Inf2B Coursework 1 Report

Task 1 – Anuran-Call analysis and classification

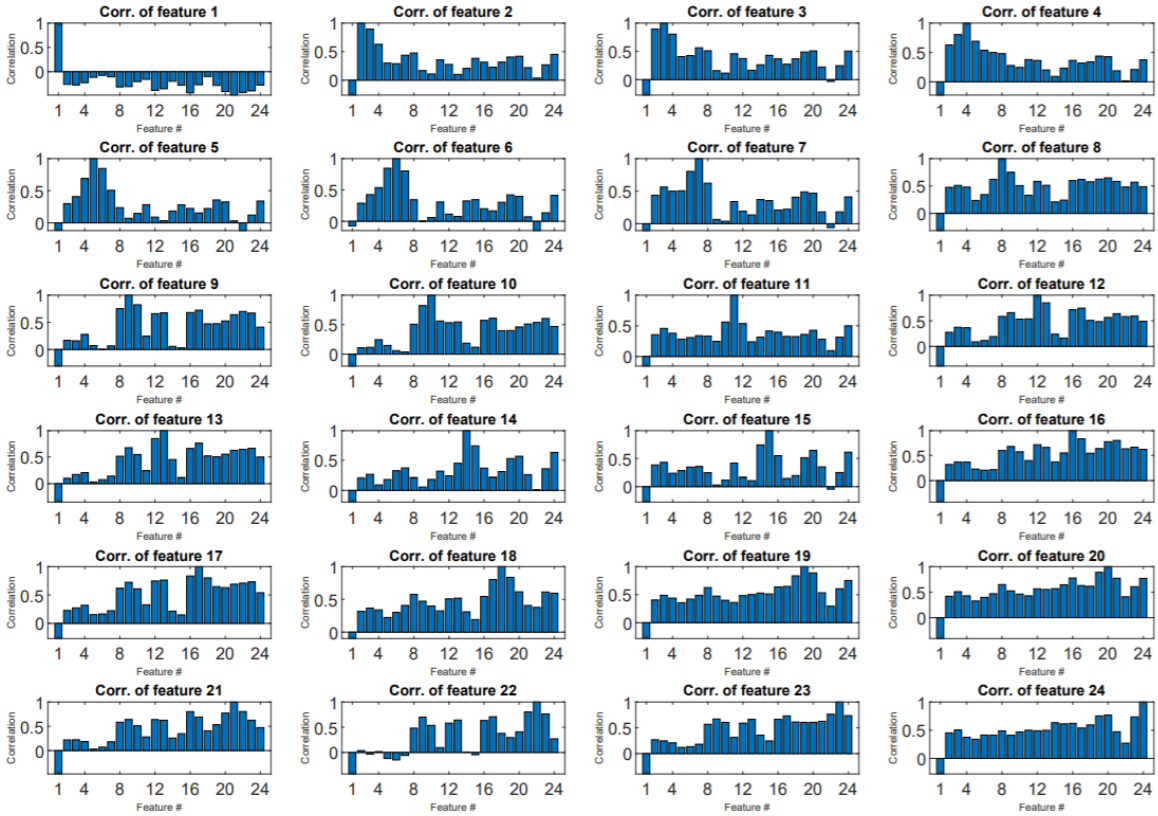
1.2) Findings from the correlation matrix:

Upon analyzing the data within correlation matrix R I found it would be useful to visualize the data in 2 ways:

1. A collection of subplots to show the relationships between all feature vectors
2. A bar graph to show the average correlation value for each feature vector

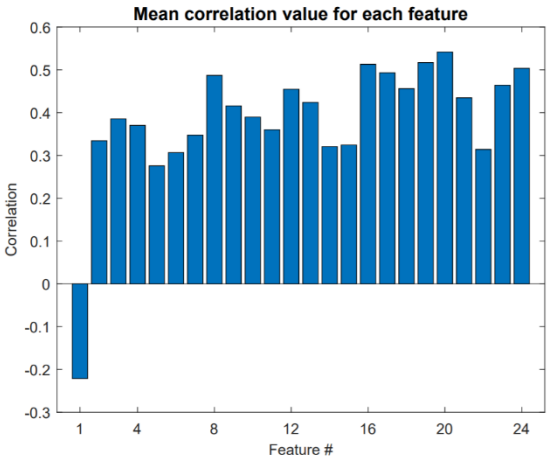
I used bar graphs to represent both visualizations so it would be easy to recognize the highest/lowest correlations.

A collection of subplots to show the relationships between all feature vectors in the data set X



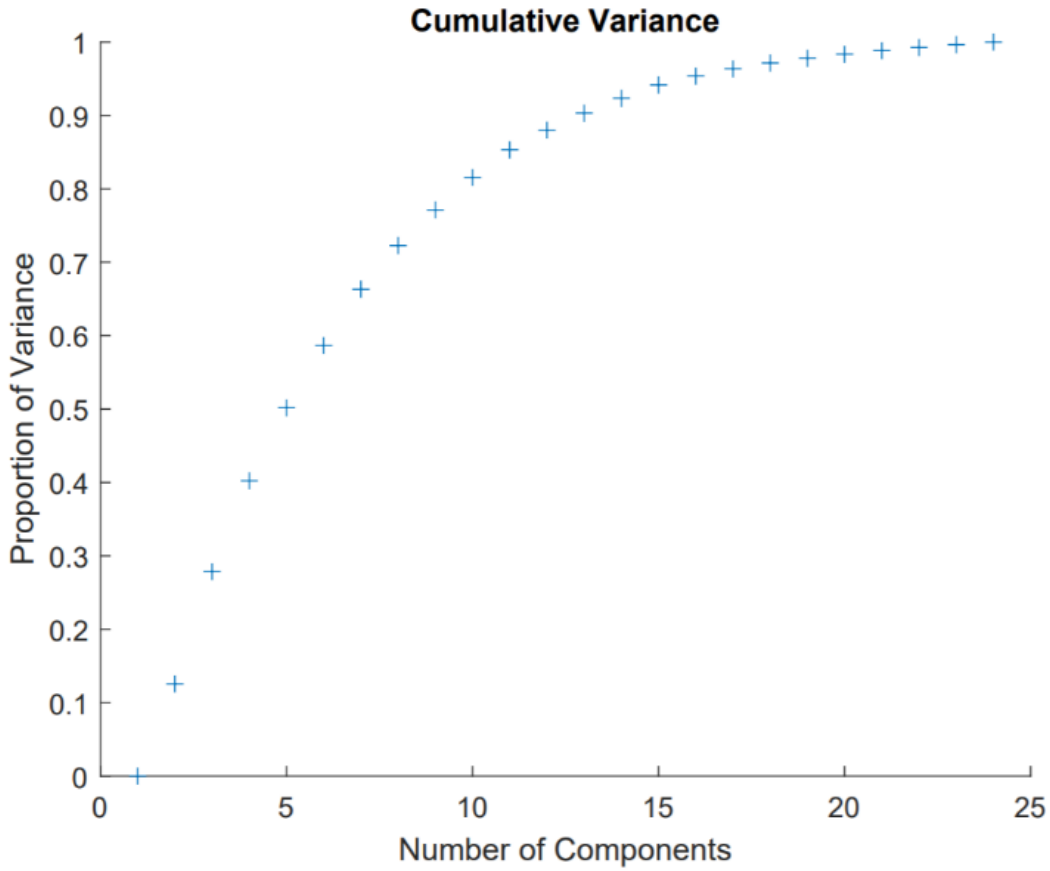
We must note that when analyzing these subplots all bars with correlation 1 represent the correlation between the same feature, and thereby do not represent anything significant.

This visualization of correlation matrix R is convenient for determining the nature of how a given feature is correlated to other features. This can be useful for making predictions about an incomplete sample (does not have data for all features), in which we can predict the value for a given missing feature by using the existing data in the sample with appropriate weightings (weighting for a given feature F is directly proportional to the correlation value between the missing feature and F).

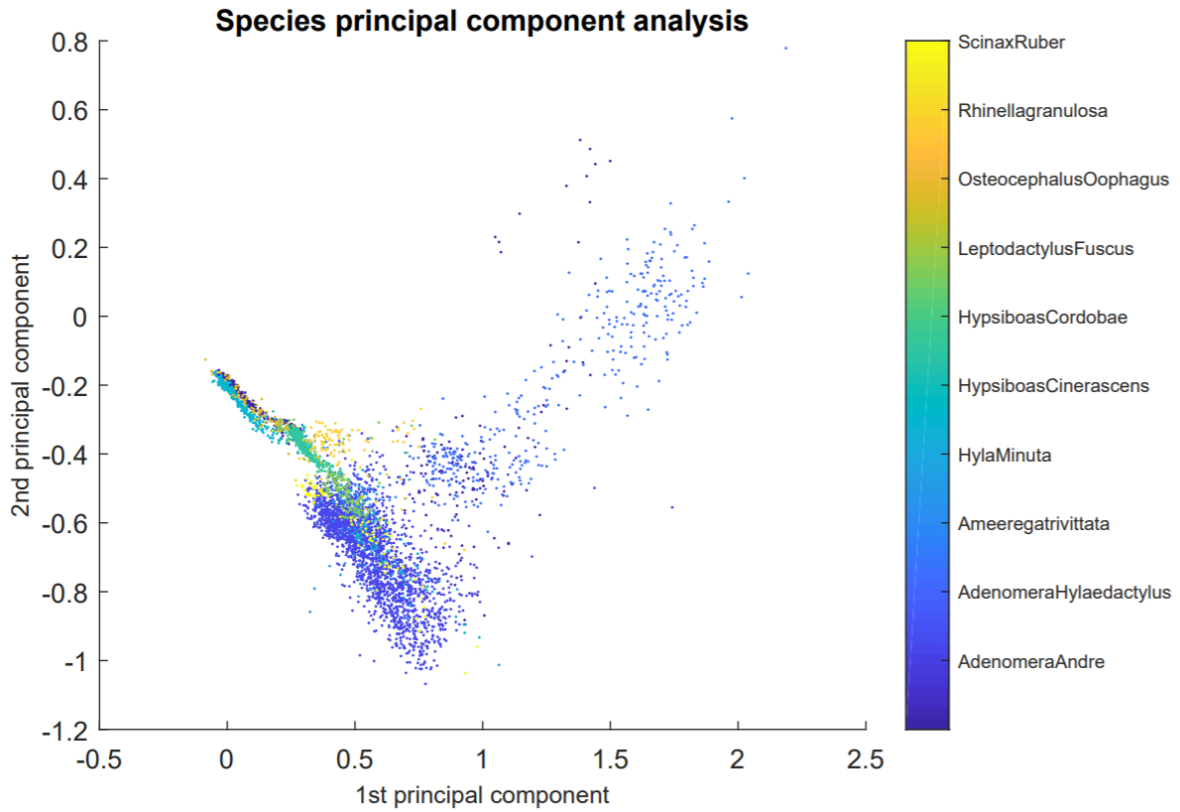


This visualization of correlation matrix R is convenient in order to determine which features are uniquely correlated. This is particularly useful when predicting feature values for an incomplete sample as it indicates the importance of normalization for each of the respective correlation values when equating weights.

1.3) b) Graph of cumulative variance



1.3) c) Plotting of data on 2D-PCA plane



1.4) b) Accuracy for CovKind = 1,2,3

	Overall accuracy for a given class										
CovKind	**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Class 6**	**Class 7**	**Class 8**	**Class 9**	**Class 10**	**Average**
1	0.3971	0.9972	0.9787	0.6035	0.8276	0.9919	0.8661	1	0.9920	0.9867	**0.8641**
2	0.2853	0.9823	0.7184	0.4000	0.0269	0.9899	0.8596	1	0.9444	0.5000	**0.6707**
3	0.3765	0.9963	0.9230	0.7139	0.8785	0.9879	0.8411	0.9765	0.9634	0.9267	**0.8584**
Average	**0.3529**	**0.9919**	**0.8733**	**0.5725**	**0.5777**	**0.9899**	**0.8556**	**0.9922**	**0.9666**	**0.8044**	**0.7977**

	Overall accuracy for a given partition					
CovKind	**Partition 1**	**Partition 2**	**Partition 3**	**Partition 4**	**Partition 5**	**Average**
1	0.9122	0.9170	0.9063	0.9229	0.9146	**0.9146**
2	0.7960	0.7995	0.7900	0.8233	0.7947	**0.8007**
3	0.9146	0.9110	0.8992	0.9336	0.8997	**0.9116**
Average	**0.8743**	**0.8758**	**0.8652**	**0.8932**	**0.8697**	**0.8756**

1.5) Classification accuracy VS epsilon

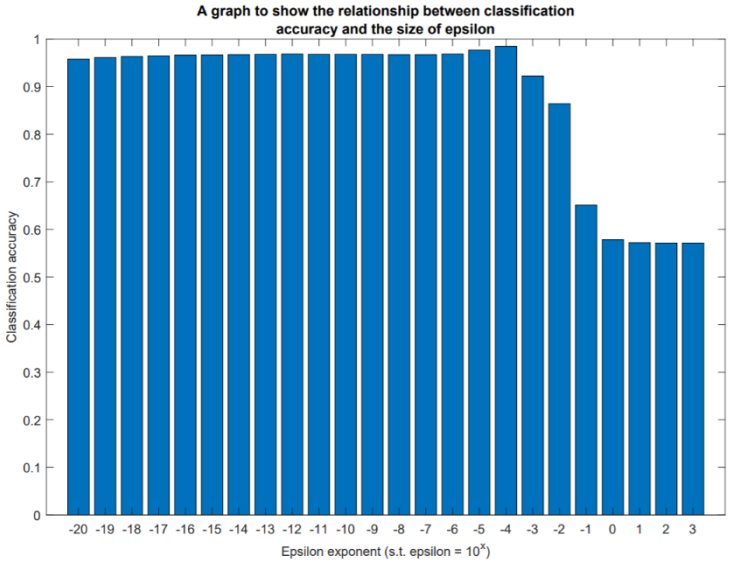
Since we are making predictions on our test samples using a multivariate Gaussian classifier, we must be wary about the stability of our statistical measures. They may become unstable due to anomalies in the data set, particularly in the implementation of the log likelihood equation due to its use of inverse covariance matrices:

in which $\sum^{-1}$ can become unstable when $|\sum|$ is small.

In order to minimize test error and make more accurate predictions we need a statistical learning method that simultaneously achieves low variance and low bias. In this case we used regularisation by adding a small positive number (epsilon) to the diagonal elements (which represent the variances) of the covariance matrix.

Upon this regularisation we must choose a value of epsilon that promotes optimal accuracy by balancing the bias-variance trade-off. In order to do this, we can test how different sizes of epsilon affect the overall classification accuracy.

We can see that for values of epsilon greater than or equal to 1 ($10^0 \leq \epsilon \leq 10^3$) that the classification accuracy lies below 60%. We can attribute this to prediction modelling with a high variance.



However, this classification accuracy exponentially increases when values of epsilon are less than 1. This is particularly evident in the range $10^{-17} \leq \epsilon \leq 10^{-3}$ in which the classification accuracy is greater than 90%.

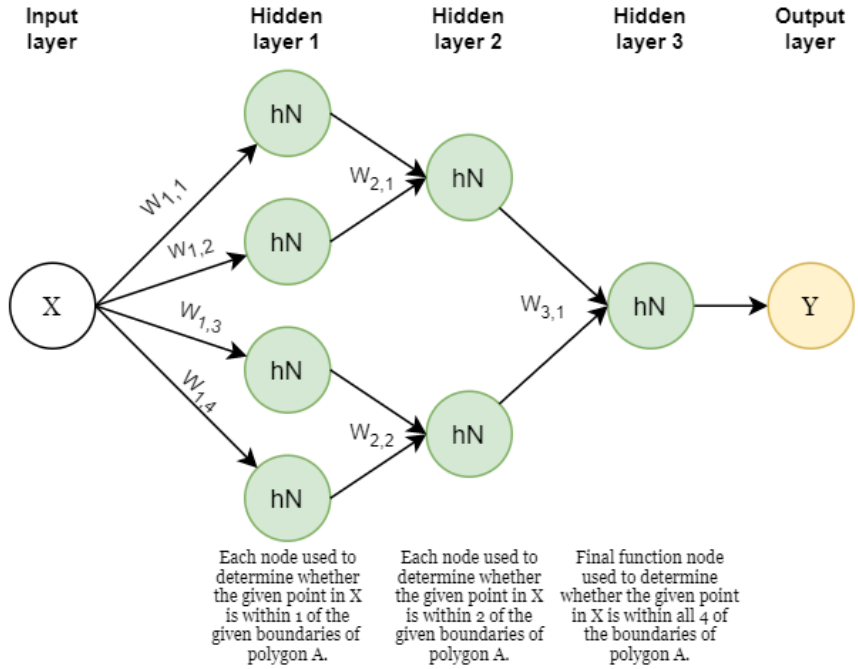
The optimal value of epsilon for this data model is that of $\epsilon = 10^{-4}$ with a classification accuracy of 98.45%. This suggests an optimal bias-variance ratio.

Overall, we can see that these classification accuracies continue to decrease on either side of the optimal value $\epsilon = 10^{-4}$ which tells us the nature of the associated bias-variance tradeoff.

Inf2B Coursework 1 Report

Task 2 – Neural networks

2.3) Structure of the neural network



How weights were determined:
For this given neural network weights are created in order to classify data correctly using the given activation function (hNeuron).

WEIGHTS: Input -> Hidden 1
(All calculations within *task2_find_hNN_A_weights.m*)

For the first layer of weights, the activation function classification is dependent on the coordinate of a point being within the boundaries of polygon A or not. Thereby we must calculate the functions of these boundaries and find how they can be converted to appropriate weights.

Given we are calculating weights we must ensure to account for every variable:

$$W_0(bias) = y - intercept \qquad W_1(X\ co-ord.\ coeff.) = gradient \qquad W_2(Y\ co-ord.\ coeff.) = -1$$

The sign of Y is negative after isolating all terms to one side of the equation, and the default coefficient of Y has a constant magnitude of 1 so we only need to calculate the gradient and y-intercept.

So firstly, I created a function *task2_calcGrads(x)* which calculates the constants for all boundary functions of a given polygon x. It takes an input of a polygon coordinate matrix and produces an output of a boundary function matrix (where each row represents a boundary function and each column represents the associated gradient and y-intercept respectively). Implementing this function promotes much higher accuracy for boundary calculations than mere hardcoding.

Now, given we have the respective boundary function constants we must identify which boundaries are the maxima and minima of this polygon. This is very relevant as it denotes which side of the boundaries represents

the polygon. I identified these maxima and minima by capitalizing on the order of polygon vertex input, in which the maximum Y vertex is always first and the other vertices follow in a clockwise/anti-clockwise fashion. In order to represent these maxima and minima boundaries we can just multiply the minima boundary functions by -1.

Given that we only account for points inside the polygon (not including the periphery) I deducted a tiny value ($-1*10^{-14}$) from the maxima boundaries and added a tiny weight ($+1*10^{-14}$) to the minima boundaries.

At this point, the only thing left to do is normalize the weights. So, I divided each weight vector by its associated maximum magnitude element.

WEIGHTS: Hidden 1 -> Output

Given that the first layer of neurons each output a 1 or a 0 to represent if the given point is within the given boundary, the rest of the neurons act like AND logic gates. This is because the point must be within <u>all</u> borders to be within the polygon.

So, in order to create a suitable weighting, I created a large negative bias which could only be overcome if both W_1 and W_2 inputs were 1.

$$W_0 = -1 \qquad W_1 = 0.51 \qquad W_2 = 0.51$$

2.10) Difference in decision boundary calculations for *task2_hNN_AB()* & *task2_sNN_AB()*

The only difference between the layout of these neural networks was the activation functions used. The first using a step function (hNeuron) and the second a sigmoid function (sNeuron).

The predominant difference I found in calculating these decision boundaries was the weights I used. This is because in contrast to the step function which produces an output of 0 or 1 the output of a sigmoid is in the range from 0 to 1 which makes it far more difficult to simulate logical gates without any threshold to classify the output of a given neuron. In order to cater for this, I decided it would be useful to use non-normalized weights for the sigmoid function, I did this by multiplying each weight vector by 10^8, this in effect:

creates a very small number

$$if \quad a > 0 \quad then \quad e^{-(a)} \to 0 \quad \therefore g(a) = \frac{1}{1+0} \cong 1$$

and a very large number

$$if \quad a \leq 0 \quad then \quad e^{-(-a)} \to \infty \quad \therefore g(a) = \frac{1}{1+\infty} \cong 0$$

This method worked perfectly in order to simulate logical gates using the sigmoid function, and no threshold was even required to classify the final output(s) of the sigmoidal neural network upon estimation of the decision boundaries.

As to be expected, the visualisations of decision boundaries produced by *task2_plot_regions_hNN_AB.mat* and *task2_plot_regions_sNN_AB.mat* were identical.