal likelihood says that if we can define a method of obtaining more optimal parameters, then we will increase out likelihood of reaching the hypothetical "true" model. Our motivation to minimize all error, dependent on our model parameters, coincides with this principle of maximal likelihood. Observe:

$$E(\mathbf{w}) = -\sum_{n=1}^{N} log(p(t_n|\mathbf{x}_n, \mathbf{w})) - \sum_{n=1}^{N} log(p(\mathbf{x}_n))$$

$$E(\mathbf{w}) = -\sum_{n=1}^{N} log(p(t_n|\mathbf{x}_n, \mathbf{w}))$$

Notice how all multiplication turned to addition/subtraction and in the final equality we removed a term because the probability $p(\mathbf{x}_n)$ clearly does not depend on $\mathbf{w}$. This is our final formula for our error function. We can see from this derivation how the properties of probability are applied to our model's outcomes, and how applying the likelihood principle can give us a much more approachable formula. I will finally just state each variable and it's purpose as requested:

N is the number of samples we are working with

$\mathbf{x}_n$ is the n-th sample within the summation

t_n is the n-th expected sample outcome value, dependent on the weight values and it's corresponding $\mathbf{x}_n$.

$\mathbf{w}$ is the current vector of weights for the model

Q3)b)

A set of Radial Basis functions (RBF) send our input data into a new space based on each point's distance to the closest RBF center.

The mean-squared error function (applied to regression) is a measure of how well a given regression line fits to a set of points. Applied to this context it is a linear measure of error in our model's output in relation to the set of values we want the model to output. So the key assumption is that our error can be accurately fitted with a linear function.

Q3)c)

With our network we have data points $\mathbf{x}_n$ (for $1 \le n \le N, N \in \mathbf{N}$), and associate weights w_m (for $1 \le m \le M, M \in \mathbf{N}, M < N$)

With this method we also have a set of basis functions $\phi_m(\mathbf{x})$ centered on M observations. So we can have our output given by:

$$y(\mathbf{x};\mathbf{w}) = w_0 + \sum_{m=1}^{M} w_n \phi(\mathbf{x})$$

Where $\mathbf{x}$ is our input data points.

This is similar to our evaluation of perceptrons. From this we can work backwards to determine our weights. Our weights calculations are given like so:

$$Z = \begin{bmatrix} 1 & \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \dots & \phi_M(\mathbf{x}_1) \\ 1 & \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \dots & \phi_M(\mathbf{x}_2) \\ 1 & \phi_1(\mathbf{x}_3) & \phi_2(\mathbf{x}_3) & \dots & \phi_M(\mathbf{x}_3) \\ \vdots & & & & \vdots \\ 1 & \phi_1(\mathbf{x}_N) & \phi_2(\mathbf{x}_N) & \dots & \phi_M(\mathbf{x}_N) \end{bmatrix} \tag{3}$$

Deriving from the initial equation for y above, We can multiply this matrix by our weights to then get the output, or target $\mathbf{t}$ of the model.

$$\mathbf{Z}\mathbf{w} = \mathbf{t}$$

from this we can measure the mean-squared error in this calculation, and attempt to minimize it. Mean squared error:

$$\frac{1}{2}||\mathbf{Z}\mathbf{w} - \mathbf{t}||^2$$

The minimization of mean-squared error:

$$\underset{\mathbf{w}}{argmin}\frac{1}{2}||\mathbf{Z}\mathbf{w} - \mathbf{t}||^2$$

We also very commonly place a regularization parameter in the function to be minimized. This term modifies the error function to favour simpler selections of weights based on a regularization parameter α:

$$\underset{\mathbf{w}}{argmin}\frac{1}{2}||\mathbf{Z}\mathbf{w} - \mathbf{t}||^2 + \frac{\alpha}{2}||\mathbf{w}||^2$$

from this we can finally derive a closed-form version of this minimization problem, and this common formula is as follows:

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X} + \alpha^2\mathbf{I}\right)^\dagger \mathbf{X}^T\mathbf{t}$$

Where † denotes the psudo-inverse. If the matrix **X** is not easily invertible then we can apply numerical analysis methods to achieve usable psudo-inverses. This is our final formula for the weights of this RBF model given the minimization of a mean squared error function. Without regularization our formula would simply be:

$$\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{\dagger}\mathbf{X}^T\mathbf{t}$$

Q3)d)

If the startup is facing issues with outliers then a switch to the sum of absolute errors function is a good idea. This is because when we consider the mean squared error function we can note that values with very high error will affect the model very greatly because of the presence of the square. If we remove this square operation in favour of an absolute value, the error function places less emphasis on the outliers. It is important to note that this change to a sum of absolute errors will also alter the way the error function interprets small error too, placing a greater emphasis on very small error due to how squaring small numbers < 1 behaves.

For example: A large error of 5 stays as 5 instead of becoming 25, and a small error of .5 would stay as .5 instead of becoming .25.

Q3)e)

In this new case with a sum of absolute error function, the expression to minimize would have to be:

$$\underset{\mathbf{w}}{argmin}\frac{1}{2}||\mathbf{Zw} - \mathbf{t}||$$

Answer to question 4:

Q4)a)

When we consider hierarchical clustering, the cores aspect is that we create a hierarchy of clusters with different resolutions as opposed to single partition clustering methods. A key benefit is the ability for the user to choose from a selection of partition resolutions, as well as a high generalization because the hierarchical clustering algorithms aren't based on vectors, they're based on a general dissimilarity measure. Hierarchical clustering is very flexible, but a key downside is that it often carries a heavier set of computations. I would say that a Hierarchical clustering algorithm would be good in a few cases:

1) When you are presented with very abstract data that isn't intuitive the generalizability of hierarchical clustering can help to making sense of it.

2) When you value project flexibility and interactivity over computation times. After all computation is done hierarchical clustering will have a more flexible analysis to offer to the user.

3) Much like the previous cases, when we are talking a more "exploratory" look at the data a hierarchical clustering method might give us a more comprehensive and generalized look at the data for us to then possibly take next steps with a different method.

Q4)b)

Agglomerative clustering stems from the word agglomerate, meaning "to collect into a group". It's the opposite of divisive clustering where we would take our cluster and breaking them into partitions iterativley. With agglomerative clustering we create N initial clusters corresponding to N data points, and from there we iterativley group them together to form the lower resolution clusters that are higher in the hierarchy.

With agglomerative clustering we start with an $N \times N$ dissimilarity matrix for all the data points. At every iteration we merge two clusters, deleting the corresponding rows and columns in the dissimilarity matrix while adding the new row and column for the new cluster dissimilarity in relation to all other clusters.

Q4)c)

We must first define the problem. Define our input data to be $X \subset \mathbf{R}^d$, where d is the size of our input data vectors. We wish to partition X into $K > 0 (K \in \mathbf{N})$ distinct clusters. K is a hyperparameter and is fixed. We represent our partition P like so: $P = \{C_1, ..., C_K\}$ where each $C_i \in P$ is a cluster and they abide by these properties:

1) $C_i \neq \emptyset, i = 1, ..., K$

2) $\bigcup_{i=1}^{K} C_i = P$

3) $C_i \cap C_j = \emptyset, i \neq j, i, j = 1, ..., k$

We also define a dissimilarity measure, in this case it will be the Euclidean metric. Our euclidean metric in this context is $|| \cdot || : \mathbf{R}^d \times \mathbf{R}^d \longrightarrow \mathbf{R}^+$. Also known as the L2-norm, and alternatively denoted $|| \cdot ||_2$. We can finally define our cluster representatives to be $\mu_j, j = 1, ..., K$. The cluster representative μ_j is the mean position of all data points belonging to cluster C_j.

We wish to minimize the squared distance between each cluster representative and its cluster's respective elements. We then sum this value across all clusters to reach our objective function J dependent on the partition P:

$$J(P) = \sum_{j=1}^{K} \left(\sum_{\mathbf{x}_i \in C_j} ||\mathbf{x}_i - \mu_j||_2^2 \right)$$

This is our final objective function we wish to minimize. With this method we favour clusters that are tightly packed and clearly differentiated from eachother.

Q4)d)

This is the k-means psudo-code:

```
Algorithm 1: K-Means Clustering
INPUT: A dataset $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\}$, hyperparameter $K$, and the loop limiter $M$
OUTPUT: A partition $P = \{C_1, ..., C_K\}$
1: $P = \emptyset, t = 0$
2: Randomly initialize each $\mu_i$ for $i = 1, ..., K$
3: loop
4:     $t+ = 1$
5:     Assignment Step: assign each sample $\mathbf{x}_j$ to the cluster with the closest representative
6:     $C_i^{(t)} = \{\mathbf{x}_j : d(\mathbf{x}_j, \mu_i) \leq d(\mathbf{x}_j, \mu_h) \forall h = 1, ..., K\}$
7:     Update Step: update all cluster representatives and clusters
8:     $\mu_i^{(t+1)} = \frac{1}{C_i^{(t)}} \sum_{\mathbf{x}_j \in_i} \mathbf{x}_j$
9:     $P^t = \{C_1^{(t)}, ..., C_K^{(t)}\}$
10:    if $t \geq M$ OR $P^t = P^{t-1}$ then:
11:        return $P^t$
12:    end if
13: end loop
```

Working down the psudo-code lines, we first intake our set of input vectors X, the hyperparameter K that determines the number of clusters, and the hard stopping value for loop iterations M. Our initial lines initialize the partition, the loop counter, and randomly set our cluster representatives.

We loop while assigning each value to the cluster with the closest representative. After that we can finally recalculate the cluster representatives themselves to better represent the new cluster sets. finally, we construct the partition from all clusters. Our loop evaluation checks to see if we have either elapsed our number of permitted loops, or if we have reached the same partition value two loops in a row. If either had happened we break the loop and return the partition.

Q4)e)

When the k-means algorithm is run a key observation to note is that we initialize the cluster representatives randomly. This is what causes odd difference in behaviour between algorithm executions on the same data. The k-means algorithm can converge to different partitions based on the different starting conditions so the simplest way to make the algorithm completely deterministic is to remove the random cluster representative initialization. Instead we can allow the user to pass explicit cluster representative start values into the k-means function. The quality of the clustering will vary with different initial representative values, but the partition result will then be deterministic and if two users supply the same initial cluster representative values then they will get the same output.