# SCC 403 Project

36668259
School of Computing & Communication
SCC 403 : Data Mining

***Abstract*—Within the report, I performed several techniques to pre-process, cluster and classify data. Throughout my investigation I found that normalising often worked much better to perform clusters on within both tasks. Furthermore, after using Spectral Clustering, Gaussian Mixture Model (GMM) and K-means to cluster. Although the methods produced similar results, K-means and GMM were the most consistent of the methods. On top of this, to reduce dimensions of the dataset for visualisations UMAP often performed better than PCA and the silhouette scores obtained are proof of this.**

## I. Introduction

This project focuses on two distinct yet methodologically connected tasks: Analysis of a Climate Dataset and Image Processing using Pet images. Both tasks utilise clustering techniques to uncover patterns and insights in the data.

The first section focuses on the climate data, from Basel, understanding and hypothesising the meanings of the clusters and the explanations for why certain methods produced better results than others.

The second section explores image processing with the 2 different pets datasets which both include cats and dogs. The goal of this task is to cluster the images and to classify them using Deep Neural Networks. The classifying will be done in terms of breeds and whether the pet is a cat or dog. This allowed me to understand and use different models to obtain results and handling different styles of data.

## II. Climate Dataset

The climate dataset was from Basel, Switzerland and it only included from the summer and winter seasons from 2010 to 2019. Pre-processing is very important in a Machine Learning pipeline, this is because it can handle missing data, remove columns that are irrelevant to focus on the most important information and also manage outliers using IQR (inter-quartile range) so they do not distort the results and lead to misleading conclusions.

### A. Pre-Processing

The steps included in pre-processing were:

- Normalise or Standardise the data
  - This ensures all columns contribute equally to clustering which reduces bias caused by different units within the dataset.
  - I chose to Normalise the data because it usually obtained better results when completing feature extraction.
- Removing Outliers
  - Reducing outliers minimises bias in distance-based algorithms like PCA and UMAP. However, outliers can sometimes hold critical information, and their removal risks skewing the analysis. In this project, removing outliers improved clustering results.
- Feature Selection and Extraction
  - Within the dataset there are 18 dimensions (this means 18 columns). Because it is climate data and it includes minimums and maximums. These values represent extreme ends of the Dataset which means they don't tend to capture the overall trend of the data. Furthermore, these columns are susceptible to noise. This is because extreme values could misrepresent data which is why the mean holds much less volatility especially when standardised and normalised.
  - The data initially had 18 columns and some observational analysis such as how the plot looked without removing features and how it looked after removing features.
  - When using PCA and UMAP I dropped these columns so data points wouldn't overlap in clusters and it would improve clustering.
- Missing Data
  - Within the dataset there was no record of missing data. However, if missing data was present techniques such as using the mean can be used or even using the median of the dataset can be used. These are the simplest way to fill missing data and these will both preserve the central tendency of the dataset which will make the dataset consistent.

### B. Methods of Dimensionality Reduction

Dimensionality reduction is important because it allows the user to simplify complex datasets, reduce computational costs, improve model performance, and enhance data visualisation while retaining the most relevant information.

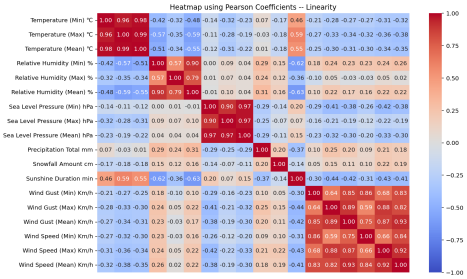


**Fig. 1:** Pearson's Correlation to identify linear relationships.

Initially, I opted for using PCA, UMAP and $t$-SNE. However, when applying all three techniques $t$-SNE often yielded results similar to PCA. So, instead I used a technique that could reduce dimensionality with the entire 18-dimensions.

Before hypothesising, I calculated Pearson Coefficients and visualised them in a heat map [1]. This showed strong linearity (close to 1, in red) among columns like Temperature min, max, and mean, but weak linearity between Temperature and other columns (this was a trend along the other columns too). Additionally, some columns showed significant negative correlations, hinting at possible non-linear relationships. Based on this, I chose Isomap to further explore the data because it can handle non-linearity well when doing dimensionality reduction.

*1) **PCA**:* PCA (Principal Component Analysis) is a linear dimensionality reduction technique that transforms data by identifying directions (principal components) with the highest variance [2]. It is commonly used to reduce the number of features while retaining the most important information, simplifying complex datasets for visualisation or modelling. PCA is particularly effective for eliminating redundancy and uncovering patterns in linearly correlated data.

*2) **Isomap**:* Isomap is an unsupervised machine learning algorithm for non-linear dimensionality reduction, mapping high-dimensional data into a lower-dimensional space while preserving local and global structures. Isomap is an extension of MDS (multi-dimensional scaling) but it uses a different distance metric [3].

Unlike MDS, which reduces dimensions by preserving relative distances using a Euclidean distance matrix, Isomap uses

geodesic distances [4]. Geodesic distances represent the shortest path along the data's manifold, making Isomap suitable for non-linear structures [5]. These distances are computed using graph-based algorithms like Dijkstra's, treating data points as nodes and distances as edge weights, resulting in a matrix (e.g., size 18 for our dataset) that preserves the true data structure. Here is more information on MDS [6].

Isomap works as follows, briefly: First, a graph is constructed from the data points, where the distances between each pair of connected points (edges) are calculated and assigned as weights. Next, the shortest geodesic distances between all data points in the graph are computed using Dijkstra's algorithm [7]. These geodesic distances approximate the shortest paths along the underlying manifold.

Subsequently, Multidimensional Scaling (MDS) is applied to the geodesic distances by minimising the cost function:

$$\sum_{i<j} \left(d_G(i,j) - \|\mathbf{y}_i - \mathbf{y}_j\|\right)^2,$$

$d_G(i,j)$ represents the geodesic distance in the high-dimensional space, and $\|\mathbf{y}_i - \mathbf{y}_j\|$ is the Euclidean distance between the corresponding points in the lower-dimensional embedding [8].

To calculate the embeddings, a method called the "Gramian" matrix is employed (see more in [9]). This involves calculating the eigenvalues and eigenvectors of the Gram matrix, which are then used to generate the lower-dimensional representation.

Isomap's ability to combine geodesic distances (capturing the global manifold (patterns) structure) and Euclidean distances (preserving local relationships) allows Isomap to effectively uncover the shape of the data. A detailed theoretical proof of Isomap's methodology is available in [10].

By reducing dimensions to 2D, Isomap separates clusters and highlights meaningful relationships while reducing noise. Unlike PCA, which maximises variance using principal components [11], and UMAP, which emphasises local relationships, Isomap captures the data's global structure. Its geodesic distance calculations also handle outliers well, making it effective for datasets with complex structures and better suited for full datasets than PCA and UMAP.

*3) **UMAP**:* UMAP (Uniform Manifold Approximation and Projection) is an unsupervised machine learning algorithm for non-linear dimensionality reduction, designed to preserve both local and global data structures while creating efficient and meaningful low-dimensional representations of high-dimensional datasets. It constructs a "fuzzy simplicial complex," [12] a geometric representation of relationships between data points, to approximate high-dimensional structures and project them into a lower-dimensional space. UMAP preserves global structures better than a lot of methods (such as PCA, $t$-SNE), making its cluster relations more meaningful, though occasional "snake-like" shapes occur due to its method of flattening curved relationships.

While both UMAP and Isomap are non-linear techniques, Isomap emphasises preserving global geometry through geodesic distances, making it effective for smooth manifolds. UMAP, however, balances local and global relationships, is computationally faster, and excels at clustering and visualising [13]. Unlike PCA, which focuses on linear variance for better interpretability, UMAP handles complex, non-linear structures and often produces clearer and more meaningful cluster separations, making it a preferred choice for exploring high-dimensional data.

### C. Methods of Clustering

The 2 clustering methods I opted for were Gaussian Mixture Method and Spectral Clustering.

*1) **Spectral Clustering**:* This is a method that uses Eigenvalues and Eigenvectors of a similarity matrix to partition data points into clusters based on their connectivity and relationships [14]. This comes in a few steps.

- First we form a distance matrix. This means calculating the distances between all points pairwise such as Euclidean distance or Cosine Distance. [15]
- This matrix is transformed into a similarity matrix.
  - Graph construction methods vary: the $\epsilon$-neighbourhood graph connects points within a fixed distance, the k-nearest neighbour graph links each point to its k-closest neighbours, and the fully connected graph connects all points with weights based on pairwise similarities. Sci-kit Learn uses the fully connected method with a Gaussian Kernel to ensure connected clusters.
- We calculate the Laplacian matrix
  - There are three ways to calculate this matrix; the un-normalised method (unreliable), the normalised method (symmetrical), the normalised (random walk). The un-normalised method is $L = D - S$. Where $D$ is the degree of the matrix or the matrix diagonals and $S$ is the similarity matrix. The normalised method (symmetrical) ($L_{sym} = I - D^{-\frac{1}{2}}SD^{-\frac{1}{2}}$). However, there is another normalised method ("random walk") ($L_{rw} = I - D^{-1}s$) Luxburg[15] explains both these normalising methods of the Laplacian are good however, it is less computationally expensive to use the random walk normalised method. This made the paper side heavily toward the random walk method as it yielded better results and it didn't "yield undesired artifacts" [15].
- Find the Eigenvalues and Eigenvectors of the Laplacian Matrix
  - And then form a matrix with the largest eigenvectors of the k-largest eigenvalues. "Largest" eigenvalues actually refers to the smallest eigenvalues. This is because the eigenvalues of Laplacian matrices are always non-negative and ordered increasingly (0 = $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$). The eigenvectors corresponding to the k smallest eigenvalues are the ones of interest for spectral clustering [15].
- Normalise the the vectors
- Finally, cluster the data points into k-dimensions

I clustered the data after normalising and dropping columns for PCA and UMAP, while for Isomap, I normalised and checked for outliers on the full dataset. UMAP performed best, separating the data into 2 clusters with a silhouette score of 0.731, indicating well-separated clusters. PCA and Isomap scored lower at 0.417 and 0.511 respectively.

The reason I chose 2 clusters was that they consistently performed better based on metrics like the Davies–Bouldin index (DBI) and silhouette score. While I initially hypothesized that 3 clusters might capture a transitional season between summer and winter, the metrics showed otherwise, with a lower silhouette score (0.417) and higher DBI (0.906). This indicates that the transitional period blends into the two primary clusters rather than forming its own. This pattern was consistent across the clustering methods in the report, including GMM, which yielded similar results.

Another clustering metric that was used was the Davies–Bouldin index (top to bottom: 0.794, 0.694, 0.401). This measures the quality of clustering by evaluating the compactness of each cluster (how close the points in a cluster are to its centre) and the separation between clusters (how far apart the clusters are).

However, standardising the data caused Spectral Clustering to perform the worst with a score of 0.033. This is because Spectral Clustering uses a fully connected graph, where every point influences one another, and standardisation created weak connections, amplifying noise and reducing performance. This caused stray clusters of points away from the main plot to cluster. This is often a limitation with Spectral clustering as this is because if clusters within the Dataset are uneven the first few eigenvectors are used to find the clusters and this causes incorrect results.

*2) **Gaussian Mixture Model**:* While Spectral Clustering is a graph-based model which constructs a similarity measure (the Gaussian kernel), GMMs are probabilistic models used for clustering and density estimation [16]. It is assumed by this

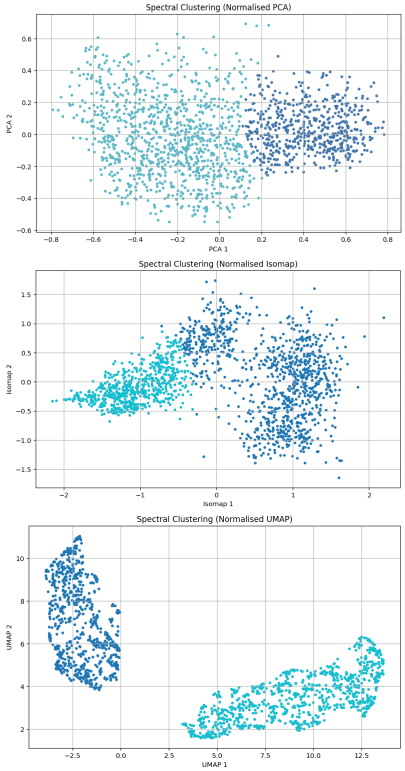


**Fig. 2:** Top: PCA, Middle: Isomap, Bottom: UMAP

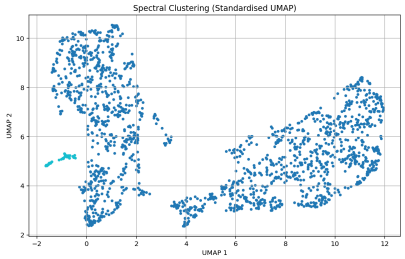


**Fig. 3:** Standardised UMAP (see small blue cluster)

model that the data generated in the form of several Gaussian components, where each component in a cluster.

GMMs overcome a key limitation of k-means, which struggles with non-circular clusters, by using a distribution-based approach. Unlike k-means, GMMs assign probabilities to clusters using Bayes' Theorem [16]. For example, if a data point has an 0.8 chance of being in the blue cluster and 0.2 in the green cluster, it is assigned to the blue cluster.

The foundation of Gaussian Mixture Models (GMMs) lies in the Gaussian distribution and the use of Maximum Likelihood Estimation (MLE) to predict the likelihood of a data point belonging to a cluster. While MLE can theoretically be used to detect outliers in a dataset, in the case of GMMs, it is applied to iteratively update the probability of data points being assigned to clusters based on parameters like the mean, covariance, and density [17].

Whereas K-means relies solely on the mean and Spectral Clustering relies on: a similarity matrix, Laplacian Matrix's, Eigenvalues and Eigenvectors [17]. So, it isn't as complex a method as Spectral clusters, but a more complex method than K-means. Which also means K-means is the cheapest computationally then GMM, then Spectral Clustering due to matrix calculations.

We can observe that both Spectral Clustering and GMM

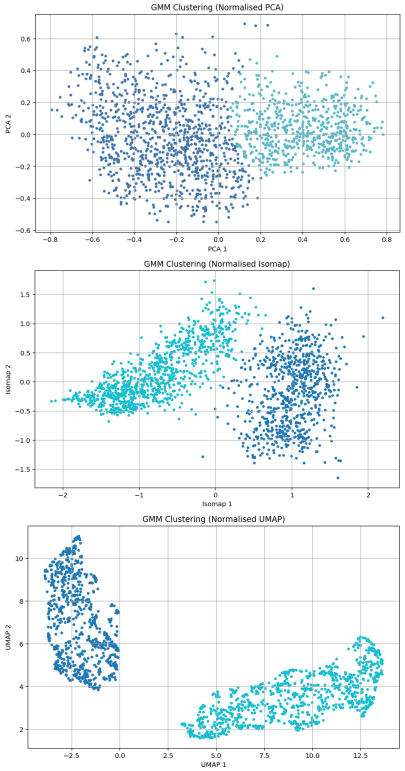


**Fig. 4:** Top: PCA, Middle: Isomap, Bottom: UMAP

performed very similarly. The silhouette scores ( top to bottom: 0.428, 0.545, 0.731) and the Davies–Bouldin indexes (top to bottom: 0.793, 0.705, 0.401) shows that the clusters for UMAP were much better compared to the ones from Isomap and PCA.

The reason GMM worked better was because of Spectral clusterings reliance on the sensitivity of the data and the global properties of the data [17]. This means that the similarity matrix formed is extremely sensitive to noise and this is why it couldn't effectively cluster compared to GMM which clustered well due to it being reliant on probabilities.

### *D. Conclusions*

Overall, GMMs provided better clustering visualisations than Spectral Clustering due to their probabilistic approach and ability to model diverse cluster shapes and sizes. In contrast, Spectral Clustering, while effective with graph-based methods, is more sensitive to noise and similarity measures (due to Eigenvalues and Eigenvectors) , which can result in less distinct boundaries compared to GMMs.

Furthermore, normalising the dataset worked better for visualisations and for clustering. Furthermore, the non-linear dimensionality reduction methods produced much better results showing their accuracy and applicability when preserving relationships and structure.

## III. Image Classification

The image classification section of this report involves two different Datasets. Both of which are pet Datasets. Namely:

- Oxford Pets : 7,000+ images containing 37 breeds of cats and dogs
- Kaggle Pets : 25,000+ images containing only cats and dogs separated by cats and dogs

In this section, I used two deep neural networks, ResNet18 and VGG16, to compare their accuracies. Additionally, I applied a custom linear layer to train and classify the images, aiming to further evaluate the accuracies. As part of the analysis, I also inspected the clusters formed by these models

and identified which approach resulted in the best clustering outcomes.

### A. Pre-Processing

For both datasets, I normalised and resized the images to ResNet-18's required input size of 224x224. Instead of calculating normalisation tensors manually, I opted to use Image-net's values [18].

This approach improves the pipeline's efficiency, as calculating these values independently would be time-consuming and likely yield results similar to Image-net's values. Given that Image-net's normalisation is based on over 14 million images, it provides a robust and reliable standard for preprocessing.

*1) **OxfordPetsIII**:* For data preprocessing, I exclusively utilised "Tensor Flow Transform" to handle tasks such as feature scaling, normalisation, and other transformations necessary to prepare the data for training.

*2) **Kaggle Pets**:* The same was done for this dataset where I used "Tensor Flow Transform" to handle feature scaling. normalisation and other transformations when necessary.

### B. Deep Neural Networks

A Deep Neural Network is a composition of multiple layers between its inputs and output layers. Each layer within the DNN applies a linear transformation which is then followed by the activation function.

This actually introduces non-linearity into a model so layers within the model can understand and learn complex patterns and structures within a Dataset. There are many activation functions such as ReLU and $tanh$ [19]. This then allows Neural Networks to solve complicated tasks such as Image Classification and Natural Language Processing.

The two DNNs used in this project were ResNet18 and VGG16. While ResNet50 is a powerful model for image classification, it is computationally more expensive than ResNet18, making ResNet18 a more practical choice for these tasks. Comparisons were still made due to time constraints and the computational demands of the models.

Before introducing the DNN's over-fitting can be an incremental problem for these networks. This is when a DNN learns to memorise the training data, including noise and irrelevant patterns, instead of generalising to unseen data.

*1) **RESNET18**:* ResNet18 is a DNN which belongs to a family of Residual Networks. They were introduced as a family in the paper "Deep Residual Learning for Image Recognition" by Kaiming He et al. This paper introduced many Neural Networks such as ResNet18, ResNet34 (A, B or C), ResNet50 etc [20].

In 2015 this was a massive deal because ResNet18 actually solved a massive problem that had been going on with DNN's. That was the Vanishing Gradient. To understand this a brief explanation of the mathematics behind DNN's will be needed.

- Linear Transformation
  - Firstly a linear transform is calculated using :

$$z = W \cdot x + b$$

    where; $z$ is the output, $b$ is bias, $W$ is a weight matrix (calculated Gradient Descent), $x$ is an input vector [21]. This is calculated within each unit within each layer of the DNN.
- Activation Function, Forward-propagation
  - As stated before, this is where non-linearity is introduced so that a network can solve complex problems.
  - To simplify things the activation function ,say $f(x)$, is applied to every layer and every node with $z$ as its input. This means $f(z)$ is calculated across $n$-layers and through all nodes [21].
  - Then, through many iterations and many calculations the loss function is minimised. This is because DNN reduces loss by adjusting its parameters again and again until the most optimised path and nodes have been calculated which effectively reduces the loss function [21].
- Backwards-propagation and Optimisation
  - This is when the gradients of the loss function are calculated using the weight function and some partial derivatives using the chain rule. Each time it is calculated for a layer it is updated and used during training.
  - In very deep DNNs, gradients shrink during backward-propagation due to repeated multiplication of small values (due to partial derivatives), causing them to approach zero. As a result, earlier layers receive little to no gradient information, making it difficult to update weights and train the network effectively [21].
  - Gradients are calculated from the output layer (last layer) to the input layer (first layer) - This is the Vanishing Gradient Problem.
  - Then, Gradient Descent or other algorithms are used to update parameters. In this project Adam was used to optimise the fully connected layer I used. ResNet also uses the Adam optimiser [20].

ResNet18 addresses the vanishing gradient problem with "Skip Connections" [20], which create shortcuts allowing inputs to bypass intermediate layers and directly influence the output. These connections forward the input, ensuring compatibility through transformations like resizing and normalising. Meanwhile, intermediate layers are calculated, enabling gradients to flow smoothly during backwards-propagation. This prevents gradients from vanishing, even in very deep networks, as the input alignment through skip connections preserves gradient values.

In ResNet18, convolutional layers extract features such as patterns, textures, and colour values using small filters or "kernels" [22]. The core innovation is the residual block, where skip connections focus the network on learning residual features—adjustments between the input and desired output. This simplifies learning by reducing unnecessary transformations and emphasising feature refinement. The result is more efficient optimisation, faster training, and better generalisation for complex tasks [20].

By reducing computational complexity and focusing on improvements over memorization, skip connections enhance gradient flow and reduce over-fitting. ResNet18's efficiency is driven by filters and a direct pipeline, as highlighted by Aditya Thakur (and others) [23], demonstrates its optimisation for real-time applications.

*2) **VGG16**:* VGG16 is a Convolutional Neural Network (CNN), a type of Deep Neural Network (DNN) specialised for images and videos, commonly used for tasks like image classification. Introduced in 2014 by the Visual Geometry Group at Oxford, VGG16 consists of 16 layers, 13 of which are convolutional [24]. These layers are organised into 5 groups, each containing two or three convolutional layers followed by pooling layers that reduce dimensions from 224x224 to 14x14.

Feature extraction occurs progressively in the network: early layers detect edges and textures, while deeper layers capture more complex features like object details. The hierarchical stacking of convolutional layers enables VGG16 to learn and represent features at multiple levels, resulting in high classification accuracy.

Max-pooling filters in the network reduce dimensions while retaining important information, mitigating over-fitting by filtering out noise and emphasising key features. This enhances the network's robustness to small input variations and preserves critical information across layers [24].

The output layer uses the softmax function to convert logits into probabilities for each class [25]:

$$y(z) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

These softmax values represent the likelihood of the input belonging to each class making it easy to interpret for decision

making [25].

Here, $z_i$ and $z_j$ are raw logit scores, and the resulting probabilities$y(z)$ indicate the likelihood of the input belonging to each class $K$, facilitating decision-making.

| Characteristic | ResNet18 | VGG16 |
|---|---|---|
| Parameters | 11.7 mil. | 138 mil. |
| Convolution (blocks) | 4 | 5 |
| Activation (Func) | ReLU | ReLU |
| Res. connections | Yes, shortcuts. | None |
| Fully Connected | 1 | 3 |
| FC | 1(37,2) | 3(4096, 4096, 37,2) |
| FC-Processing | 1 x 1 x 512 | 7 x 7 x 512 |
| Comp. Cost | Low | High |

**TABLE I:** Some comparisons of ResNet18 and VGG16. 37 or 2 in "FC" because OxfordPets has 37 classes or 2 and Kaggle Pets has 2 classes.

*3) Comparisons:* Another comparison that can be made is that ResNet18 has only 1 fully connected layer (1000 outputs) whereas VGG16 has 3 fully connected layers with (4096, 4096 and 1000) outputs [24]. The more FC layers the better the performance compared to without them this is because each FC layer learns a set of weights that represent the features of a layer before. More comparisons can be found here [26].

However, these layers are **very** computationally expensive as the parameter size equation is : Parameter size = (Input Size + 1) · Output Size.

So for VGG16 if the layer has 4096 outputs (refer to I ) then the first layer will have (25,088 + 1) · 4096 ≈ 103 million. This makes this very computationally demanding compared to the one fully connected layer for ResNet18 which is 513,000 parameters. This is 200 times more calculations for VGG16.

Now, using the two DNN's above I will discuss my observations from both the OxfordPetsIII dataset and the Kaggle Pets dataset.

### C. OxfordPetsIII

This dataset could be classified in two ways. One could be as 37 breeds or the other as "Cat" or "Dog". I experimented with both and here are the results for only 37 breeds.

*1) Clusters:* Using UMAP on the original features these clusters were observed.

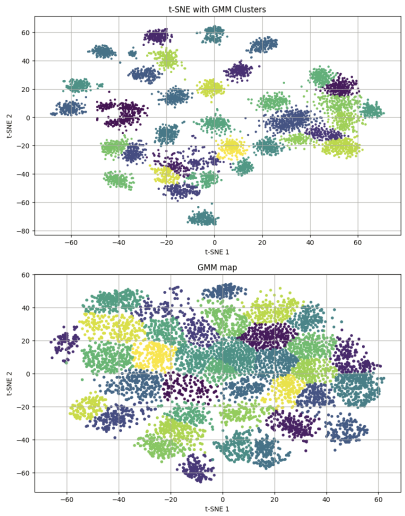


**Fig. 5:** Top : ResNet18. Bottom: VGG16.

ResNet18 had a silhouette score of 0.45 and a purity score of 0.820 whereas VGG16 had a silhouette score of 0.18 and a purity score of 0.508.

This means ResNet18 creates better feature representations for clustering compared to VGG16. This is because higher silhouette and purity scores indicate that ResNet18 extracts features that are better for clustering tasks, creating more distinct and meaningful groupings of the data. Furthermore, the purity score suggests ResNet18 clusters are well-aligned with the true labels however VGG16 has moderate alignment suggesting less effective feature separation.

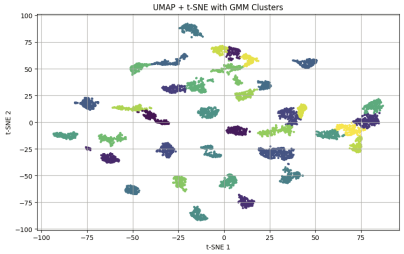


**Fig. 6:** Using UMAP + t-SNE on ResNet18 Features

I then experimented with using UMAP and then t-SNE to visualise the clusters and obtained way better results [27]. This article (referenced) was the inspiration for this and I obtained a sil. score of 0.64 with a purity of 0.802. This shows combining UMAP with t-SNE. This shows combining UMAP with t-SNE can show the strengths of both methods, where UMAP preserves global and local structures, and t-SNE refines the representation by emphasising details in the local relationships.

*2) Training and Classification:* Initially, I used the features from the model to see how the model was doing when classifying images and then I used my own fully connected layer using "TensorFlow" and I trained this layer using the features from ResNet18 and VGG16.

This technique is called Transfer Learning" [28]. Weiss Karl and DingDing notes in Transfer Learning Applications" that 'Ouqab' used a CNN internal layer for better classification, consistently achieving improved results. This works because fine-tuning reduces training time and enables the layer to learn general features.

So, This is what I opted to do with the DNN's I had chosen. Firstly, I would train VGG16 and ResNet18 on the original photos and then after I would save and extract these features and train them to see if I was getting better results.

These are confusion matrices showing true positives along the leading diagonal, with percentages indicating class accuracy.

I used SVM (Support Vector Machine) to classify the images as it is a non-linear supervised ML technique effective for non-linearly separable data [28]. However, SVM is computationally expensive, especially when combined with VGG16.

ResNet18 took 0.8 seconds (GPU) and 1 minute (CPU) to train and classify images with an average accuracy of 90%. After training a custom fully connected layer, I achieved 97% accuracy, 7% better than the base ResNet18.

VGG16 took over 6 minutes (GPU) and 17 minutes (CPU) to train due to its large parameter size. Initially, VGG16 had 90% accuracy, but after training my custom layer, it reached 99%. While VGG16 outperformed ResNet18, it was significantly slower, demonstrating ResNet18's efficiency.

Testing my custom dataset showed VGG16 achieving near-perfect accuracy on most images (low of 98%), whereas ResNet18 was less consistent (low of 78%). Using pre-trained features, VGG16 took 32 minutes to train (37 classes) and 1 second to classify, while ResNet18 took up to a maximum of 4 seconds for both.

Both models struggled to distinguish between a cat and a dog when the cat had a leash, as leashes are more commonly associated with dogs. These were the only notable misclassification.

In summary, ResNet18 is more efficient but less accurate than VGG16. VGG16, while computationally expensive, consistently produced better results, achieving near-perfect accu-

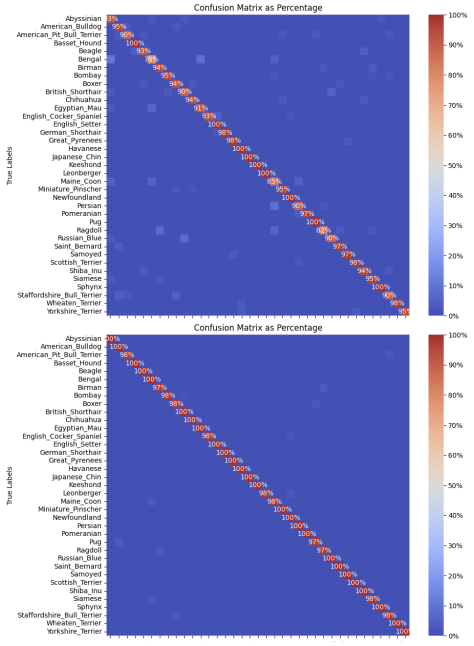


**Fig. 7:** Confusion Matrix showing true classifications. Top : ResNet18 After Training and Classification. Bottom: VGG16 After Training and Classification.

racy (lowest: 98%) for most breeds, compared to ResNet18's variability (sometimes as low as 78%).

### *D. Kaggle Pets*

This dataset had only 2 labels and they were "Cat" or "Dog".

*1)* ***Clusters****:* Here are the clusters per DNN:

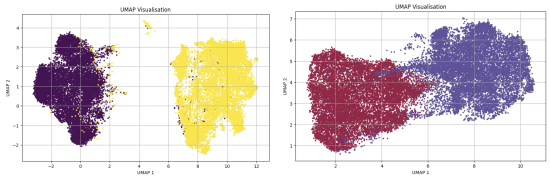


**Fig. 8:** Left : ResNet18. Right: VGG16.

As shown in Fig. 9, ResNet18 achieved better clustering with a higher silhouette score (0.73) and purity (0.99), indicating well-aligned features with the true labels. VGG16 had a silhouette score of 0.59 and purity of 0.97, reflecting decent clustering but less alignment with true labels (by 2%). Additionally, ResNet18 was faster (20 seconds) compared to VGG16 (1 minute), highlighting VGG16's much higher computational cost.

*2)* ***Training and Classification (SVM)****:* The same that was done with OxfordPetsIII was done here.

Again, VGG16 was much more computationally intensive compared to ResNet18. However, the accuracy cannot be faltered because it did perform better than ResNet18 after training and before training. Having an almost 100% accuracy whereas ResNet18's accuracy was 99%

*3)* ***Evaluations****:* This further shows the efficiency and effectiveness of using ResNet18 even though it lost out in terms of percentage accuracy it completed the task in 5 seconds compared to VGG16's 9 mins. If I used ResNet34, or ResNet50 with more layers I would get better results and better accuracy all whilst increasing the computational side. This shows that even with the 18 layers that ResNet has it does a brilliant job of classifying (almost) correctly.

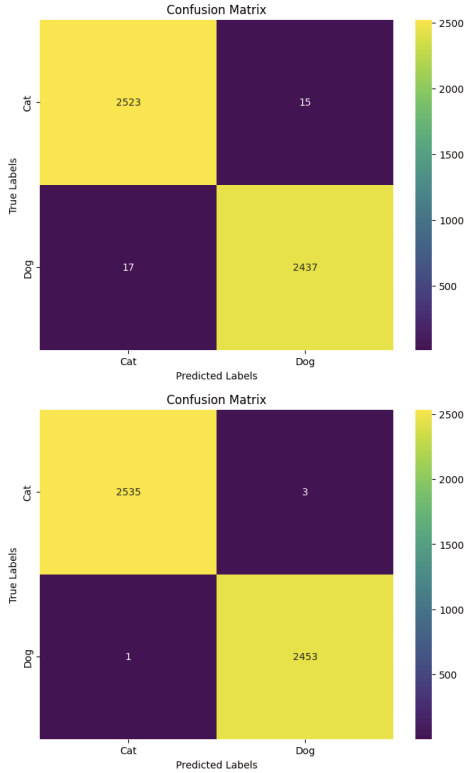


**Fig. 9:** Top : ResNet18. Bottom: VGG16.

| Training Phase | ResNet18 | VGG16 |
|---|---|---|
| Before (%) | 98.2%. | 98.6% |
| After (%) | 99.2% | 99.8% |
| Before (mins) | 5s | 9m 34s |
| After (mins) | 8s | 13s |
| Classif. (type) | SVM | SVM |

**TABLE II:** Evaluation between VGG16 and ResNet18 Kaggle

### *E. Conclusions*

In conclusion, ResNet18 and VGG16 both have advantages depending on the task and available resources. Running both models on and off CUDA allowed me to evaluate their strengths in different areas.

ResNet18 was more computationally efficient than VGG16, with faster training and classification, making it ideal for real-time scenarios with limited resources. Its skip connections gave it an advantage in feature extraction by turning inputs into outputs.

VGG16, however, consistently delivered higher accuracy, particularly in multi-class breed classification, OxfordPetsIII 2-class verification (see Appendix), and Kaggle Pets. While computationally expensive, it excelled in precision.

ResNet18 is practical for quick and accurate predictions, whereas VGG16 suits tasks prioritising precision over speed. Both models benefited from transfer learning, and deeper ResNet variants like ResNet34 or ResNet50 could potentially balance accuracy and efficiency. The choice ultimately depends on the trade-off between accuracy and computational cost.

To carry on this research for the future, using a DNN such as Visual Transformers (ViT's) because it is a balance of accuracy and computational efficiency. Furthermore, using another classifications technique in ensemble with SVM such as gradient boosting could yield even better results.

## References


[1] Newcastle University. *Strength of Correlation*. Accessed: 8/12/2024. URL: https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/statistics/regression-and-correlation/strength-of-correlation.html.

[2] Builtin. *Step-by-Step Explanation: Principal Component Analysis*. Accessed: 8/12/2024. 2024. URL: https://builtin.com/data-science/step-step-explanation-principal-component-analysis.

[3] Tanzeel U. Rehman & Li Jing Mahwish Yousaf. "An Extended Isomap Approach for Nonlinear Dimension Reduction". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* (2020). URL: https://link.springer.com/article/10.1007/s42979-020-00179-y.

[4] Yan-Bin Jia. *Geodesics on Manifolds*. Accessed: 8/12/2024. 2022. URL: https://faculty.sites.iastate.edu/jia/files/inline-files/geodesics.pdf.

[5] Eric W. Weisstein. *Geodesic*. Accessed: 8/12/2024. URL: https://mathworld.wolfram.com/Geodesic.html.

[6] Stringham Jessica. *Multidimensional Scaling*. Accessed: 2024-12-12. 2018. URL: https://jessicastringham.net/2018/05/20/Multidimensional-Scaling/.

[7] W3Schools. *Dijkstra Algorithm for Graphs*. Accessed: 8/12/2024. URL: https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php.

[8] Guangliang Chen. *Lecture 10: Isomap*. Accessed: 8/12/2024. n.d. URL: https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec10ISOmap.pdf.

[9] John D. Cook. *Gram Matrix*. Accessed: 2024-12-12. 2023. URL: https://www.johndcook.com/blog/2023/07/09/gram-matrix/.

[10] Y. Yao. *Lecture 08: Introduction to Dimensionality Reduction*. Accessed: 2024-12-12. 2011. URL: https://www.math.pku.edu.cn/teachers/yaoy/Spring2011/lecture08.pdf.

[11] Shalizi Cosma Rohilla. *Lecture 18: Dimensionality Reduction*. Accessed: 2024-12-12. 2012. URL: https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch18.pdf.

[12] Wikipedia Contributors. *Simplicial Complex*. Accessed: 8/12/2024. 2024. URL: https://en.wikipedia.org/wiki/Simplicial_complex.

[13] Pair Code. *Understanding UMAP*. Accessed: 8/12/2024. n.d. URL: https://pair-code.github.io/understanding-umap/.

[14] Analytics Vidhya. *What, Why, and How of Spectral Clustering*. Accessed: 8/12/2024. 2021. URL: https://www.analyticsvidhya.com/blog/2021/05/what-why-and-how-of-spectral-clustering/#h-pros-and-cons-of-spectral-clustering.

[15] Ulrike von Luxburg. *A Tutorial on Spectral Clustering*. Accessed: 8/12/2024. 2007. URL: https://people.csail.mit.edu/dsontag/courses/ml14/notes/Luxburg07_tutorial_spectral_clustering.pdf.

[16] Dong Yu and Li Deng. "Gaussian Mixture Models". In: *Automatic Speech Recognition: A Deep Learning Approach*. Accessed: 2024-12-15. Springer, 2011. URL: https://link.springer.com/book/10.1007/978-1-4471-5779-3.

[17] Anderson Y. Zhang Matthias L¨offler and Harrison H. Zhou. "OPTIMALITY OF SPECTRAL CLUSTERING IN THE GAUSSIAN MIXTURE MODEL". In: *arXiv preprint arXiv:1911.00538* (2019). Accessed: 2024-12-15. URL: https://arxiv.org/pdf/1911.00538.

[18] PyTorch Team. *ImageNet training in PyTorch*. Accessed: 2024-12-14, Check line 236-237. URL: https://github.com/pytorch/examples/blob/main/imagenet/main.py.

[19] Encord Blog Team. *Activation Functions in Neural Networks: A Complete Guide*. Accessed: 2024-12-14. 2023. URL: https://encord.com/blog/activation-functions-neural-networks/.

[20] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[21] Towards Data Science. *Introduction to Math Behind Neural Networks*. Accessed: 2024-12-15. 2024. URL: https://towardsdatascience.com/introduction-to-math-behind-neural-networks-e8b60dbbdeba.

[22] Youngmin Cho and Lawrence K. Saul. "Kernel Methods for Deep Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Accessed: 2024-12-15. 2009. URL: https://proceedings.neurips.cc/paper_files/paper/2009/file/5751ec3e9a4feab575962e78e006250d-Paper.pdf.

[23] Nikunj Gupta Aditya Thakur Harish Chauhan. "Efficient ResNets: Residual Network Design". In: *arXiv preprint arXiv:2306.12100* (2023). Accessed: 2024-12-15. URL: https://arxiv.org/pdf/2306.12100.

[24] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations (ICLR)*. 2015. URL: https://arxiv.org/abs/1409.1556.

[25] SingleStore. *A Guide to Softmax Activation Function*. Accessed: 2024-12-15. 2024. URL: https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/.

[26] Author(s). *Comparison of VGG-16, VGG-19, and ResNet-101 CNN Models*. Accessed: 2024-12-15. Year. URL: https://www.researchgate.net/publication/369224507_Comparison_of_VGG-16_VGG-19_and_ResNet-101_CNN_Models_for_the_purpose_of_Suspicious_Activity_Detection.

[27] AI Competence. *Comparing t-SNE and UMAP for Dimensionality Reduction and Visualization*. Accessed: 2024-12-16. 2024. URL: https://aicompetence.org/comparing-t-sne-and-umap/?utm_source=chatgpt.com.

[28] Khoshgoftaar Taghi M. Weiss Karl and Wang DingDing. *A Survey of Transfer Learning*. Accessed: 2024-12-12. 2016. URL: https://doi.org/10.1186/s40537-016-0043-6.

[29] The Pandas Development Team. *Pandas Documentation*. Accessed: 2024-12-15. 2024. URL: https://pandas.pydata.org/docs/.

[30] The TensorFlow Authors. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Accessed: 2024-12-15. 2015. URL: https://www.tensorflow.org/.

[31] The NumPy Developers. *NumPy Documentation*. Accessed: 2024-12-15. 2020. URL: https://numpy.org/doc/.

[32] The Scikit-learn Developers. *Scikit-learn: Machine Learning in Python*. Accessed: 2024-12-15. 2024. URL: https://scikit-learn.org/stable/.

[33] The Matplotlib Development Team. *Matplotlib Documentation*. Accessed: 2024-12-15. 2007. URL: https://matplotlib.org/stable/contents.html.

[34] The PyTorch Developers. *PyTorch Documentation*. Accessed: 2024-12-15. 2019. URL: https://pytorch.org/docs/stable/index.html.

[35] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. Accessed: 2024-12-15. 2018. URL: https://umap-learn.readthedocs.io/.

[36] The Seaborn Development Team. *Seaborn Documentation*. Accessed: 2024-12-15. 2024. URL: https://seaborn.pydata.org/.


## Acknowledgments


This project uses libraries such as pandas, sk-learn, matplotlib, numpy, TensorFlow, Pytorch, umap-learn and seaborn. These libraries were used in the data processing and understanding throughout task 1 and task 2.

- **pandas** to store dataframes and manipulate them [29]
- **TensorFlow** for building and training deep learning models and for transformations such as normalising and resizing.[30].

- **NumPy** for numerical computations and sorting arrays [31].
- **scikit-learn** for implementing machine learning algorithms such as PCA, t-SNE, SVM, GMM, Spectral Clustering etc.[32].
- **Matplotlib** for data visualisation [33].
- **PyTorch** for deep learning (ResNet18 and VGG16) [34].
- **umap-learn** to use UMAP for task 1 and task 2 [35]
- **Seaborn** to visualise heatmaps and confusion matrix's. [36]

# APPENDIX
# CLIMATE DATASET EXTRAS

## A. Standardising Results

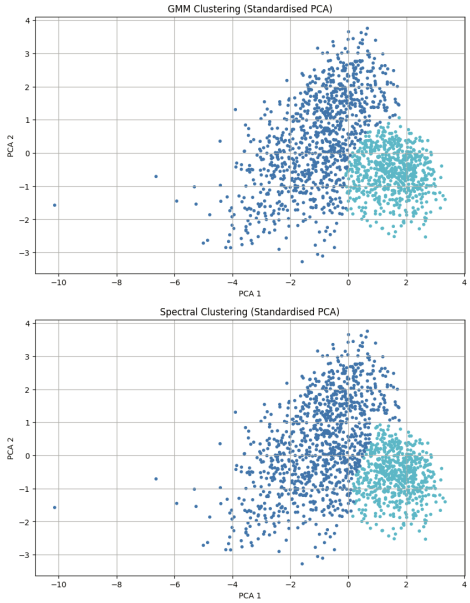


**Fig. 10:** Standardising Results were generally worse than (PCA)

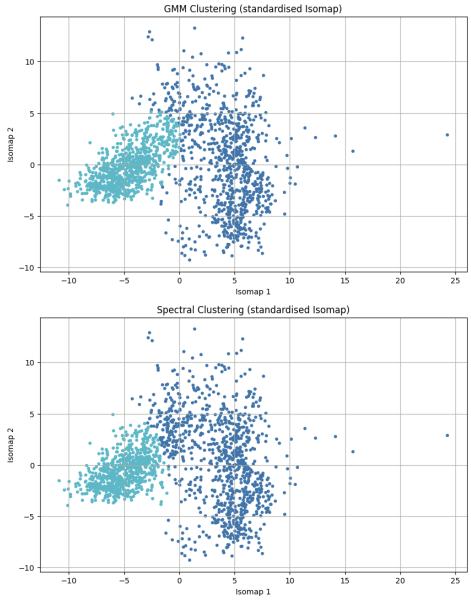


**Fig. 11:** Standardising Results were generally worse than (Isomap)

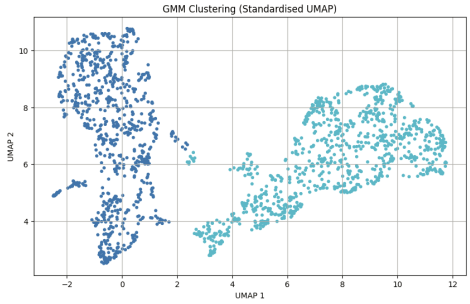


**Fig. 12:** Standardising Results were generally worse than (UMAP). GMM performed much better than spectral here referenced in Fig. 3

# CODE FOR TASK 1

Ensure that all code referenced in Acknowledgments (III-E) is properly downloaded using pip or conda.

## B. Pearson Correlation Heatmap

```
pearson_corr = CB.corr(method='pearson')

# Plot the heatmap
plt.figure(figsize=(12, 8))  # Adjust the figure size as needed
sns.heatmap(pearson_corr, annot=True, fmt=".2f", cmap='coolwarm', vmin=-1, vmax=1, xticklabels=False)
plt.title("Heatmap using Pearson Coefficients -- Linearity")

plt.savefig('Pearsons_Coefficients.pdf')

plt.show()
```

## C. Normalised Code

```
def normalise(data):
    # normalise scaler
    scaler = MinMaxScaler()

    # could use np.number but no need, so use float or integer
    numerical_data = data.select_dtypes(include=['float64', 'int64'])

    # fit the data on the numerical columns
    data_scaled = scaler.fit_transform(numerical_data)

    # normalise
    df_normalised = pd.DataFrame(data_scaled, columns=data.columns)
    return df_normalised
```

## D. Standardised Code

```
def standardise(data):
    # this is the standardise scaler
    scaler = StandardScaler()

    # could use np.number but no need, so use float or integer
    numerical_data = data.select_dtypes(include=['float64', 'int64'])

    # fit the data on the numerical columns
    data_scaled = scaler.fit_transform(numerical_data)
```

```
    # standardise
    df_standardised =
    ↪  pd.DataFrame(data_scaled,
    ↪  columns=data.columns)
    return df_standardised
```

### E. Outliers

```
def find_outliers(data):

    cleaned = data.copy()

    # select a number in the columns
    # np.number --> every type of number

    for i in data.select_dtypes(
        include=[np.number]).columns:
        Q1 = data[i].quantile(0.25)
        Q3 = data[i].quantile(0.75)
        # Interquartile Range Equation
        IQR = Q3 - Q1

        lo_outliers = Q1 - 1.5*IQR
        hi_outliers = Q3 + 1.5*IQR

        # Remove outliers from data set
        outliers = (data[i] < lo_outliers) |
        ↪  (data[i] > hi_outliers)

    cleaned = cleaned[~outliers]


    return cleaned
```

Note “include=[np.number]).columns:” may have to be on same line as ”include” to work.

### F. PCA

```
def perform_pca(data, n_components):
    numerical_data =
    ↪  data.select_dtypes(include=['float64',
    ↪  'int64'])

    # Using PCA
    pca_df = PCA(n_components=n_components)
    principalComponents =
    ↪  pca_df.fit_transform(numerical_data)

    # PCA dataframe to plot
    columns = [f'Principal Comp {i + 1}' for
    ↪  i in range(n_components)]
    dataframe_pca = pd.DataFrame(
        principalComponents,columns=columns)

    if n_components == 2:
        plt.figure(figsize=(10, 6))
        plt.scatter(dataframe_pca.iloc[:,0],
        ↪  dataframe_pca.iloc[:,1], s = 5)
        plt.xlabel("PC 1")
        plt.ylabel("PC 2")
        plt.grid(True)
        plt.show()

    else:
        raise ValueError(' 2 dimensions is
        ↪  best')

    return dataframe_pca
```

### G. Isomap

```
def perform_isomap(data, n_components=2,
↪  n_neighbors=5):
    # Using Isomap
    isomap =
    ↪  Isomap(n_components=n_components,
    ↪  n_neighbors=n_neighbors)
    isomap_result =
    ↪  isomap.fit_transform(data)

    # Plot
    plt.figure(figsize=(10, 6))
    plt.scatter(isomap_result[:, 0],
    ↪  isomap_result[:, 1], s=10)
    plt.title("Isomap Visualisation")
    plt.xlabel("Isomap 1")
    plt.ylabel("Isomap 2")
    plt.grid(True)
    plt.show()
    return isomap_result
```

### H. UMAP

```
def umap_(data, n_components= 2):
    numerical_data =
    ↪  data.select_dtypes(include=['float64',
    ↪  'int64'])

    # Using UMAP
    UMAP_df = UMAP(n_components=n_components,
    ↪  random_state=42)
    UMAP_comps =
    ↪  UMAP_df.fit_transform(numerical_data)

    # Create a UMAP Data frame to plot
    columns = [f'UMAP {i + 1}' for i in
    ↪  range(n_components)]
    dataframe_UMAP = pd.DataFrame(
        UMAP_comps,columns=columns)

    # Plot the UMAP
    if n_components == 2:
        plt.figure(figsize=(10, 6))
        plt.scatter(dataframe_UMAP.iloc[:,0],
        ↪  dataframe_UMAP.iloc[:,1], s = 5)
        plt.xlabel("UMAP 1")
        plt.ylabel("UMAP 2")
        plt.grid(True)
        plt.show()

    else:
        raise ValueError(' 2 dimensions is
        ↪  best')

    return dataframe_UMAP
```

### I. GMM, Spectral

```
def GMM(data, n_clusters = 3):
    # GMM clusters here (using minimal
    ↪  inputs)
    gmm =
    ↪  GaussianMixture(n_components=n_clusters,
    ↪  random_state=42)
    gmm_labels = gmm.fit_predict(data)

    return gmm_labels
```

```
def Spectral(data, n_clusters = 3):
    # Spectral clusters here
    # 'nearest_neighbours' is chosen for
    ↪  affinity because it constructs the
    ↪  graph
    # based on k-nearest neighbours. This
    ↪  method ensures local relationships
    # between data points are preserved,
    ↪  which is effective for out dataset
    # as it has some non-linear relaitonships
    spectral =
    ↪  SpectralClustering(n_clusters=n_clusters,
    ↪  affinity='nearest_neighbors',
    ↪  random_state=42)
    spectral_labels =
    ↪  spectral.fit_predict(data)

    return spectral_labels
```

## TASK 2 EXTRAS

### *J. Image Classifications: OxfordPetsIII + Kaggle*

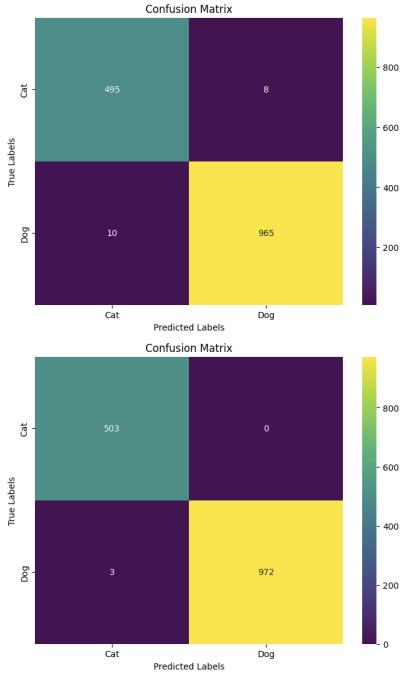


**Fig. 13:** ResNet18 Cat or Dog Classification: OxfordPetsIII

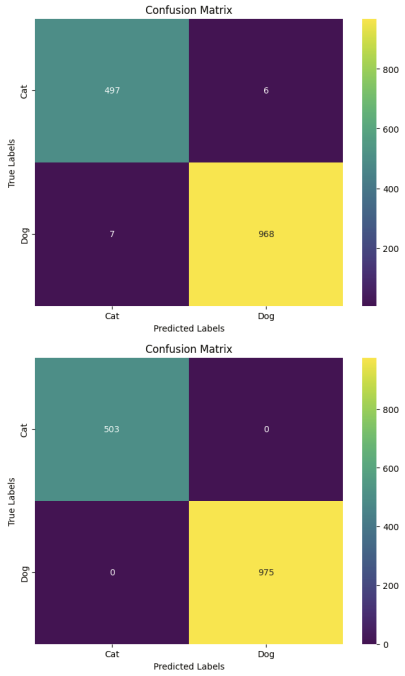


**Fig. 14:** VGG16 Cat or Dog Classification: OxfordPetsIII

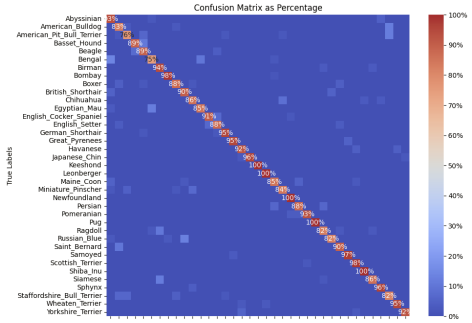


**Fig. 15:** Before Training ResNet18: OxfordPetsIII

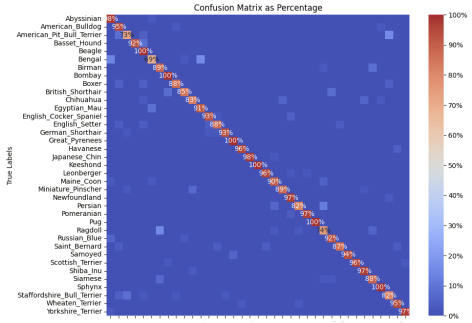


**Fig. 16:** Before Training VGG16: OxfordPetsIII

```python
# Dataset path
dataset_path = r'C:/Data/full_organised_breeds1'

# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images for VGG16
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # normalisation for ImageNet
])

# Load the dataset
dataset = datasets.ImageFolder(root=dataset_path, transform=transform)

# Use `dict_values` directly
breed_values = list(dataset.class_to_idx.keys())  # Get your custom labels
print("Custom labels from dict_values:", breed_values)

# Create DataLoader
data_loader = DataLoader(dataset, batch_size=32, shuffle=False)

# Load VGG16 pre-trained on ImageNet
vgg16 = models.vgg16(weights=models.VGG16_Weights.DEFAULT)
vgg16.eval()  # Set to evaluation mode

# Remove the fully connected layers to use VGG16 as a feature extractor
feature_extractor_vgg = torch.nn.Sequential(*list(vgg16.features.children()))

# Extract features and assign labels
vgg_features = []
vgg_labels = []

with torch.no_grad():  # No gradients needed for feature extraction
```

```
    for inputs, targets in data_loader:
        # Extract features
        features =
        ↪  feature_extractor_vgg(inputs)
        features =
        ↪  features.flatten(start_dim=1)  #
        ↪  Flatten the spatial dimensions
        vgg_features.append(features)

        # Directly use dict_values to assign
        ↪  labels
        custom_labels =
        ↪  [breed_values[target.item(
        )] for target in targets]

        vgg_labels.extend(
        targets.numpy().tolist())  # Use
        ↪  numeric indices directly

# Combine features into a single tensor
vgg_features = torch.cat(vgg_features, dim=0)

# Print results
print("Feature tensor shape:",
↪  vgg_features.shape)
print("First 10 custom labels:",
↪  vgg_labels[:10])
```

This was the VGG16 code for feature extraction. This is very similar to the code used for ResNet18.

```
# Load precomputed features and labels
features, labels =
↪  torch.load("OxfordIII_Breeds.pth",
↪  weights_only=True)

features_res_breeds = torch.tensor(features)
labels_res_breeds = torch.tensor(labels)
# Create a DataLoader for training
dataset = TensorDataset(features_res_breeds,
↪  labels_res_breeds)
data_loader = DataLoader(dataset,
↪  batch_size=32, shuffle=True)

# Load ResNet-18 pre-trained on ImageNet
resnet18 = models.resnet18(weights=
models.ResNet18_Weights.DEFAULT)

# Remove the last fully connected layer (fc)
feature_extractor =
↪  nn.Sequential(*list(resnet18.children())[:-1])

# Freeze the feature extractor
for param in feature_extractor.parameters():
    param.requires_grad = False

# Define the fully connected layer
num_classes = 37
input_dim = features_res_breeds.shape[1]  #
↪  Dimension of the extracted features
fc_layer = nn.Linear(input_dim, num_classes)

# Define optimiser and loss function
optimiser = optim.Adam(fc_layer.parameters(),
↪  lr=0.001)
criterion = nn.CrossEntropyLoss()

# Training loop
num_epochs = 10
fc_layer.train()  # Set to training mode

for epoch in range(num_epochs):
    running_loss = 0.0
    correct_predictions = 0  # Initialise
    ↪  correct predictions counter for each
    ↪  epoch

    for inputs, targets in data_loader:
        # Forward pass
        outputs = fc_layer(inputs)
        loss = criterion(outputs, targets)

        # Backward pass and optimisation
        optimiser.zero_grad()
        loss.backward()
        optimiser.step()

        running_loss += loss.item()

        # Calculate predictions and accuracy
        _, preds = torch.max(outputs, 1)  #
        ↪  Get the predicted class indices
        correct_predictions +=
        ↪  torch.sum(preds ==
        ↪  targets).item()

    # Calculate epoch loss and accuracy
    epoch_loss = running_loss /
    ↪  len(data_loader)
    epoch_acc = correct_predictions /
    ↪  len(dataset)

    print(f"Epoch [{epoch + 1}/{num_epochs}],
    ↪  Loss: {epoch_loss:.4f}, Accuracy:
    ↪  {epoch_acc:.4f}")

resnet18.fc = fc_layer

# Save the trained layer
torch.save(resnet18.fc.state_dict(),
↪  'trained_fc_layerq.pth')
print("Trained fully connected layer saved.")
```

Training code for ResNet18, same used for VGG16 but, different variables

```
Image: 4a2e40b8... .jpeg # This is a Pug (Dog)
  Pug: 100.00%
  Persian: 0.00%
  Leonberger: 0.00%
Image: Musa_1_pond.JPG # This is a Bengal (Cat)
  Bengal: 100.00%
  Saint_Bernard: 0.00%
  American_Bulldog: 0.00%
Image: Musa_2_leash.jpg # This is a Ragdoll (cat)
  Keeshond: 100.00%
  Birman: 0.00%
  Persian: 0.00%
```

"Musa_2_Leash" → Ragdoll with a leash classified as a dog. (outputs)

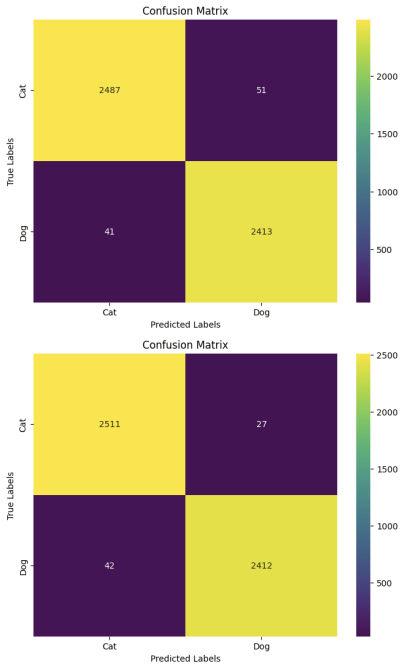


**Fig. 17:** VGG16 Cat or Dog Classification : Kaggle Pets

**To run the code**

*Climate Basel*

Climate Basel Dataset was run straight from the .csv file
This is called **ClimateDataBasel.csv**

*Kaggle*

The Kaggle dataset had labels as folders. This made the process much easier as folders held the labels. The folder name was **Kaggle** instead of the automatic archive file Kaggle saves it as.

```
Kaggle_cats_dogs = r"C:\Data\Kaggle\kagglecatsanddogs_3367a\PetImages"
```

*OxfordPetsIII*

OxfordPetsIII from the website.
I will provide the folder with the labels for OxfordPetsIII. It is called **all_columns.csv**
OxfordPetsIII images folder was named **images** and this is the folder I separated locally into 2 classes and then into 37 classes
These are the OxfordPetsIII paths. Everything was run locally.

```
full_data = r"C:\Data\images"
Cat_or_Dog = r"C:\Data\CatorDog"
Breeds_Full_Path = r"C:/Data/full_organised_breeds1"
```

The only file absolutely required is images (rename downloaded images to this)
CatorDog will be automatically made in the .ipynb script
full_organised_breeds1 will also be made automatically in the script

*Extra Files*

The "**my_test**" folder is in the code provided in the submission. So, you can use :

```
code\my_test
```