Comparing Generative Adversarial Networks and Variational Autoencoders for Image Data Augmentation in Sunset Classification

Minerva University

CS156: Finding Patterns in Data with Machine Learning

Professor Watson

April 2023

Table of Contents

Executive Summary

This paper addresses the binary classification problem of categorising images as either sunset or not sunset. Data augmentation was employed using generative adversarial networks (GANs) and variational autoencoders (VAEs) to expand the training set. The VAE with early stopping after 1000 training epochs yielded superior results, generating more visually accurate sunset images with less noise.

Five models were utilised for classification: logistic regression, support vector classifier (SVC), K-nearest neighbours (KNN), decision tree, and MobileNetV2. A five-fold cross-validation and grid search were performed to determine the optimal classifier and hyperparameters. Precision was chosen as the primary metric to minimise the cost of false positives, which could lead to privacy breaches if non-sunset images were shared on a public sunset page.

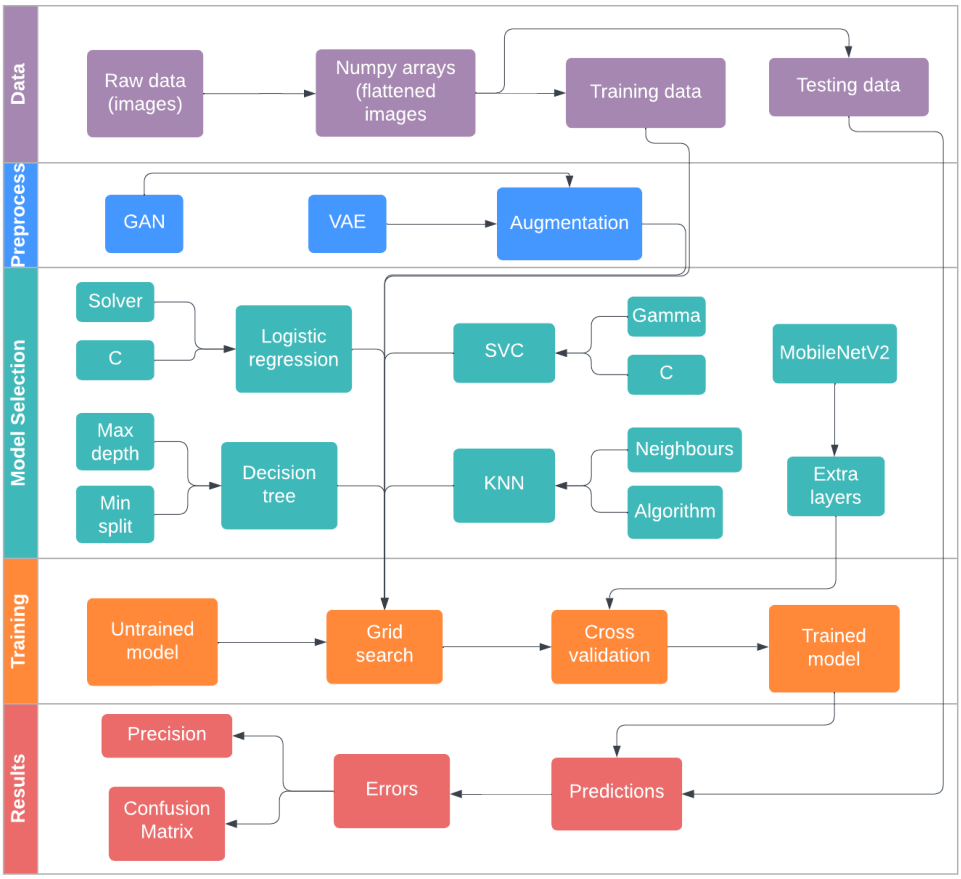
The support vector classifier with a radial basis function and C value of 10 delivered the highest validation precision score of 0.980. MobileNetV2 achieved a precision validation score of 0.944 after 10 training epochs, showing no further improvement with additional epochs.

The SVC with the aforementioned configuration was used to report the test set performance metrics, resulting in a precision of 0.869, AUC of 0.923, and accuracy of 0.92. The confusion matrix analysis revealed 3 false negatives and 23 false positives out of 325 images.

A model harness is displayed in Figure 1 below.

Figure 1.

Model Harness



The Data

The aim of this project is to classify photos as 'sunset' or 'not sunset' to determine which images from my camera roll I should post to different Instagram accounts. Though I am not an avid social media participant, I do have two Instagram pages that I use occasionally. One of these is my personal account, where I post photos of myself and my friends and family. The other I use to post pictures of sunsets.

Included in this assignment are 154 photos from my Instagram accounts (55 from my personal account, and 96 from my sunset account) used as training data. I initially wanted to

pull these directly from the accounts, however I didn't want to risk my privacy by making them public, and so I settled for downloading the images instead. Also included are 325 photos from my camera roll that were hand-selected by me as photos I would post on one of these Instagram pages (170 of these were non-sunset images, and the other 155 sunsets). The data is located in the google drive folder found here, separated into training & testing sets, and sunsets & non-sunsets.

The test set here is not taken as a random sample from the training data since it will be more valuable to see how the model performs on new data, and to report metrics on a new set of data that represents the models performance on real-world examples. The model will be optimised on the training data using cross validation, and metrics reported on the test data.

All individuals included in the images have consented to being included in this assignment. Given that these are personal photos and that there are privacy concerns, the photo albums have only been shared with individuals with Minerva University emails who have the link included in this assignment. They have not been shared with individuals outside this circle. The insights from this assignment would only be used by myself, since the model is very specific to my situation and is only really applicable to my data (my photos). False positives could breach privacy in my case, since posting 'non-sunset' photos to a non-private Instagram account shares personal information with lots of people. This is discussed in more detail in the following sections.

This notebook can be run in google colab here. Note that file paths will need to be changed to those in the local drive.

The Task

The task here is to classify the images as either sunset or not sunset. Since there are only two categories, this is a binary classification problem. To achieve this classification, there

are multiple steps that we must take. Firstly, we will augment the data to expand the training set. We will use two methods (generative adversarial networks and variational autoencoders) and compare the results based on multiple metrics (loss, accuracy, FID score, inception score, KL divergence, and visual inspection). After augmenting the data, we will use multiple models (logistic regression, SVC, KNN, decision tree and MobileNetV2) and find the one that produces the best performance.

Data Augmentation

There are some basic methods of data augmentation that can be used to increase the amount of training data in a dataset. Some of these methods include stretching images, zooming in on images, and rotating images. However, all of these techniques involve transformations that will eliminate the key features of ‘sunset images’, which I believe to be a distinct block of darkness at the bottom of the screen, and lightness in the middle (the sun) emanating outwards to the top and sides of the image.

We might also choose to augment our data with online datasets. In this case, however, there are no readily available datasets of sunsets specifically.

Consequently, we require a more sophisticated data augmentation technique that will retain the features of sunsets. In this section, we will compare the sunset images produced by a generative adversarial network to those produced by a variational autoencoder.

Generative Adversarial Network (GAN)

What are GANs?

Generative Adversarial Networks (GANs) are a class of machine learning models that consist of two neural networks, a generator and a discriminator. These two networks compete against each other to improve their performance. The discriminator aims to distinguish between

the training set and the generated images from the generator, and the generator aims to fool the discriminator by producing images that match the same underlying data distribution as those in the training set. GANs have been applied in various fields, including image synthesis, data augmentation, and art generation.

How do GANs work?

First, we initialise the generator (later referred to as G) and discriminator (later referred to as D) networks with random weights. We then set the number of training iterations (epochs), batch size, learning rates and optimisers for G and D. These are hyperparameters of GANs that should be altered to produce the best model performance. Here, we use 15,000 epochs (due to computational constraints), a batch size of 32, a learning rate of 0.0002 (after an informal gridsearch over different learning rates, this was found to produce the best results) and the Keras optimizers 'Adam'.

We can then train the networks. For each epoch:

1. Sample a mini-batch of real data (x) from the dataset.
2. Generate a mini-batch of random noise vectors (z) using a suitable distribution (Gaussian in this case).
3. Use G to generate fake data samples ($G(z)$) from the noise vectors (z).
4. Train D to classify real and fake samples:
    4.1. Forward pass:
        4.1.1. Pass real samples (x) through D: $D(x)$. This computes the probability that each sample is real according to the discriminator.
        4.1.2. Pass random noise vectors (z) through G: $G(z)$. This generates fake samples.
        4.1.3. Pass the fake samples through D: $D(G(z))$. This computes the probability that each generated sample is real according to the discriminator.

4.2. Calculate the loss:

4.2.1. $L_{real} = log(D(x))$: The loss for real samples measures how well D classifies real samples as real.

4.2.2. $L_{fake} = log(1 - D(G(z)))$: The loss for fake samples measures how well D classifies fake samples as fake.

4.2.3. Compute the total discriminator loss:

$L_D = L_{real} + L_{fake} = log(D(x)) + log(1 - D(G(z)))$. This combines the losses for real and fake samples, providing a single value that represents how well D distinguishes between them.

4.3. Backpropagation: Compute the gradients of D with respect to D's parameters (weights and biases) using the chain rule. These gradients indicate the direction of the steepest increase in the loss function.

4.4. Update D's parameters using gradient descent: Adjust the weights and biases of D by subtracting a fraction (learning rate) of the computed gradients. This moves D's parameters in the direction of the steepest decrease in L_D, improving D's classification performance.

5. Train G to generate samples that can fool D:

5.1. Forward pass:

5.1.1. Generate a new mini-batch of random noise vectors (z).

5.1.2. Pass the noise vectors through G: $G(z)$. This generates new fake samples.

5.1.3. Pass the fake samples through D: $D(G(z))$. This computes the probability that each generated sample is real according to the discriminator.

5.2. Calculate the generator loss: $L_G = -\ log(D(G(z)))$. This loss measures how well G generates samples that D classifies as real. The generator aims to maximise the probability that D classifies its outputs as real.

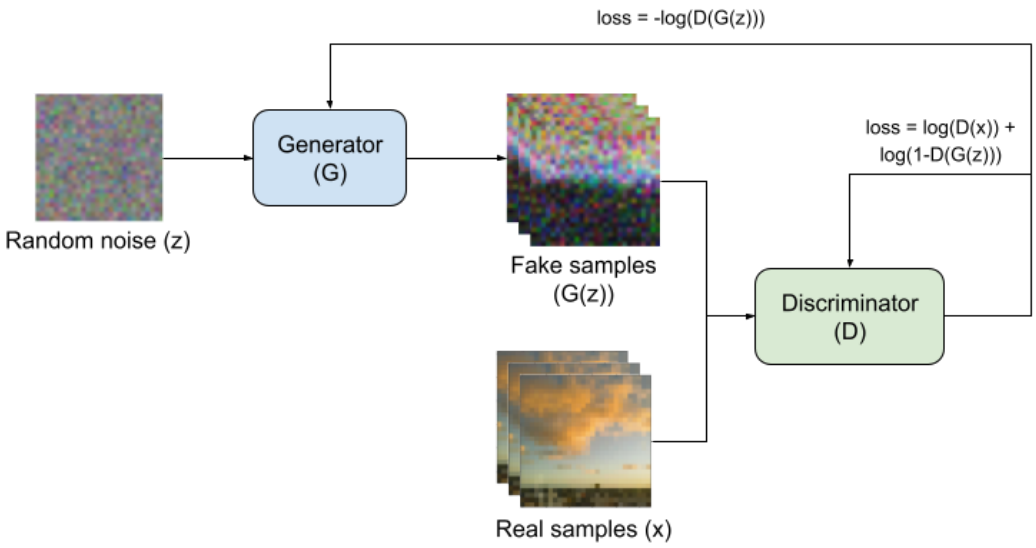
5.3. Backpropagation: Compute the gradients of L_G with respect to G's parameters (weights and biases) using the chain rule. These gradients indicate the direction of the steepest increase in the loss function.

5.4. Update G's parameters using gradient descent: Adjust the weights and biases of G by adding a fraction (learning rate) of the computed gradients. This moves G's parameters in the direction of the steepest increase in L_G, improving G's ability to generate samples that D classifies as real.

This training process is repeated until a stopping criterion is met, such as a fixed number of iterations for convergence of the loss functions (a fixed number of 15,000 epochs in this case). The process is illustrated in Figure 2.

Figure 2.

Generative Adversarial Network Training for Sunset Image Generation



Discriminator Performance

After training the GAN, we should evaluate its performance. There are several metrics, both quantitative and qualitative, that we should use to evaluate the performance of the GAN. The first is discriminator loss. Discriminator loss (see Figure 3) tells us how well the discriminator network is managing to distinguish between real and fake images. In our case, the loss decreases over time, indicating that the discriminator is improving and is able to distinguish more readily between the images from the generator and our training data.

Figure 3.

Discriminator Loss Over GAN Training

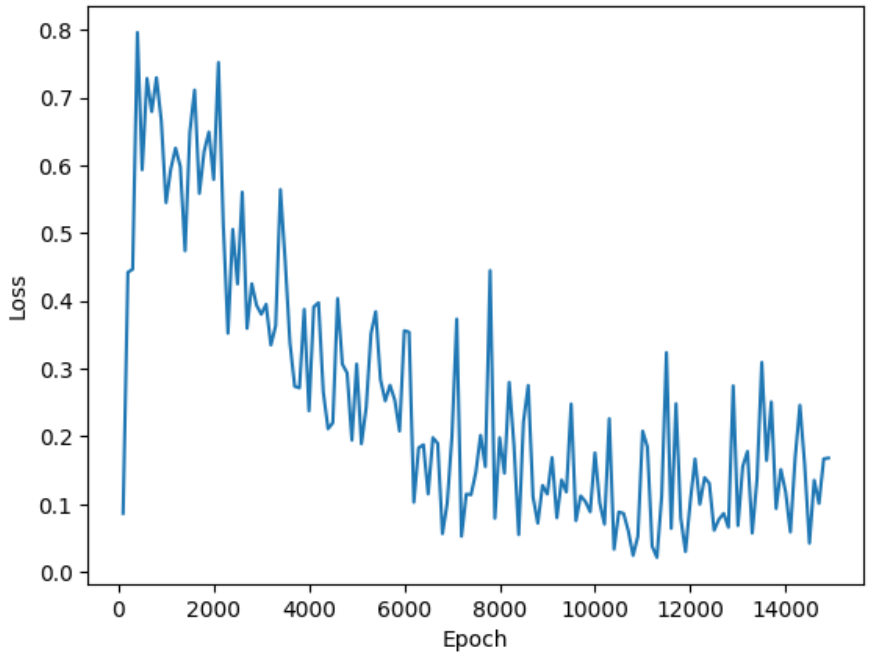
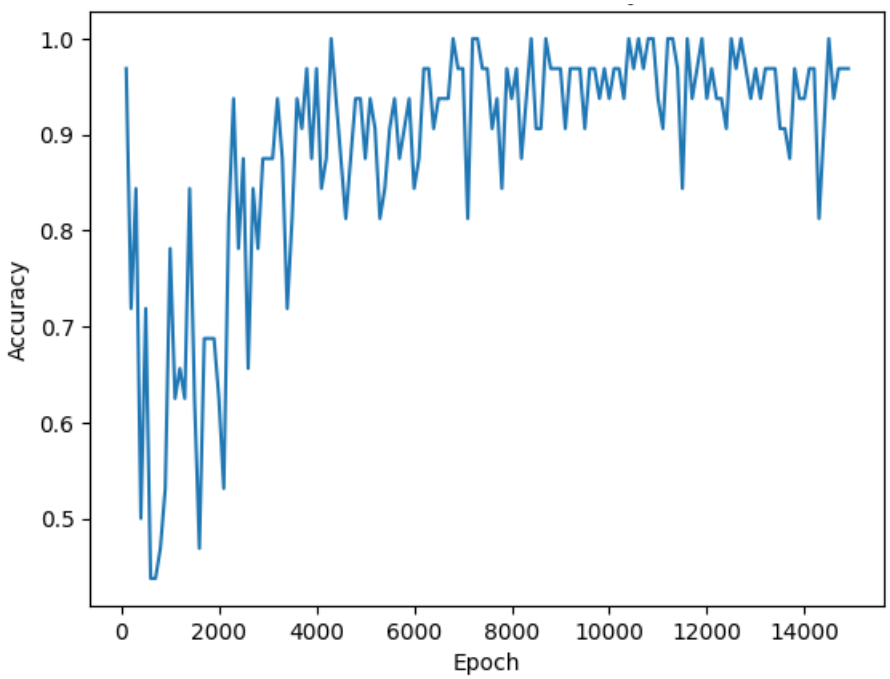


Figure 4 shows the discriminator accuracy over time. This is another measure that tells us how well the discriminator is performing over the training period. We see that, over the first few thousand epochs, the accuracy of the discriminator improves, meaning that it is more readily able to discriminate between the fake and real images. This corroborates the information from Figure 3.

Figure 4.

Discriminator Accuracy Over GAN Training

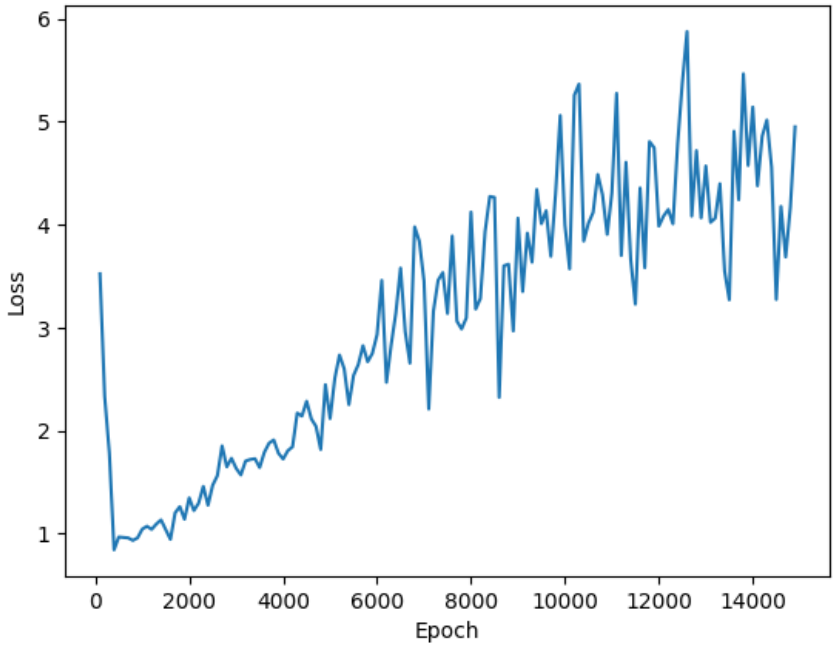


Generator Performance

The performance of the generator over time can be determined by its loss function (see Figure 5 below). Here, we see that the generator loss actually increases over time. This means that its performance relative to the discriminator is not improving.

Figure 5.

Generator Loss Over GAN Training



The combined information from Figures 3, 4 and 5 suggests that the discriminator may be dominating the generator. There are a few things we can (and have) tried to rectify this issue.

The first is reducing the learning rate of the discriminator/increasing the learning rate of the generator. Playing around with these values in the code, we found that, upon visual inspection, the images produced by these learning rates and the Adam optimiser were the best.

We might also try adapting the model architecture. For example, we have tried adding extra dense layers to the generator and discriminator. Again, this resulted in generated images that were visually not as good as those produced by the model used here.

We might also decide to stop the training early (after fewer epochs) where both the discriminator and generator loss are low.

FID Score

The Fréchet Inception Distance (FID) score measures the distance between the feature representations of the real and generated images in a high-dimensional feature space. Specifically, it computes the Fréchet distance between the multivariate Gaussian distribution of the real images and the generated images. FID scores are calculated as follows:

1. Feature extraction: First, we need to extract features from both the generated images and real images. We pass the images through the Inception-V3 network (trained on ImageNet) and extract the activations from an intermediate layer. These activations are used as feature vectors representing the images.
2. Calculate statistics: Now, we calculate the mean and covariance of the feature vectors for both the generated and real images. Let μ_g and μ_r denote the mean feature vectors for the generated and real images, respectively, and Σ_g and Σ_r denote their respective covariance matrices.
3. Fréchet distance: The FID score is based on the Fréchet distance between two multivariate Gaussian distributions. If we assume the feature vectors follow a multivariate Gaussian distribution, we can compute the Fréchet distance between the distributions of generated and real images. The Fréchet distance between two multivariate Gaussian distributions with means μ_g and μ_r, and covariance matrices Σ_g and Σ_r, is given by:

$$FID = ||\mu_g - \mu_r||^2 + Tr(\Sigma_g + \Sigma_r - 2(\Sigma_g\Sigma_r)^{0.5}) \quad (1)$$

where $||.||$ denotes the L2 norm (Euclidean distance), $Tr(.)$ is the trace of a matrix (sum of diagonal elements), and $(.)^{0.5}$ represents the square root of the matrix (computed using the positive definite square root). This term can be broken down as follows:

a. Mean difference: $||\mu_g - \mu_r||^2$ is the squared Euclidean distance between the means of the generated and real image distributions. This term penalises the model if the generated images have a different mean feature vector than the real images. In other words, it measures how far apart the average generated image is from the average real image in terms of features.

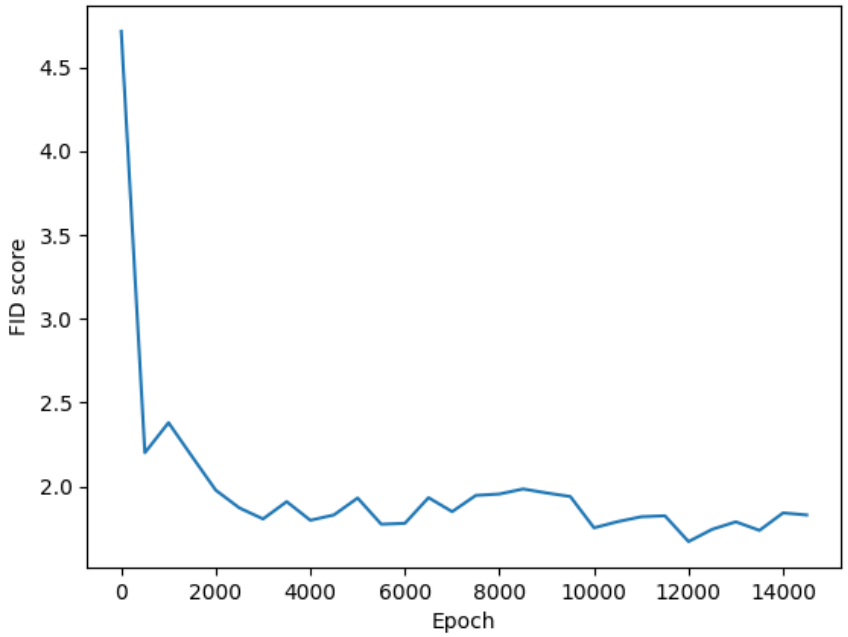
b. Covariance difference: $Tr(\Sigma_g + \Sigma_r - 2(\Sigma_g \Sigma_r)^{0.5})$ captures the difference in the spread and shape of the distributions by comparing their covariance matrices. This term penalises the model if the generated images have different variations or correlations between their features compared to the real images. The trace function is used to reduce the matrix difference to a scalar value, making it easier to combine with the mean difference term.

4. Interpretation: A lower FID score indicates better quality of the generated images, as it means that the generated images are closer to the real images in terms of their feature representations. A high FID score, on the other hand, indicates that the generated images are significantly different from the real images and may lack certain characteristics or details.

We can see from Figure 6 that the FID score decreases over the course of training, indicating the discrepancies between the real and generated samples are decreasing. This suggests that our training is working, and that the feature representations of the generated samples are becoming more similar to the real samples.

Figure 6.

FID Score Over GAN Training



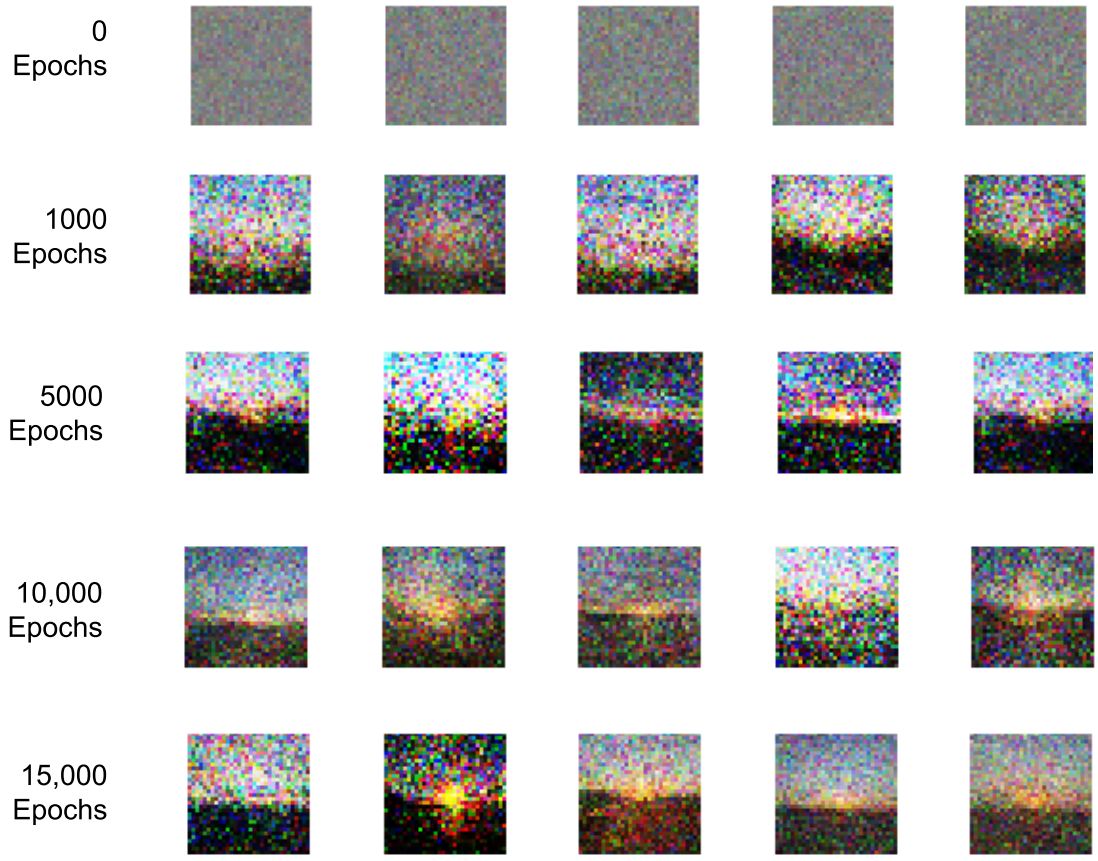
Visual Inspection

Finally, we will also consider the performance of the GAN using visual inspection. From Figure 7, we see that the images produced by the generator portray the general features of sunsets. By epoch 1000, we see that there is a distinct area of darkness at the bottom of the image and a lighter block at the top. By epoch 10,000, we see that the lightest portion of the image lies in the middle and is circular in shape (somewhat sunlike).

Despite the presence of the key features of sunsets that we identified at the start of this data augmentation section, there is still a significant amount of noise present in the images, even by epoch 15,000. Changing the model architecture, learning rates, optimiser, and batch size did not affect this noise. We speculate that it is a cause of the lack of training data.

Figure 7.

Visual Performance of GAN Over Training



Variational Autoencoder (VAE)

What are VAEs?

Similarly to autoencoders (AEs), variational autoencoders (VAEs) consist of two parts, an encoder and a decoder. The encoder consists of several layers with subsequently fewer nodes and the decoder layers with subsequently more nodes, rendering a bottleneck in the middle of the network. In AEs, the bottleneck contains key features of the training dataset. However, in VAEs, the bottleneck contains parameters (mean and log variance) of a Gaussian

distribution (or other probability distribution). It is from this probabilistic distribution that the network generates images in the decoder.

How do VAEs Work?

First, we initialise the encoder and decoder networks and set the hyperparameters (learning rate, optimiser, number of training epochs, and batch size). These hyperparameters should be set carefully to produce the best performance. Here, we use 10,000 epochs (due to computational constraints), a batch size of 100, a learning rate of 0.001 (after an informal gridsearch over different learning rates, this was found to produce the best results) and the Keras optimizers 'Adam'.

We can then train the model as follows. For each epoch:

1. Randomly divide the dataset into batches and sample a batch.
2. Forward pass:
    2.1. Feed the real images (x) to the encoder.
    2.2. Compute the mean (μ) and log variance ($log(var)$) of the latent space distribution using the encoder output.
    2.3. Sample an value ϵ from a standard Gaussian distribution ($\epsilon \sim N(0, 1)$).
    2.4. Compute the standard deviation (σ) by taking the exponential of half the log variance ($\sigma = e^{log(var)/2}$).
    2.5. Compute the sampled latent vector (z) using the parameterization: $z = \mu + \sigma\epsilon$.
    2.6. Feed z to the decoder.
    2.7. Compute the generated images ($\hat{x}$) using the decoder output.
3. Calculate the loss:
    3.1. Compute the loss (L) between the true images (x) and the generated images ($\hat{x}$). Here, we use binary cross-entropy loss, which is given as:

$$L_{BCE} = -\sum x log(\widehat{x}) + (1 - x) log(1 - \widehat{x}) \quad (2)$$

3.2. Compute the KL divergence (L_{KL}) between the learned gaussian distribution and the standard gaussian as:

$$L_{KL} = \frac{\sum e^{log(var)} + \mu^2 - 1 - log(var)}{2} \quad (3)$$

This is a regularisation term that ensures that the learned gaussian distribution does not sway too far from the standard gaussian.

3.3. Compute the total loss as $L_{total} = L_{BCE} + L_{KL}$.

4. Perform backpropagation to compute the gradients of the loss with respect to the model parameters.

4.1. Calculate the gradient of the total loss (L_{total}) with respect to the decoder output: $dL_{total}/d\widehat{x}$.

4.2. Backpropagate the gradients through the decoder network, computing the gradients for all weights and biases for each layer: $dL_{total}/dW_{decoder}$ and $dL_{total}/db_{decoder}$.

4.3. Calculate the gradient with respect to the latent vector: dL_{total}/dz.

4.4. Backpropagate the gradients back through the reparameterization trick. Here, the chain rule is applied as follows:

4.4.1. For $dL_{total}/d\mu$: Given that $z = \mu + \sigma\epsilon$, $dz/d\mu = 1$. Thus,

$$dL_{total}/d\mu = dL_{total}/dz \cdot dz/d\mu = dL_{total}/dz.$$

4.4.2. For $dL_{total}/d\sigma$: Given that $z = \mu + \sigma\epsilon$, $dz/d\sigma = \epsilon$. Also given that $\sigma = e^{log(var)/2}$, $d\sigma/dlog(var) = e^{log(var)/2}/2 = \sigma/2$. Thus, $dL_{total}/d\sigma = dL_{total}/dz \cdot dz/d\sigma \cdot d\sigma/dlog(var) = dL_{total}/dz \cdot \epsilon\sigma/2$.

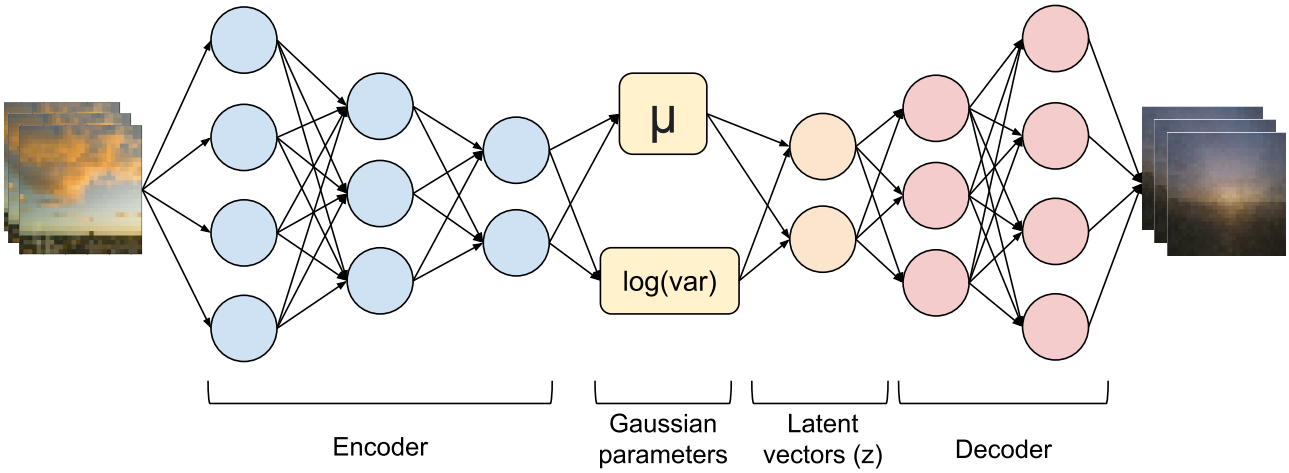
4.5. Backpropagate the gradients through the encode, computing the gradients for all layers with respect to their weights and biases: $dL_{total}/dW_{encoder}$ and $dL_{total}/db_{encoder}$.

5. Update the model parameters using the gradients computed in backpropagation and the learning rate.

This training process is repeated until a stopping criterion is met, such as a fixed number of iterations for convergence of the loss functions (a fixed number of 10,000 epochs in this case). The general architecture of variational autoencoders is displayed in Figure 8 below.

Figure 8.

Variational Autoencoder Architecture for Sunset Image Generation

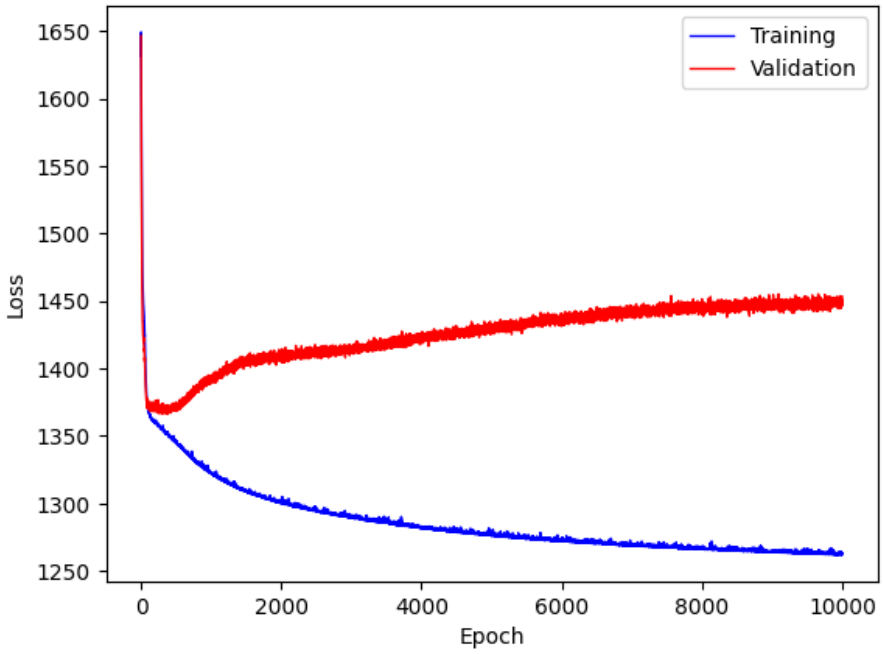


Loss

As with GANs, we will use a variety of metrics to determine the performance of the VAE. The first is binary cross-entropy loss, the first portion of the total loss function. Over the training process (Figure 9), we see that loss initially decreases for both training and validation groups. However, after ~500 epochs, the validation loss begins to increase again. This suggests that the model has begun to overfit the data and is learning noise that is specific to the training group and not generalisable to the validation group images.

Figure 9.

Binary Cross-Entropy Loss Over VAE Training



KL Divergence

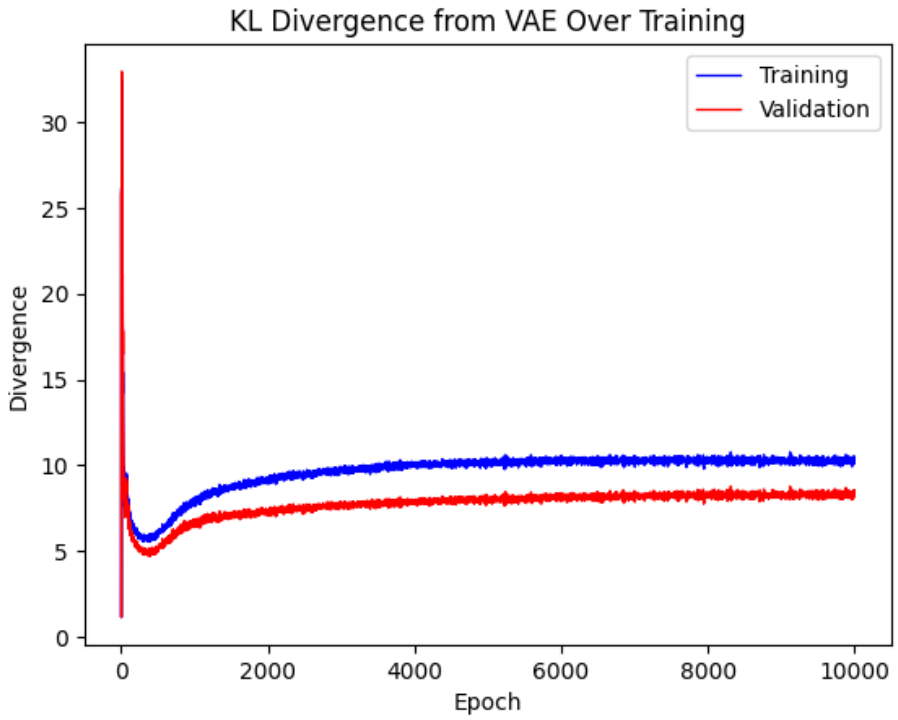
Another metric that we will consider here is Kullback-Leibler (KL) divergence, the second portion of the overall loss function for the network. The KL divergence, given in equation (3) above, measures the difference between the learned gaussian distribution and the standard

gaussian distribution. By minimising this term, the VAE ensures that the latent space representation does not diverge too much from the chosen prior distribution.

From Figure 10, we see that the KL divergence decreases initially, showing that the model is learning to represent the data in a way that is closer to the prior standard gaussian distribution. After ~500 epochs, the divergence increases slightly and then stabilises at a small divergence value, suggesting that the model has learned a distribution that doesn't deviate too much from the standard gaussian. The training and validation set follow the same patterns, thus there is no indication of overfitting.

Figure 10.

KL Divergence Over VAE Training



Visual Inspection

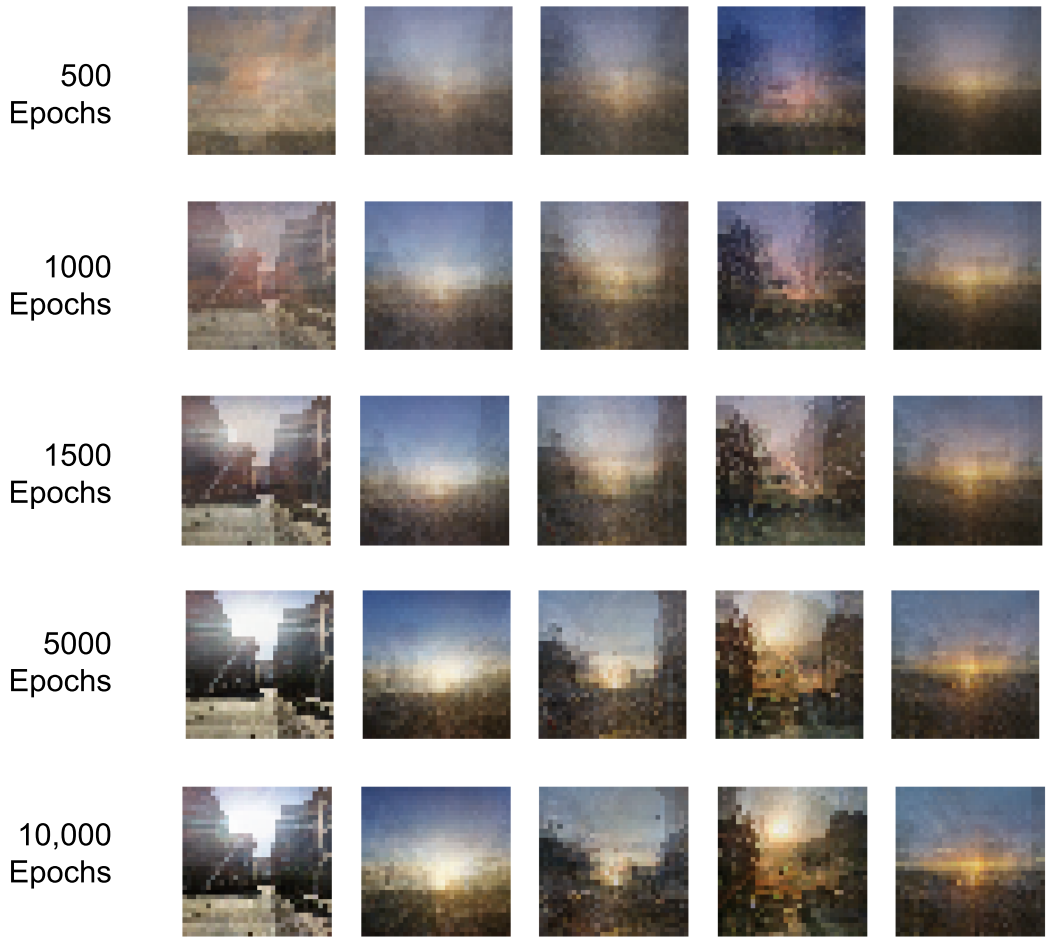
Finally, as with GANs, we will use visual inspection to determine how good the generated images are from VAEs. After 500 epochs, the model has already learnt the key features of sunsets (dark block at the bottom, and a circle of light emanating to the top and sides of the image). The colours are also yellow-ish around the sun and blue for the sky which matches with our knowledge of sunsets.

After 5000 epochs, it appears that the model may be overfitting on the training images. The generated images appear to closely resemble those in the training set. This is in line with our observations about the loss function above.

Given that we have already included regularisation terms and layers, there are very few layers in the model and we cannot add any more training data, there is not much we can do to prevent this overfitting. Thus, we might choose to stop the training process early rather than training for all 10,000 epochs. Stopping around 500-1000 epochs seems like a good point, where the images resemble sunsets and before they appear too similar to the training set.

Figure 11.

Visual Performance of VAE Over Training



Comparison of Data Augmentation Methods

On balance, it seems that the VAE produced images that were much more in-line with what we would imagine a sunset to look like. This model generated images with much less noise than the GAN. However, the best images were produced by the VAE with early stopping. We will thus augment our current training dataset with images from the VAE model after 1000 training epochs.

Model Selection for Classification of Sunsets

Logistic Regression

Logistic regression works by fitting a sigmoid function to the data as follows:

1. Initialise the model parameters: $weights = [0, 0, \ldots, 0]$ (one weight for each predictor, initialised to 0) and $bias = 0$.
2. Define the logistic function: $sigmoid(x) = \frac{1}{1+e^{-x}}$.
3. For each iteration of the training process:
    a. Calculate the weighted sum of the predictors and the bias:

    $z = \sum_{i=1}^{n} weights_i \cdot X_i + bias$, where n is the number of predictors.

    b. Pass the weighted sum through the logistic function to get the predicted probability of the image being of a sunset (positive class):

    $prediction = sigmoid(z)$.

    c. Calculate the error between the predicted probability and the actual label:

    $error = y - prediction$.

    d. Update the weights and bias based on the error and a learning rate (α):

    $weights_i = weights_i + \alpha \cdot error \cdot X_i$ and $bias = bias + \alpha \cdot error$.
4. Repeat step 3 until the model has converged (the error doesn't significantly change over a number of iterations) or a maximum number of iterations has been reached.
5. After the model has been trained, new data can be input and the predicted probabilities can be calculated using the logistic function and the final weights and bias.

In scikit-learn, the logistic regression implementation uses an optimisation algorithm, such as liblinear or lbfgs, to find the optimal weights that minimise the cost function instead of

the iterative approach outlined above. This allows for a more efficient and scalable solution, especially when the number of predictors is large as in this case. Additionally, scikit-learn's logistic regression implementation also uses an 'L2' penalty that is added to the cost function to prevent overfitting.

SVC

Support-vector classifications works by creating a hyperplane in high-dimensional space that separates the data belonging to sunset and non-sunset groups as follows:

1. Initialise the parameter C, which determines the trade-off between achieving a low training error and a low testing error, and the kernel function, which transforms the flattened image data into a higher-dimensional space where a linear boundary can better separate the classes, sunset and not sunset.
2. Train the SVM model by solving the following optimisation problem: $min\frac{1}{2}|w|^2 + C\sum_{i=1}^{n}\xi_i$.

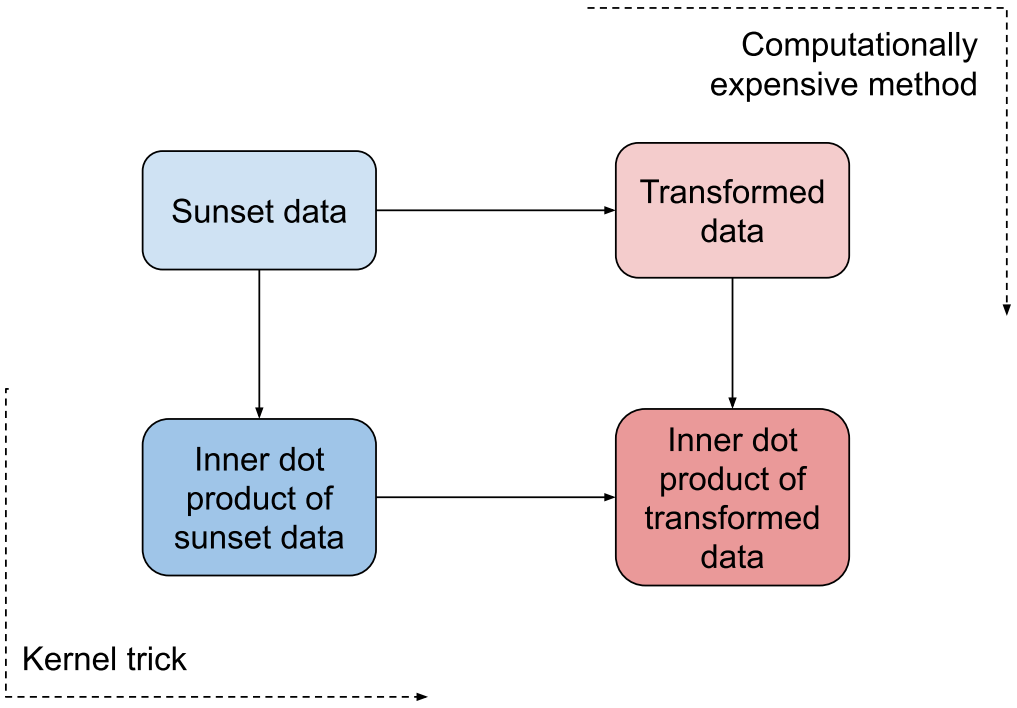
   Subject to the constraints: $y_i(w^T x_i + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$,

   where w is the weight vector, b is the bias term, x_i is the slack variable for the ith training example (x_i, y_i), and C is the regularisation parameter. The optimisation problem is aimed to find the optimal values for w, b, and x_i that maximises the margin between the hyperplane (defined by w and b) and the closest data points (support vectors) while still accommodating some misclassified examples through the use of slack variables.
3. Use the obtained hyperplane parameters w and b to make predictions on new data by computing the dot product of the feature representation $\Phi(x)$ of a new example x and the weight vector w, plus the bias b. The sign of the result determines the predicted class label.

In SVCs, we can use kernels to improve the model performance. This works by transforming the data into a higher-dimensional space to improve the chances that it is linearly-separable. This, however, can be a computationally expensive endeavour (not the transformation itself, but the steps that come after transformation). So, instead of actually transforming the data we can use the kernel trick. This involves implicitly calculating the inner dot product of the transformed data via the inner dot product of the original image data. This means that we never have to actually transform the data into a higher dimensional state. This process is illustrated in Figure 12.

Figure 12.

Kernel Trick Inner Workings



There are several types of kernels which we consider here:

1. Radial basis functions: $K(x, y) = e^{-\gamma||x-y||^2}$.

2. Polynomial basis functions: $K(x, y) = (x \cdot y + c)^d$.
3. Sigmoid basis functions: $K(x, y) = tanh(\alpha(x \cdot y) + c)$.

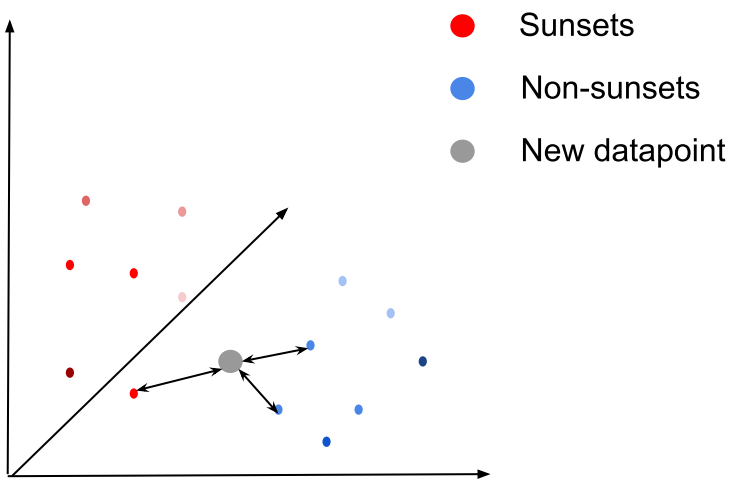
KNN

The k-nearest neighbours (KNN) model works a bit differently from other models. Instead of training on the data per se, the algorithm just stores the initial training data in the original space. We can imagine the image data at points in a high-dimensional space which are labelled as either ‘sunset’ or ‘not sunset’. Hopefully, these points form two distinct clusters, one containing the sunset images and the other containing the non-sunset images.

When new classifications need to be made, the k-nearest neighbours are taken and a majority vote decides how the new datapoint should be classified. The number of nearest neighbours will depend on the problem at hand. In this case, we optimise over k as a hyperparameter and test out 3, 5, 7, and 9 neighbours. Figure 13 shows how these distances are calculated for new data points using 3 nearest neighbours.

Figure 13.

KNN Distances Using 3 Nearest Neighbours



Note. This diagram uses only 3 dimensions for explanatory purposes. In reality, these distances are calculated in a higher-dimensional space.

Decision Tree

Decision trees work by creating linear decision boundaries in space (eg. Figure 14). The training process is as follows. First, we start with an empty decision tree (root node). Then, for each node in the tree, perform the following steps:

1. Calculate the impurity of the node (e.g., Gini index): Gini index is a measure of impurity, which calculates the probability of misclassifying an instance if it were randomly labelled according to the class distribution in the dataset.

   The Gini index for a node can be calculated as $Gini(node) = 1 - \sum_{i=1}^{n} p_i^2$, where p_i is the probability of class i in the node.

2. Select the best feature (and the corresponding split value) to split the node based on the Gini indexes and decision tree algorithm. This algorithm is as follows:

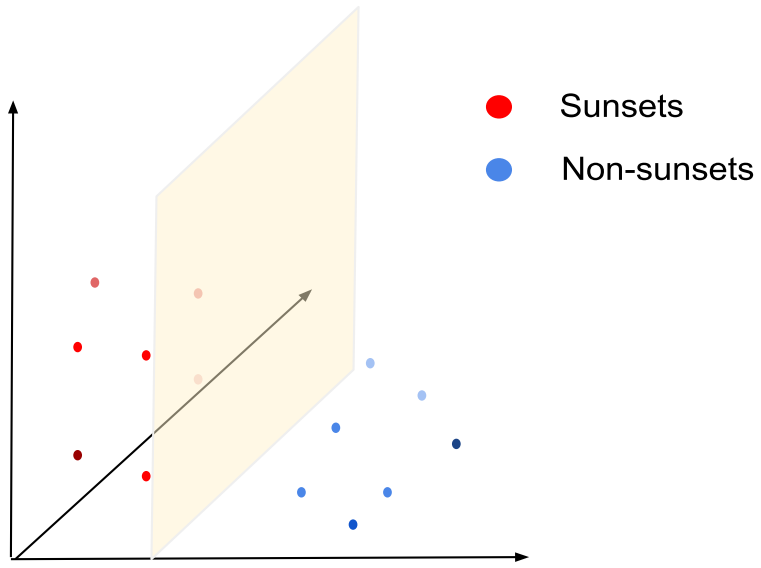
   a. Sort the feature values.

   b. Calculate the Gini index for each possible split value:
      i. Split the dataset into two subsets based on the split value.
      ii. Calculate the weighted average of the Gini indices for the two subsets.
   c. Choose the split value that results in the lowest weighted Gini index.
3. Check stopping criteria:
   a. Determine if the stopping criteria, such as maximum tree depth, minimum samples per leaf, minimum information gain, or maximum number of leaf nodes, have been met.
   b. If any of the stopping criteria are met or no significant improvement in impurity is observed, proceed to step 4. Otherwise, proceed to step 5.

4. Mark the node as a leaf and assign the majority class label as the output:
   a. If the stopping criteria are met, declare the current node as a leaf node.
   b. Assign the majority class label (the most frequent class) of the samples in the node as the output for this leaf node.
5. Create child nodes for each split and repeat the process from step 2:
   a. If the stopping criteria are not met, split the current node into two child nodes based on the best feature and split value determined in step 2.
   b. For each child node, repeat the process starting from step 1 (calculating the impurity of the node) until all nodes in the tree are processed.

Figure 14.

Example Decision Boundary Split



Note. This diagram uses only 3 dimensions for explanatory purposes. In reality, these distances are calculated in a higher-dimensional space.

MobileNetV2

MobileNetV2 is a pre-trained model that has been widely used for large-scale image classification. This model is suitable as a base model for binary classification problems like identifying whether an image is a sunset or not. It has already learned to extract high-level features from millions of images with thousands of labels, and it has achieved high accuracy on the MobileNetV2 benchmark, indicating that it can generalise well and perform accurately on a range of tasks. In addition, MobileNetV2 was trained on a diverse dataset of over a million images, which includes many different categories, such as animals, vehicles, and natural scenes which means that it should contain the necessary features to categorise sunsets. Using a pre-trained model like MobileNetV2 allows us to save time and resources. This capability, combined with its high accuracy and diverse training data, makes MobileNetV2 an ideal choice for binary classification tasks like identifying sunset images.

Using a pre-trained model like MobileNetV2 also helps to prevent overfitting when training a model on a smaller dataset for a specific task. This can happen when there are too few examples in the training set, or when the model is too complex for the task at hand. By starting with a pre-trained model, the model has already learned to recognise common features and patterns, which allows it to focus on learning the specific features that are important for the binary classification task. This can lead to better generalisation and prevent the model from overfitting to the training data.

The CNN model architecture is based on the MobileNetV2 convolutional neural network architecture. It takes as input an image of size (224, 224, 3) and produces a single output representing whether the image is a sunset or not. The MobileNetV2 architecture consists of multiple convolutional and pooling layers followed by a few fully connected layers. In this model, the MobileNetV2 architecture is used as a base model with its top layers (fully connected layers) removed. The output of the base model is then fed into a global average pooling layer, followed by a fully connected layer with 1024 units and a ReLU activation function, and a dropout layer with a dropout rate of 0.5. We add an L2 regularisation term to this layer to ensure that the model doesn’t overfit the data. Finally, a single neuron output layer is added with no activation function. The entire model is trained using the binary cross-entropy loss function and the Adam optimiser with a learning rate of 0.0001. The layers of the base model are frozen to prevent overfitting, while the additional layers are trained to learn features specific to the binary sunset classification task.

The purpose of the 1024 ReLU fully-connected layer in the model is to perform dimensionality reduction and learn higher-level features specific to the binary sunset classification task. The ReLU activation function helps to introduce non-linearity in the output of this layer, which can increase the model's expressive power and improve its ability to learn complex features.

During the training process, the forward pass is used to compute the output of the model given an input. In this case, the input is an image, and the output is a prediction of whether the image is a sunset or not. The forward pass involves passing the input through the layers of the model to compute the output.

The backpropagation algorithm is used to compute the gradients of the loss function with respect to the parameters of the model, which are used to update the parameters during the optimisation process. In this case, the loss function is binary cross-entropy, which measures the difference between the predicted probability of the image being a sunset and the true label.

During the backpropagation step, the gradients of the loss function with respect to the parameters of the model are computed using the chain rule of calculus. These gradients are then used to update the parameters of the model in the opposite direction of the gradient, with the aim of minimising the loss function.The parameters that are updated during backpropagation are the weights and biases of the dense layers added on top of the pre-trained MobileNetV2 base model. The pre-trained weights in the base model are kept frozen during training, as they have already been trained on a large dataset and are considered useful features for the task of sunset classification.

Cross Validation Results

Cross Validation Process

To train all the above models, the data is split into five folds, and a grid search is completed to find the optimal classifier (logistic regression versus SVC) and respective parameters. For logistic regression, the hyperparameters to tune include C, a regularisation term to reduce overfitting, and the solver, the algorithm used to find the optimal weights and bias that minimise the cost function. For SVC, the hyperparameters to tune include C (same definition as for logistic regression), and gamma, a parameter that determines the flexibility of

the decision boundary. For KNN, the hyperparameters to tune include the number of nearest neighbours (used to classify new data points) and the algorithm used to train the KNN. Finally, the hyperparameters for the decision tree include the maximum depth and the minimum split (the number of nodes that need to be in a left/right child node for a split to be made).

For each fold, the data is fitted on the remaining four folds, and the validation precision is measured on the final fold that was left out. The validation performance is given as a mean of these five values. This mean precision is calculated for each set of parameters as above. The cross-validation performed here helps to prevent overfitting by ensuring the model does not tune to specific features of the images that aren't generalisable to all potential images that would be run through the model.

Precision is used in this analysis due to the cost of identifying an image as a sunset when it is not. I don't mind if an image of a sunset is shared on my personal Instagram account (false negative), however sharing an image of myself on my sunset page (false positive), which is public, could severely breach my privacy. I only care about false positives and use precision, which gives the proportion of correctly-classified sunset images to all images that are classified as sunsets, as the optimisation target.

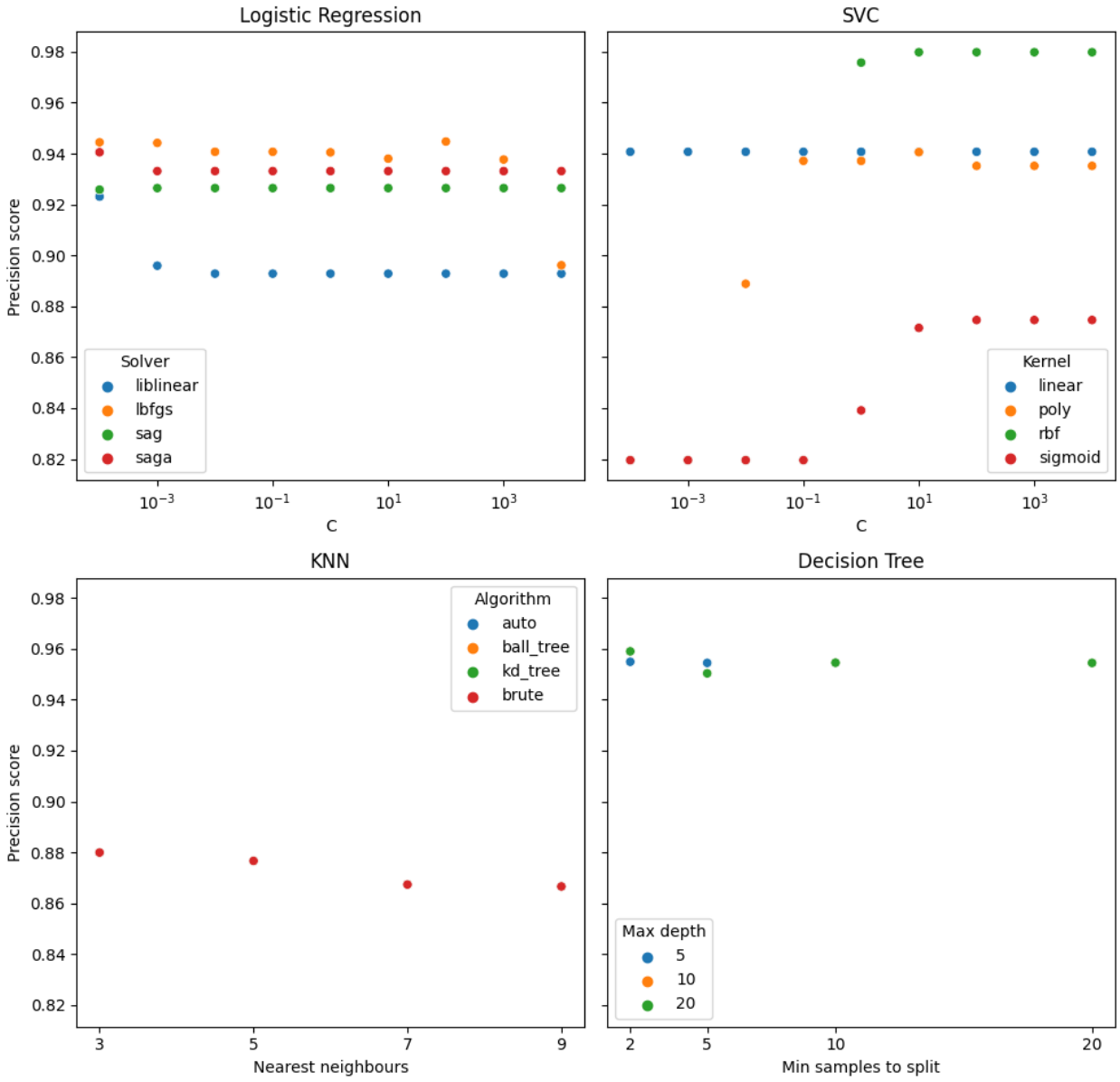
Validation Scores

Logistic Regression, SVC, KNN & Decision Tree

For these four models, the validation results are displayed in Figure 15. Here, we see that the best performance came from the support-vector classifier with a radial basis function and a C value of 10. The validation precision score for this model was **0.980**.

Figure 15.

Precision Scores for Different Models and Parameters with Augmented Data



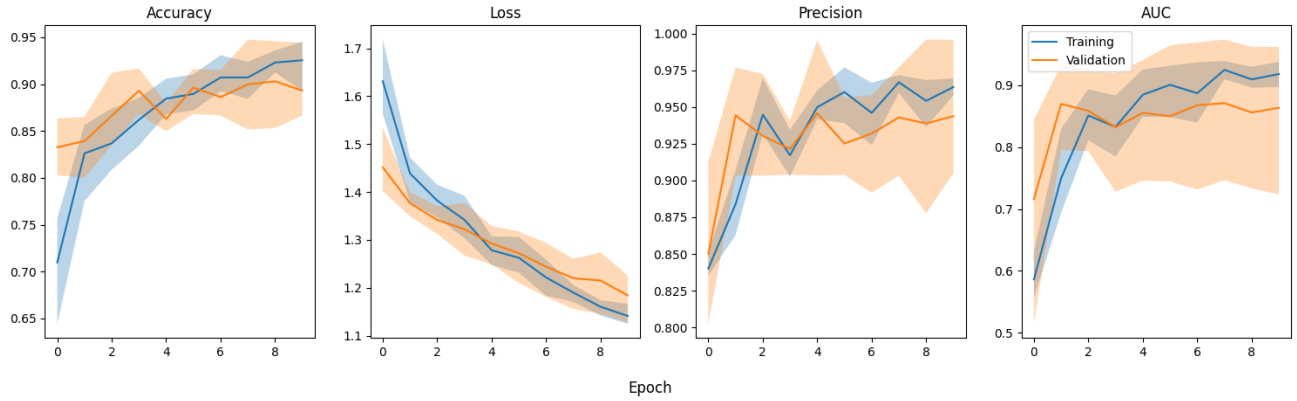
MobileNetV2

For the MobileNetV2 model, the validation and training scores and confidence intervals for four different metrics (accuracy, loss, precision, and AUC) are displayed in Figure 16 below. This shows that loss decreases across the epochs, and accuracy, precision, and AUC scores all

increase. This indicates that our model is working as expected. It also shows that we don't need to continue training the model (i.e. using a larger number of epochs), since that validation scores have already begun to plateau (they likely wouldn't improve if we increased the number of epochs). After 10 training epochs, we achieved a precision validation score of **0.944**.

Figure 16.

Training and Validation Metrics Over Training Process for MobileNetV2 Model with Augmented Data



Test Set Results

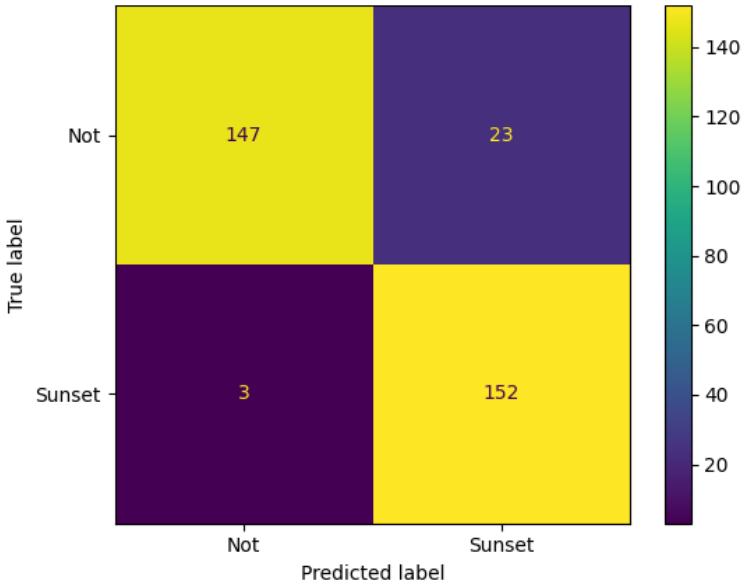
Given that the SVC (with radial basis function and C as 10) produced the best overall precision validation score, we will report metrics here based on this model. The results here are based only on the test set (data that was not used throughout training) and are as follows:

- Precision: 0.869.
- AUC: 0.923.
- Accuracy: 0.92.

A confusion matrix is displayed below (Figure 17) with the false positive and false negative classifications. Here, we see that there are 3 false negatives and 23 false positives out of a dataset of 325 images.

Figure 17.

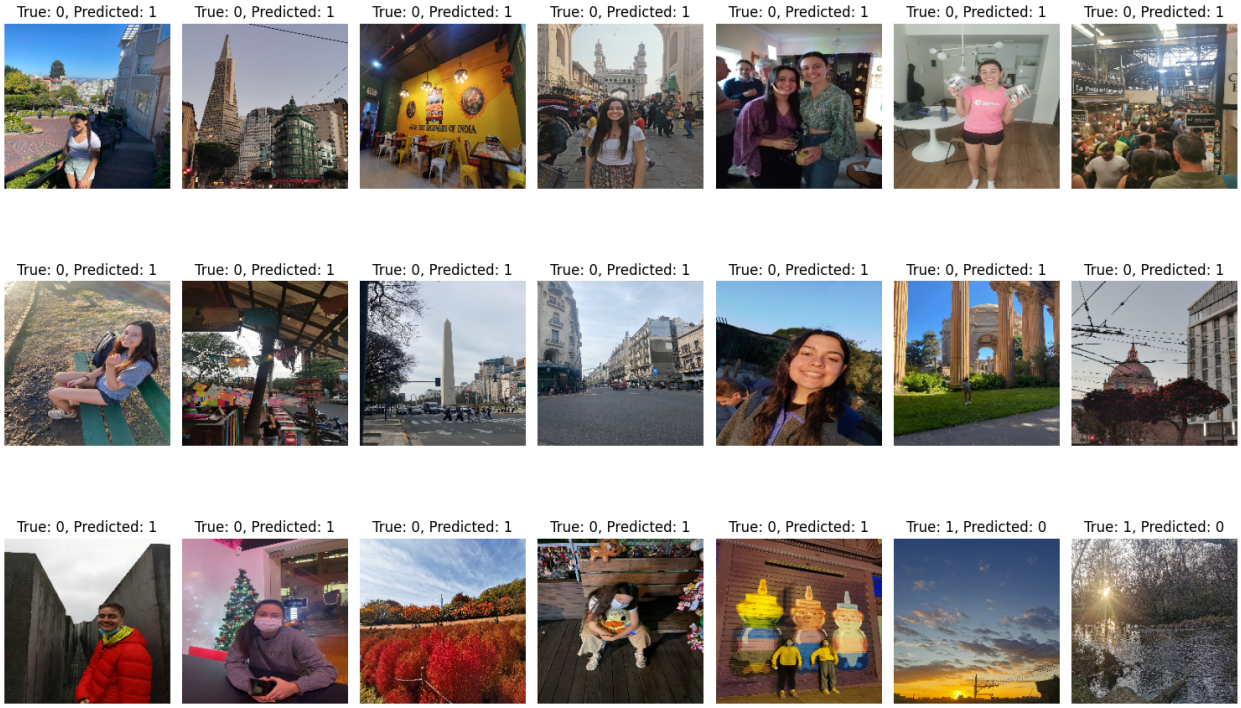
Confusion Matrix for Test Set Predictions with SVC Model



We will also look at the misclassifications made by the model in Figure 18. Here, we see that the majority of misclassifications are images that are actually non-sunsets that the model predicts as sunsets. This might be a cause of the large amount of light in lots of the images. In many of them, there are patches of light (some artificial and some from the sun) which could lead the classifier to believe they are a sunset.

Figure 18.

Misclassifications from SVC Model on Test Set



AI Tools Declaration: No AI Tools were used in this assignment.

References

Acharya, H. (2017, June 9). Why my image is different being plotted in Opencv-Python? Stack Overflow. https://stackoverflow.com/questions/44447957/why-my-image-is-different-being-plotted-in-opencv-python

Brownlee, J. (2019, June 16). A Gentle Introduction to Generative Adversarial Networks (GANs) - MachineLearningMastery.com. MachineLearningMastery.com. https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

Kumar, S. (2021, October 30). How to tune multiple ML models with GridSearchCV at once? Medium; Towards Data Science. https://towardsdatascience.com/how-to-tune-multiple-ml-models-with-gridsearchcv-at-once-9fcebfcc6c23

Luk, K. (2019, March 30). What libraries can load image in Python and what are their difference? Medium; Towards Data Science. https://towardsdatascience.com/what-library-can-load-image-in-python-and-what-are-their-difference-d1628c6623ad

Rocca, J. (2019, September 24). Understanding Variational Autoencoders (VAEs) - Towards Data Science. Medium; Towards Data Science. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73

Scikit-Learn. (2014). sklearn.linear_model.LogisticRegression. Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

Scikit-Learn. (2023a). 1.4. Support Vector Machines. Scikit-Learn. https://scikit-learn.org/stable/modules/svm.html

Scikit-Learn. (2023b). sklearn.neighbors.KNeighborsClassifier. Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Scikit-Learn. (2023c). sklearn.svm.SVC. Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

Scitkit-Learn. (2023). 1.10. Decision Trees. Scikit-Learn. https://scikit-learn.org/stable/modules/tree.html#:~:text=Decision%20Trees%20(DTs)%20are%20a,as%20a%20piecewise%20constant%20approximation.Tensorflow Authors. (2019).

Google Colaboratory. Colab.research.google.com. https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/images/transfer_learning.ipynb?force_kitty_mode=1&force_corgi_mode=1#scrollTo=77gENRVX40S7

Visually Explained. (2021). Support Vector Machine (SVM) in 2 minutes [YouTube Video]. In YouTube. https://www.youtube.com/watch?v=_YPScrckx28&ab_channel=VisuallyExplained

Code Appendix

0.1 Data

[1]:
```
from google.colab import drive
drive.mount("/content/drive/")
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call
drive.mount("/content/drive/", force_remount=True).
```

[2]:
```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt

def get_images_from_file(paths, show_idxs = [], resize = 32):
    """
    Function retrieves images from local file paths, resizes the images, and␣
 ↪flattens them into a 1-dimensional
    array. The resized images can be displayed using the show_idxs attribute.

    Parameters
    ----------
    paths: lst
        List of paths of local files containing images to be read.
    show_idxs: lst
        List of index locations of the resized images to be displayed.
    resize: int
        The pixel dimensions used to resize all images.

    Returns
    -------
    X: lst
        List of numpy arrays of image attributes (integers on scale [0, 256]␣
 ↪representing RGB colour intensities for
        each pixel) taken from resized flattened images.
    y: lst
        List of labels from 0 to len(paths)-1. In the case of binary␣
 ↪classification, this is a list of 0s and 1s
        corresponding to the locations of the flattened images in X.
```

```
    """

    X = []
    y = []

    # loops over the paths then filenames where data is located
    for i, path in enumerate(paths):
        for j, filename in enumerate(os.listdir(path)):

            # skip over filenames that are not images
            if filename == '.ipynb_checkpoints':
                continue

            # read and resize images
            img = cv2.imread(os.path.join(path, filename))
            if img is None:
                continue
            resized_img = cv2.resize(img, dsize = (resize, resize))

            # display a portion of loaded resized images
            if j in show_idxs:

                # change colour order of images from OpenCV (BGR) -> matplotlib␣
 ↪(RBG)
                rgb_img = cv2.cvtColor(resized_img, cv2.COLOR_BGR2RGB)
                plt.matshow(rgb_img)
                plt.show()

            X.append(resized_img)
            y.append(i)

    return X, y
```

[3]:
```
# load training data
path_train_not_sunsets = '/content/drive/MyDrive/CS156_Assignment/train/
 ↪not_sunsets'
path_train_sunsets = '/content/drive/MyDrive/CS156_Assignment/train/sunsets'

X_train, y_train = get_images_from_file([path_train_not_sunsets,␣
 ↪path_train_sunsets])
```

[4]:
```
# load testing data
path_test_not_sunsets = '/content/drive/MyDrive/CS156_Assignment/test/
 ↪not_sunsets'
path_test_sunsets = '/content/drive/MyDrive/CS156_Assignment/test/sunsets'
```

```
X_test, y_test = get_images_from_file([path_test_not_sunsets,␣
 ↪path_test_sunsets])
```

0.2 Aumentation: GAN

[5]:
```
# extract only sunset images
x_train = np.array(X_train)[np.array(y_train) == 1]
x_test = np.array(X_test)[np.array(y_test) == 1]

# combine all sunset images
x_train_concatenated = np.concatenate((x_train, x_test), axis = 0)
x_train_floats = x_train_concatenated.astype(np.float32)

# normalise to be in range [-1, 1]
x_train_norm = (x_train_floats - 127.5) / 127.5
```

[6]:
```
from keras.models import Sequential, Model
from keras.layers import Dense, LeakyReLU, BatchNormalization, Input, Flatten,␣
 ↪Reshape
from keras.optimizers import Adam
from keras import initializers

latent_dim = 100

def build_generator():
    """
    Build a generator model for a Generative Adversarial Network (GAN).

    Returns
    --------
    Model(noise, img)
        A Keras `Model` object.
    """
    model = Sequential()
    model.add(Dense(128, input_dim=latent_dim))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(256))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod((32, 32, 3)), activation='tanh'))
    model.add(Reshape((32, 32, 3)))

    noise = Input(shape=(latent_dim,))
```

```
    img = model(noise)

    return Model(noise, img)

def build_discriminator():
    """
    Build a discriminator model for a Generative Adversarial Network (GAN).

    Returns
    --------
    Model(img, validity)
        A Keras `Model` object.
    """
    model = Sequential()
    model.add(Flatten(input_shape=(32, 32, 3)))
    model.add(Dense(512))
    model.add(LeakyReLU(0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(0.2))
    model.add(Dense(1, activation='sigmoid'))

    img = Input(shape=(32, 32, 3))
    validity = model(img)

    return Model(img, validity)

# build discriminator
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0001, 0.5),␣
 ↪metrics=['accuracy'])

# build generator
generator = build_generator()
z = Input(shape=(latent_dim,))
img = generator(z)

discriminator.trainable = False
validity = discriminator(img)

# combine model
combined = Model(z, validity)
combined.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```

[7]:
```
import tensorflow as tf
from scipy.linalg import sqrtm

def preprocess_images(images):
```

```
    """
    Preprocess images by resizing them and converting them to an acceptable␣
 ↪format for
    inception_v3.

    Parameters
    ----------
    images: lst
        List of images to be preprocessed.

    Returns
    -------
    images: lst
        The images that have been preprocessed.
    """
    images = tf.image.resize(images, (299, 299))
    images = tf.keras.applications.inception_v3.preprocess_input(images)
    return images

def calculate_fid(real_images, generated_images):
    """
    Calculates the Fréchet Inception Distance (FID) between two sets of images.
    The FID is a measure of the similarity between the distributions of␣
 ↪features
    extracted from the Inception V3 model for the real and generated images.

    Parameters
    ----------
    real_images: tensor
        A tensor of shape `(n_samples, height, width, channels)` representing␣
 ↪the real images.
    generated_images: tensor
        A tensor of shape `(n_samples, height, width, channels)` representing␣
 ↪the generated images.

    Returns
    -------
    fid: float
        The calculated FID score.
    """
    real_images = preprocess_images(real_images)
    generated_images = preprocess_images(generated_images)

    inception_model = tf.keras.applications.InceptionV3(include_top=False,␣
 ↪pooling='avg', input_shape=(299, 299, 3))

    # extract the feature activations for the real and generated images
```

```
    real_activations = inception_model.predict(real_images, verbose = 0)
    generated_activations = inception_model.predict(generated_images, verbose =␣
↪0)

    # calculate the mean and covariance of the feature activations for the real␣
↪and generated images
    mu1, sigma1 = real_activations.mean(axis=0), np.cov(real_activations,␣
↪rowvar=False)
    mu2, sigma2 = generated_activations.mean(axis=0), np.
↪cov(generated_activations, rowvar=False)

    ssd = np.sum((mu1 - mu2)**2)
    cov_mean = sqrtm(sigma1.dot(sigma2))

    # if the covariance is complex, take only the real part
    if np.iscomplexobj(cov_mean):
        cov_mean = cov_mean.real

    fid = ssd + np.trace(sigma1 + sigma2 - 2 * cov_mean)
    return fid
```

[8]:
```
def train(X_train, epochs, batch_size, progress_interval, save_interval):
    """
    Trains a generative adversarial network.

    Parameters
    ----------
    X_train : np.ndarray
        The training data, a numpy array of shape (num_samples, height, width,␣
↪channels).
    epochs : int
        The number of epochs to train the model for.
    batch_size : int
        The batch size used during training.
    progress_interval : int
        The interval at which to print progress updates.
    save_interval : int
        The interval at which to calculate FID scores and save generated images.

    Returns
    -------
    d_losses : lst
        The discriminator loss at each epoch.
    d_accuracies : lst
        The discriminator accuracy at each epoch.
    g_losses : lst
        The generator loss at each epoch.
```

```
    fid_scores : lst
        The FID score at each epoch when `save_interval` is reached.
    """
    half_batch = int(batch_size / 2)
    d_losses = []
    d_accuracies = []
    g_losses = []
    fid_scores = []

    for epoch in range(epochs):
        # train discriminator
        idx = np.random.randint(0, X_train.shape[0], half_batch)
        imgs = X_train[idx]

        noise = np.random.normal(0, 1, (half_batch, latent_dim))
        gen_imgs = generator.predict(noise, verbose = 0)

        # calculate disciminator loss
        d_loss_real = discriminator.train_on_batch(imgs, np.ones((half_batch,␣
↪1)))
        d_loss_fake = discriminator.train_on_batch(gen_imgs, np.
↪zeros((half_batch, 1)))
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # train generator
        noise = np.random.normal(0, 1, (batch_size, latent_dim))
        valid_y = np.array([1] * batch_size)

        # calculate generator loss
        g_loss = combined.train_on_batch(noise, valid_y)

        # print progress
        if epoch % progress_interval == 0:
            print(f"Epoch {epoch}: [D loss: {d_loss[0]}, acc.: {100 *␣
↪d_loss[1]}%] [G loss: {g_loss}]")
            d_losses.append(d_loss[0])
            d_accuracies.append(d_loss[1])
            g_losses.append(g_loss)

        # calculate FID and Inception scores every 500 epochs
        if epoch % save_interval == 0:
            n_samples = 100
            noise = np.random.normal(0, 1, (n_samples, latent_dim))
            gen_imgs = generator.predict(noise, verbose = 0)
            gen_imgs = 0.5 * gen_imgs + 0.5

            fid_score = calculate_fid(X_train[:n_samples], gen_imgs)
```

```
            print(f"Epoch {epoch}: FID score: {fid_score}")
            fid_scores.append(fid_score)

            save_imgs(epoch)

    return d_losses, d_accuracies, g_losses, fid_scores


def save_imgs(epoch):
    """
    Saves generated images to disk.

    Parameters
    ----------
    epoch : int
        The current epoch number.
    """

    # generate 25 images
    r, c = 5, 5
    noise = np.random.normal(0, 1, (r * c, latent_dim))
    gen_imgs = generator.predict(noise, verbose = 0)

    # rescale images from [-1, 1] to [0, 1]
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(r, c)
    count = 0
    for i in range(r):
        for j in range(c):

            # convert colours from CV2 to matplotlib format
            axs[i, j].imshow(cv2.cvtColor(gen_imgs[count, :, :, :], cv2.
 ↪COLOR_BGR2RGB))
            axs[i, j].axis('off')
            count += 1
    plt.savefig(f"sunsets_GAN_{epoch}.png")
    plt.close()
```

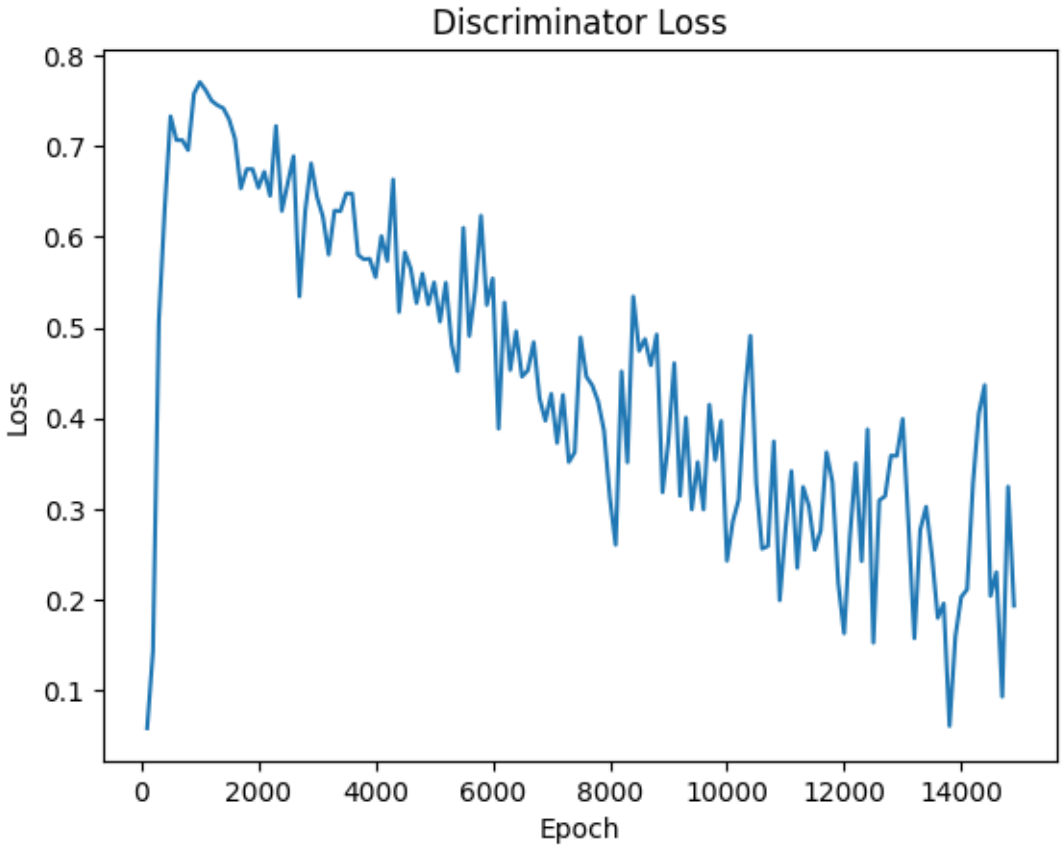
[9]:
```
progress_interval = 100
save_interval = 500
d_losses, d_accuracies, g_losses, fid_scores = train(x_train_norm,␣
 ↪epochs=15000, batch_size=32, progress_interval = progress_interval,␣
 ↪save_interval=save_interval)
```

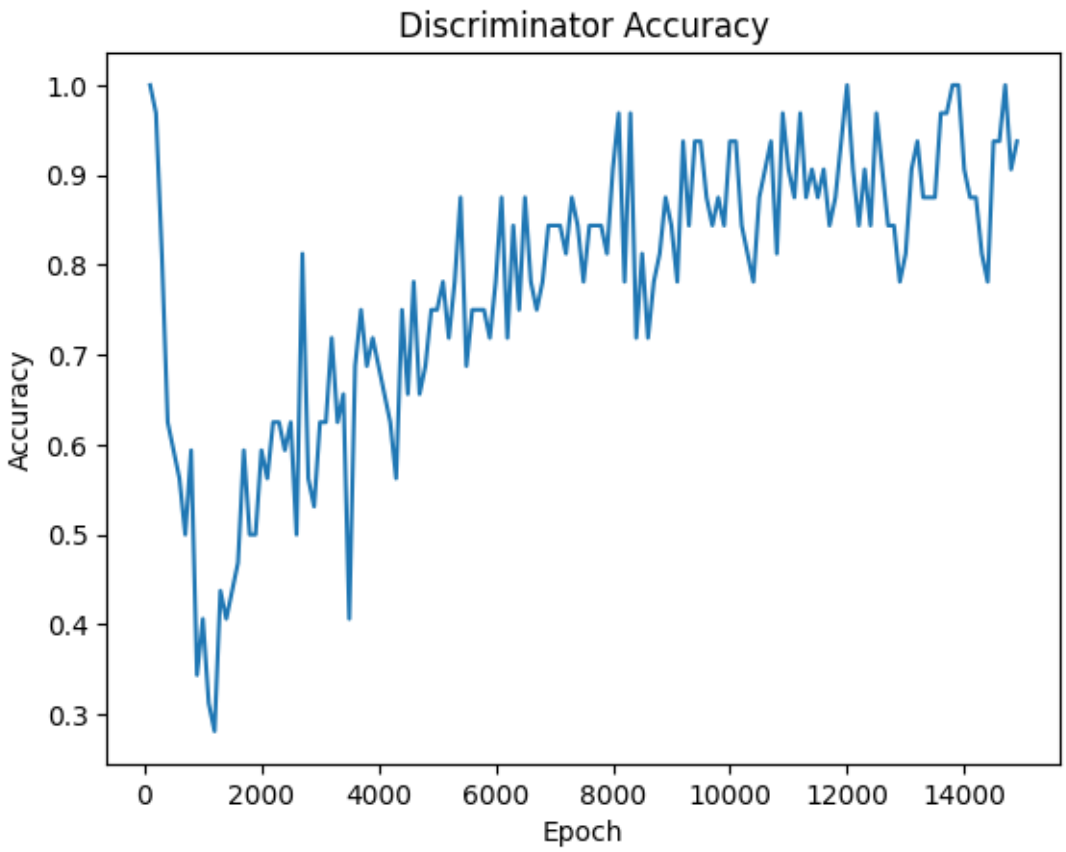
```
Epoch 0: [D loss: 0.6676372587680817, acc.: 37.5%] [G loss: 0.4661162495613098]
Epoch 0: FID score: 5.489790522869095
Epoch 100: [D loss: 0.058498816564679146, acc.: 100.0%] [G loss:
```

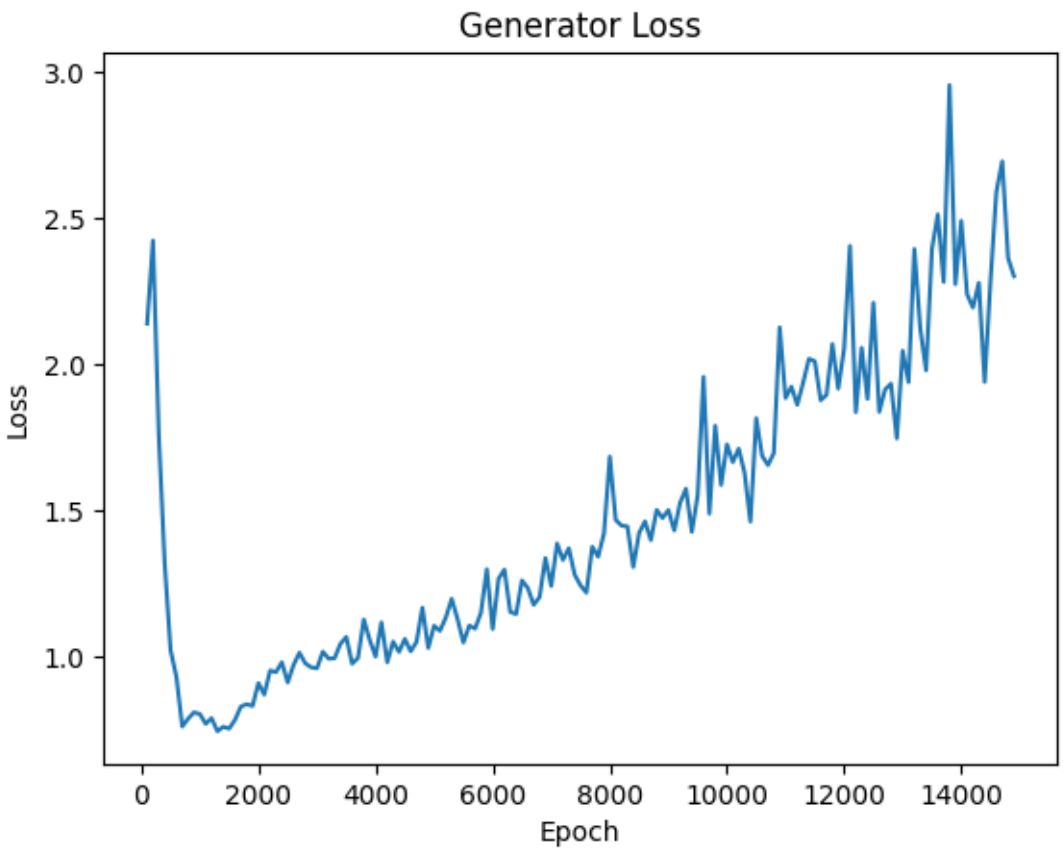
```
Epoch 13400: [D loss: 0.3025885969400406, acc.: 87.5%] [G loss:
1.978806495666504]
Epoch 13500: [D loss: 0.24807336926460266, acc.: 87.5%] [G loss:
2.394073486328125]
Epoch 13500: FID score: 2.1634148793715964
Epoch 13600: [D loss: 0.17997442185878754, acc.: 96.875%] [G loss:
2.5121002197265625]
Epoch 13700: [D loss: 0.19597551226615906, acc.: 96.875%] [G loss:
2.280881404876709]
Epoch 13800: [D loss: 0.0609592255204916, acc.: 100.0%] [G loss:
2.9540042877197266]
Epoch 13900: [D loss: 0.15981625765562057, acc.: 100.0%] [G loss:
2.273376941680908]
Epoch 14000: [D loss: 0.2031320482492447, acc.: 90.625%] [G loss:
2.4903244972229004]
Epoch 14000: FID score: 2.2602656678574826
Epoch 14100: [D loss: 0.21145174652338028, acc.: 87.5%] [G loss:
2.2387046813964844]
Epoch 14200: [D loss: 0.327826663851738, acc.: 87.5%] [G loss:
2.193573474884033]
Epoch 14300: [D loss: 0.405260905623436, acc.: 81.25%] [G loss:
2.2779955863952637]
Epoch 14400: [D loss: 0.4365740567445755, acc.: 78.125%] [G loss:
1.9393210411071777]
Epoch 14500: [D loss: 0.20433222502470016, acc.: 93.75%] [G loss:
2.287503242492676]
Epoch 14500: FID score: 2.315854960992314
Epoch 14600: [D loss: 0.2304304763674736, acc.: 93.75%] [G loss:
2.589226007461548]
Epoch 14700: [D loss: 0.09333259053528309, acc.: 100.0%] [G loss:
2.693509101867676]
Epoch 14800: [D loss: 0.3246871083974838, acc.: 90.625%] [G loss:
2.3630785942077637]
Epoch 14900: [D loss: 0.19354227930307388, acc.: 93.75%] [G loss:
2.3007712364196777]
```

[10]:
```
# plot resulting metrics
plt.plot(np.arange(progress_interval, len(d_losses)*progress_interval,␣
 ↪progress_interval), d_losses[1:])
plt.title('Discriminator Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

plt.plot(np.arange(progress_interval, len(d_accuracies)*progress_interval,␣
 ↪progress_interval), d_accuracies[1:])
plt.title('Discriminator Accuracy')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()

plt.plot(np.arange(progress_interval, len(g_losses)*progress_interval,␣
 ↪progress_interval), g_losses[1:])
plt.title('Generator Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()

plt.plot(np.arange(0, len(fid_scores)*save_interval, save_interval), fid_scores)
plt.title('FID Score')
plt.xlabel('Epoch')
plt.ylabel('FID score')
plt.show()
```




Discriminator Accuracy
Accuracy
Epoch
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
2000
4000
6000
8000
10000
12000
14000


Generator Loss
3.0
2.5
2.0
1.5
1.0
Loss
0
2000
4000
6000
8000
10000
12000
14000
Epoch

0.3 Augmentation: VAE

[11]:
```
# extract only sunset images
x_train = np.array(X_train)[np.array(y_train) == 1]
x_test = np.array(X_test)[np.array(y_test) == 1]

# normalize the images to be in the range [0, 1]
x_train, x_test = np.array(x_train) / 255.0, np.array(x_test) / 255.0

# flatten the images from (28, 28, 3) to (2352,)
x_train = np.array(x_train).reshape(-1, 32 * 32 * 3)
x_test = np.array(x_test).reshape(-1, 32 * 32 * 3)
```

[12]:
```
import tensorflow.keras.metrics as metrics

class KLDivergenceMetric(metrics.Metric):
    """
    Computes the Kullback-Leibler divergence metric.
```

```
    Attributes
    ----------
    kl_divergence : tf.Variable
        The accumulated KL divergence.
    num_samples : tf.Variable
        The total number of samples.

    Methods
    -------
    update_state(y_true, y_pred, sample_weight=None)
        Accumulates KL divergence from a batch of data.
    result()
        Computes the final KL divergence.
    reset_state()
        Resets accumulated KL divergence and number of samples.
    """
    def __init__(self, name='kl_divergence', **kwargs):
        super(KLDivergenceMetric, self).__init__(name=name, **kwargs)
        self.kl_divergence = self.add_weight(name='kl_div', initializer='zeros')
        self.num_samples = self.add_weight(name='num_samples',␣
↪initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        """
        Accumulates KL divergence from a batch of data.

        Parameters
        ----------
        y_true : tf.Tensor
            The true values.
        y_pred : tf.Tensor
            The predicted values.
        sample_weight : tf.Tensor
            Sample weights.
        """
        z_mean, z_log_var = encoder(y_true)[:2]

        # calculate KL divergence/loss
        kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
        kl_loss = K.sum(kl_loss, axis=-1)
        kl_loss *= -0.5
        num_samples_in_batch = K.cast(K.shape(y_true)[0], dtype='float32')
        self.kl_divergence.assign_add(K.sum(kl_loss))
        self.num_samples.assign_add(num_samples_in_batch)

    def result(self):
        """
```

```
        Computes the final KL divergence.

        Returns
        -------
        float
            The KL divergence.
        """
        return self.kl_divergence / self.num_samples

    def reset_state(self):
        """
        Resets accumulated KL divergence and number of samples.
        """
        self.kl_divergence.assign(0.0)
        self.num_samples.assign(0.0)
```

[13]:
```
class VisualizeExamples(tf.keras.callbacks.Callback):
    """
    Callback that generates and saves images of examples to visualise.

    Attributes
    ----------
    decoder : tf.keras.Model
        The decoder model used to generate images.
    examples_to_visualize : tf.Tensor
        A tensor containing examples to visualize.

    Methods
    -------
    on_epoch_end(epoch, logs=None)
        Generates and saves images of examples to visualise at the end of each␣
↪epoch.
    """
    def __init__(self, decoder, examples_to_visualize):
        super(VisualizeExamples, self).__init__()
        self.decoder = decoder
        self.examples_to_visualize = examples_to_visualize

    def on_epoch_end(self, epoch, logs=None):
        """
        Generates and saves images of examples to visualise at the end of each␣
↪epoch.

        Parameters
        ----------
        epoch : int
            The current epoch number.
```

```
        logs : dict
            The dictionary of logs.
        """

        if (epoch+1)%500 == 0:
            # generate images from the examples
            generated_images = self.decoder.predict(self.examples_to_visualize)

            # save the generated images
            n = len(self.examples_to_visualize)
            plt.figure(figsize=(2 * n, 2))
            for i in range(n):
                ax = plt.subplot(1, n, i + 1)

                # convert colours from cv to matplotlib to visualise
                plt.imshow(cv2.cvtColor(generated_images[i].reshape(32, 32, 3),␣
↪cv2.COLOR_BGR2RGB))
                plt.axis('off')
            plt.savefig(f'sunsets_{epoch+1}.png')
            plt.close()
```

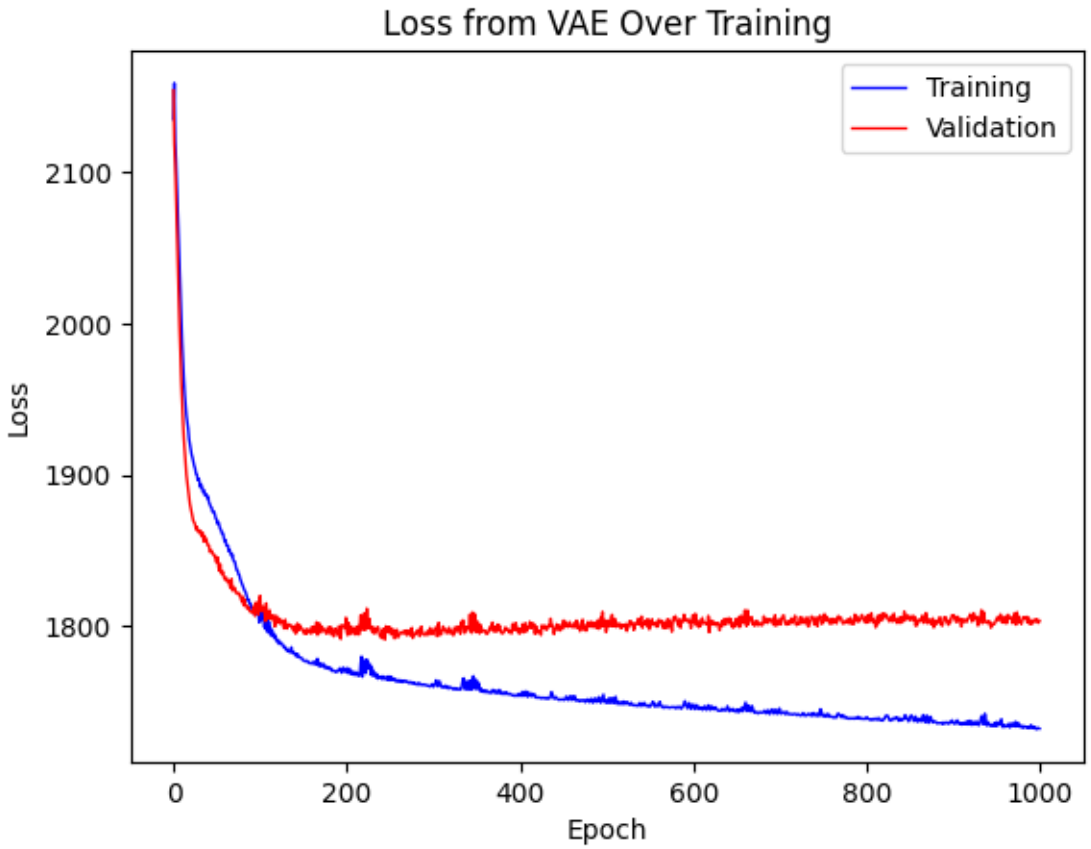
[14]:
```
from tensorflow.keras.layers import Lambda
from tensorflow.keras.losses import binary_crossentropy
from tensorflow.keras import backend as K

# define VAE architecture
latent_dim = 2

def sampling(args):
    """
    Uses the reparameterisation trick to sample from the latent space.

    Parameters
    ----------
    args : tuple of tensors
        The mean and log variance of the latent distribution.

    Returns
    -------
    tensor
        The sampled tensor from the latent space.
    """
    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon
```

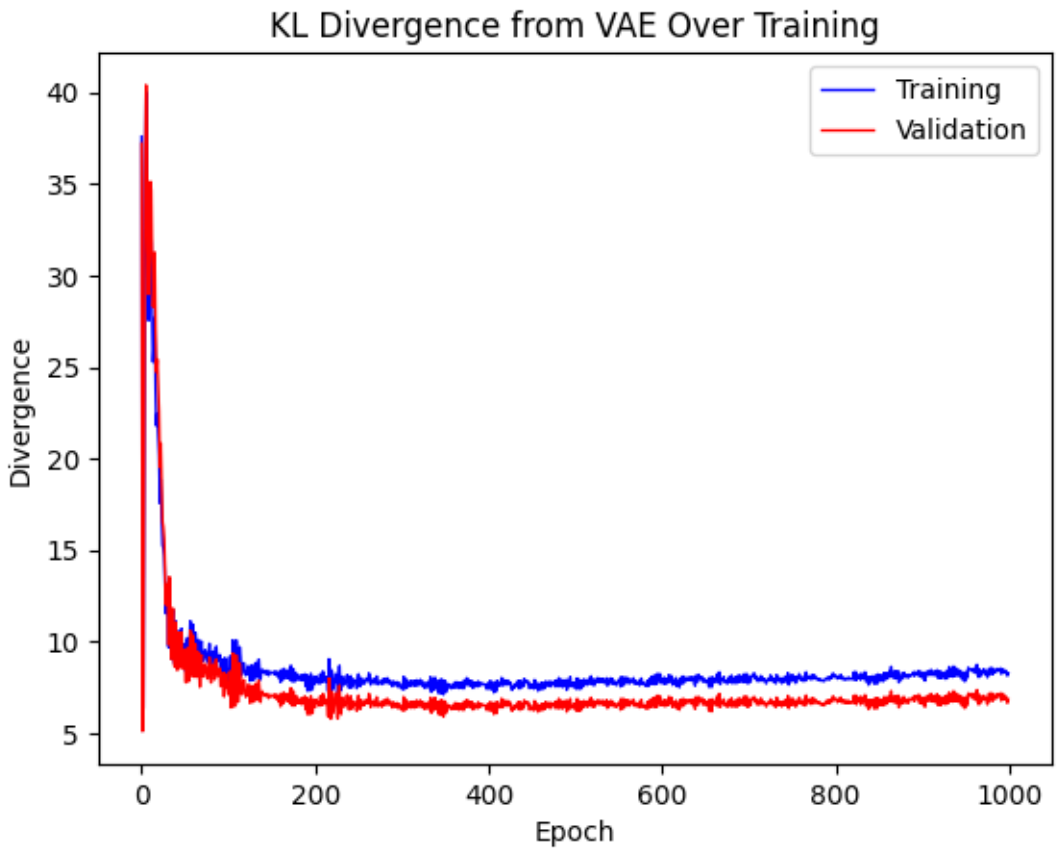
```
# encoder
inputs = Input(shape=(32 * 32 * 3,))
h_enc = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.
 ↪l2(0.01))(inputs)
z_mean = Dense(latent_dim)(h_enc)
z_log_var = Dense(latent_dim)(h_enc)
z = Lambda(sampling, output_shape=(latent_dim,))([z_mean, z_log_var])
encoder = Model(inputs, [z_mean, z_log_var, z])

# decoder
latent_inputs = Input(shape=(latent_dim,))
h_dec = Dense(256, activation='relu', kernel_regularizer=tf.keras.regularizers.
 ↪l2(0.01))(latent_inputs)
outputs = Dense(3072, activation='sigmoid')(h_dec)
decoder = Model(latent_inputs, outputs)

# VAE
outputs = decoder(encoder(inputs)[2])
vae = Model(inputs, outputs)

# VAE loss (binary cross entropy)
reconstruction_loss = binary_crossentropy(inputs, outputs) * 3072
kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)

vae.add_loss(vae_loss)
vae.compile(optimizer='adam', metrics=[KLDivergenceMetric()])

# set a random seed for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# choose some random examples from the latent space
examples_to_visualize = np.random.normal(size=(10, latent_dim))

# create a callback instance with the chosen examples
visualize_examples_callback = VisualizeExamples(decoder, examples_to_visualize)

# train the VAE with the custom callback
history = vae.fit(x_train, x_train,
        epochs=1000,
        batch_size=100,
        validation_data=(x_test, x_test),
        callbacks=[visualize_examples_callback])
```

```
kl_divergence: 8.2433 - val_loss: 1801.8145 - val_kl_divergence: 6.7812
Epoch 993/1000
1/1 [==============================] - 0s 160ms/step - loss: 1733.6588 -
kl_divergence: 8.3904 - val_loss: 1802.3340 - val_kl_divergence: 7.0335
Epoch 994/1000
1/1 [==============================] - 0s 158ms/step - loss: 1732.2109 -
kl_divergence: 8.4999 - val_loss: 1803.4849 - val_kl_divergence: 7.1563
Epoch 995/1000
1/1 [==============================] - 0s 140ms/step - loss: 1732.6957 -
kl_divergence: 8.3005 - val_loss: 1804.2671 - val_kl_divergence: 6.8402
Epoch 996/1000
1/1 [==============================] - 0s 138ms/step - loss: 1732.4459 -
kl_divergence: 8.3743 - val_loss: 1803.2891 - val_kl_divergence: 6.9168
Epoch 997/1000
1/1 [==============================] - 0s 174ms/step - loss: 1731.7892 -
kl_divergence: 8.4153 - val_loss: 1804.5054 - val_kl_divergence: 7.0171
Epoch 998/1000
1/1 [==============================] - 0s 138ms/step - loss: 1732.6812 -
kl_divergence: 8.3615 - val_loss: 1802.1843 - val_kl_divergence: 6.8990
Epoch 999/1000
1/1 [==============================] - 0s 173ms/step - loss: 1732.4518 -
kl_divergence: 8.1431 - val_loss: 1802.1783 - val_kl_divergence: 6.6529
Epoch 1000/1000
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 1s 514ms/step - loss: 1732.2384 -
kl_divergence: 8.2686 - val_loss: 1803.5377 - val_kl_divergence: 6.8567
```

[15]:
```
# plot metrics over training
training_loss = history.history['loss']
validation_loss = history.history['val_loss']
kl_divergence = history.history['kl_divergence']
val_kl_divergence = history.history['val_kl_divergence']

plt.plot(range(len(training_loss)), training_loss, color = 'blue', label =␣
 ↪'Training', linewidth = 1)
plt.plot(range(len(validation_loss)), validation_loss, color = 'red', label =␣
 ↪'Validation', linewidth = 1)
plt.title('Loss from VAE Over Training')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(range(len(kl_divergence)), kl_divergence, color = 'blue', label =␣
 ↪'Training', linewidth = 1)
plt.plot(range(len(val_kl_divergence)), val_kl_divergence, color = 'red', label␣
 ↪= 'Validation', linewidth = 1)
```

```
plt.title('KL Divergence from VAE Over Training')
plt.xlabel('Epoch')
plt.ylabel('Divergence')
plt.legend()
plt.show()
```





[16]:
```
# generate new images
num_images = 150
latent_vectors = np.random.normal(size=(num_images, latent_dim))

# feed latent vectors into decoder
generated_images = decoder.predict(latent_vectors)
generated_images = generated_images.reshape((-1, 32, 32, 3))

# append the new images to the training set
augmented_X_train = np.concatenate((X_train, generated_images))
augmented_y_train = np.concatenate((y_train, [1]*num_images))
```

```
5/5 [==============================] - 0s 5ms/step
```

0.4 MobileNetV2

[17]:
```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Dropout, GlobalAveragePooling2D
from tensorflow.keras import metrics
from sklearn.model_selection import KFold
```

```
from tensorflow.keras import regularizers

def resize_images(images, size):
    """
    Resizes a batch of images to a specified size.

    Parameters
    ----------
    images : array
        The batch of images to resize.
    size : int
        The size to resize the images to.

    Returns
    -------
    array
        The resized batch of images.
    """
    return np.array([cv2.resize(img, (size, size)) for img in images])

def mobilenet_cv(X_train, y_train, k = 5, lr = 0.0001):
    """
    Perform K-Fold Cross-Validation on a MobileNetV2 model.

    Parameters
    ----------
    X_train : lst
        The training input data.
    y_train : lst
        The target training data.
    k : int, optional
        The number of folds to use for cross-validation. Defaults to 5.
    lr : float, optional
        The learning rate of the new model layers. Defaults to 0.0001.

    Returns
    -------
    tuple of list of floats
        A tuple of metric scores for each fold of the cross-validation.
        The first element is a list of accuracy scores.
        The second element is a list of validation accuracy scores.
        The third element is a list of loss scores.
        The fourth element is a list of validation loss scores.
        The fifth element is a list of precision scores.
        The sixth element is a list of validation precision scores.
        The seventh element is a list of AUC scores.
        The eighth element is a list of validation AUC scores.
```

```
    """
    tf.random.set_seed(42)
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    # reshape the entire dataset first
    X_train_reshaped = resize_images(X_train, 224)

    acc, val_acc, loss, val_loss, prec, val_prec, auc, val_auc = [], [], [],␣
↪[], [], [], [], []

    for fold, (train_indices, val_indices) in enumerate(kf.split(X_train)):

        # split data into train and validation sets
        x_train_fold = X_train_reshaped[train_indices]
        y_train_fold = np.array(y_train)[train_indices]
        x_val_fold = X_train_reshaped[val_indices]
        y_val_fold = np.array(y_train)[val_indices]

        # create a new model with added layers
        base_model = MobileNetV2(input_shape=(224, 224, 3), include_top=False,␣
↪weights='imagenet')
        x = base_model.output
        x = GlobalAveragePooling2D()(x)
        x = Dense(1024, activation='relu', kernel_regularizer=regularizers.l2(0.
↪001))(x)
        x = Dropout(0.5)(x)

        # added dimensionality reduction
        predictions = Dense(1)(x)
        model = Model(inputs=base_model.input, outputs=predictions)

        # freeze the base model layers
        for layer in base_model.layers:
            layer.trainable = False

        # compile the model
        model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                      loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
                      metrics=[metrics.BinaryAccuracy(name='accuracy'),
                          metrics.Precision(name='precision'),
                          metrics.AUC(name='auc')])

        # train the model on the current fold
        history = model.fit(x_train_fold, y_train_fold, epochs=10,␣
↪batch_size=32,
                            validation_data=(x_val_fold, y_val_fold), verbose =␣
↪0)
```

```
        # record metric scores
        acc.append(history.history['accuracy'])
        val_acc.append(history.history['val_accuracy'])
        loss.append(history.history['loss'])
        val_loss.append(history.history['val_loss'])
        prec.append(history.history['precision'])
        val_prec.append(history.history['val_precision'])
        auc.append(history.history['auc'])
        val_auc.append(history.history['val_auc'])

    return model, acc, val_acc, loss, val_loss, prec, val_prec, auc, val_auc
```

[18]:
```
def plot_mobilenet_cv(acc, val_acc, loss, val_loss, prec, val_prec, auc,␣
↪val_auc):
    """
    Plot training and validation metrics for a MobileNet model trained with␣
↪cross-validation.
    This function creates a 1x4 grid of subplots, where each subplot␣
↪corresponds to one of the metrics
    (accuracy, loss, precision, and AUC). For each metric, the function plots␣
↪the mean score across
    the K cross-validation folds for both training and validation, along with␣
↪the 95% confidence
    interval obtained by computing the 2.5th and 97.5th percentiles of the␣
↪scores across folds at each epoch.
    The x-axis of the plot corresponds to the number of training epochs (T),␣
↪while the y-axis corresponds
    to the value of the metric.

    Parameters
    ----------
    acc : lst
        Training accuracy scores for K cross-validation folds and T epochs.
    val_acc : lst
        Validation accuracy scores for K cross-validation folds and T epochs.
    loss : lst
        Training loss values for K cross-validation folds and T epochs.
    val_loss : lst
        Validation loss values for K cross-validation folds and T epochs.
    prec : lst
        Training precision scores for K cross-validation folds and T epochs.
    val_prec : lst
        Validation precision scores for K cross-validation folds and T epochs.
    auc : lst
        Training AUC scores for K cross-validation folds and T epochs.
```

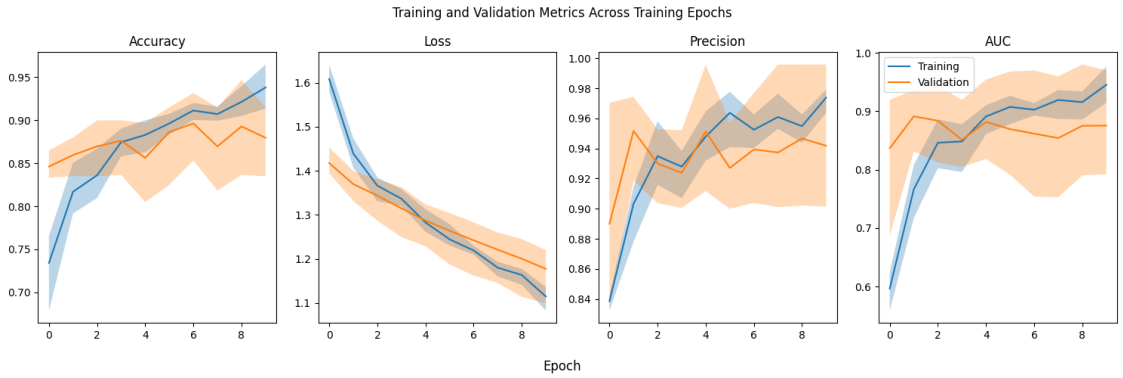
```
    val_auc : lst
        Validation AUC scores for K cross-validation folds and T epochs.

    Returns
    -------
    None
    """
    fig, ax = plt.subplots(1, 4, figsize = (15, 5))

    labels = ['Accuracy', 'Loss', 'Precision', 'AUC']
    for i, metric in enumerate([[acc, val_acc], [loss, val_loss], [prec,␣
 ↪val_prec], [auc, val_auc]]):
        ax[i].plot(np.mean(metric[0], axis = 0), label='Training')
        ax[i].fill_between(range(10), np.percentile(metric[0], 2.5, axis = 0),␣
 ↪np.percentile(metric[0], 97.5, axis = 0), alpha = 0.3)
        ax[i].plot(np.mean(metric[1], axis = 0), label='Validation')
        ax[i].fill_between(range(10), np.percentile(metric[1], 2.5, axis = 0),␣
 ↪np.percentile(metric[1], 97.5, axis = 0), alpha = 0.3)
        if i == 3:
            ax[i].legend()
        ax[i].set_title(labels[i])

    fig.suptitle('Training and Validation Metrics Across Training Epochs')
    fig.supxlabel('Epoch')
    fig.tight_layout()
    plt.show()
```

[19]:
```
cnn_model, acc, val_acc, loss, val_loss, prec, val_prec, auc, val_auc =␣
 ↪mobilenet_cv(augmented_X_train, augmented_y_train)
```

[20]:
```
print('Precision Validation Score:', round(np.mean(val_prec, axis = 0)[-1], 3))
print('Accuracy Validation Score:', round(np.mean(val_auc, axis = 0)[-1], 3))
print('AUC Validation Score:', round(np.mean(val_acc, axis = 0)[-1], 3))
```

```
Precision Validation Score: 0.942
Accuracy Validation Score: 0.875
AUC Validation Score: 0.88
```

[21]:
```
plot_mobilenet_cv(acc, val_acc, loss, val_loss, prec, val_prec, auc, val_auc)
```



0.5 Traditional ML Models: Logistic Regression, SVC, KNN & Decision Trees

[22]:
```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

def logreg_svc_knn_CV(X_train, y_train):
    """
    Completes grid search cross validation on logistic regression, KNN, SVC and␣
 ↪decision tree models with varying hyperparameters.
    For each of 5 folds, the model is fit on 4/5, the other being reserved for␣
 ↪measuring precision. GridSearchCV
    does this for all classifier/parameter combinations and returns an object␣
 ↪containing the results for each
    combination, including precisions for each fold of the CV.

    Parameters
    ----------
    X_train: lst
        Nested list of predictors for each observation. In the case of sunset␣
 ↪classification, this is a list of image
        attributes (integers on scale [0, 256] representing RGB colour␣
 ↪intensities for each pixel).
    y_train: lst
        List of labels, the targets of the classification (0: not sunset, 1:␣
 ↪sunset).

    Returns
    -------
    gs: GridSearchCV obj
```

```
        An object containing the results for each combination, including␣
↪precisions for each fold of the CV.
    """

    # number of folds for CV
    kfold = KFold(n_splits = 5, random_state = 123, shuffle = True)

    clf_logreg = LogisticRegression(random_state = 123, max_iter = 1000)
    clf_svc = SVC(random_state = 123)
    clf_knn = KNeighborsClassifier()
    clf_dt = DecisionTreeClassifier(random_state=123)

    # hyperparameters to optimise over for logistic regression
    param_logreg = {
        'classifier': [clf_logreg],
        'classifier__C': np.power(10.0, np.arange(-4, 5)),
        'classifier__solver': ['liblinear', 'lbfgs', 'sag', 'saga']
    }

    # hyperparameters to optimise over for SVC
    param_svc = {
        'classifier': [clf_svc],
        'classifier__C': np.power(10.0, np.arange(-4, 5)),
        'classifier__kernel': ['linear', 'poly', 'rbf', 'sigmoid']
    }

    # hyperparameters to optimise over for KNN
    param_knn = {
        'classifier': [clf_knn],
        'classifier__n_neighbors': [3, 5, 7, 9],
        'classifier__algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    }

    # hyperparameters to optimise over for decision tree
    param_dt = {
        'classifier': [clf_dt],
        'classifier__max_depth': [None, 5, 10, 20],
        'classifier__min_samples_split': [2, 5, 10, 20],
    }

    pipeline = Pipeline([('classifier', clf_logreg)])
    param_grid = [param_logreg, param_svc, param_knn, param_dt]

    # fit for different parameters & models, for each of k folds
    gs = GridSearchCV(pipeline, param_grid = param_grid, scoring = 'precision',␣
↪cv = kfold, n_jobs = -1, verbose = 3)
    gs.fit(X_train, y_train)
```

```
    print('Best parameters:', gs.best_params_)

    return gs
```

[23]:
```
num_images = augmented_X_train.shape[0]
augmented_X_train_reshaped = augmented_X_train.reshape(num_images, -1)
gs = logreg_svc_knn_CV(augmented_X_train_reshaped, augmented_y_train)
```

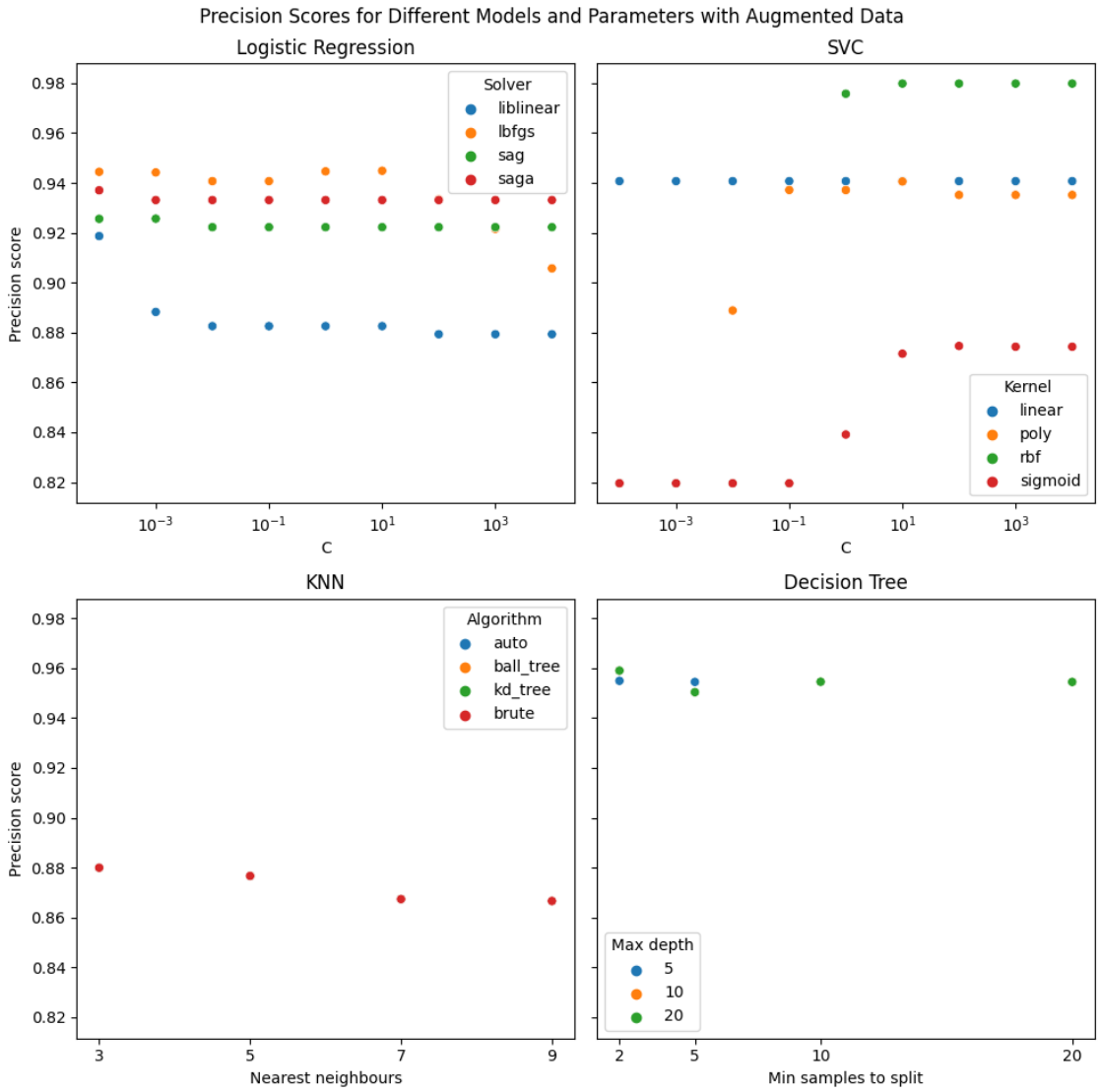
```
Fitting 5 folds for each of 104 candidates, totalling 520 fits
Best parameters: {'classifier': SVC(C=10.0, random_state=123), 'classifier__C':
10.0, 'classifier__kernel': 'rbf'}
```

[24]:
```
import seaborn as sns

def plot_gs_results(df_gs, params1, params2, params3, params4, model1, model2,␣
 ↪model3, model4, title):
    """
    Plots the results of precision scores from gridsearch dataframe. The␣
 ↪function creates four subplots, one containing
    the results for the first model based on two chosen parameters, the second␣
 ↪containing the results of the second model
    based on two chosen parameters, the third containing the results of the␣
 ↪third model based on two chosen parameters
    and the fourth containing the results of the fourth model based on two␣
 ↪chosen parameters
    (can be the same of different features).

    Parameters
    ----------
    df_gs: df
        Dataframe containing the results from a GridSearchCV object.
    params1: lst
        List of parameters to plot on subplot one. The first element of the␣
 ↪list is the x axis, the second the y axis, and
        the third the hue.
    params2: lst
        List of parameters to plot on subplot two. The first element of the␣
 ↪list is the x axis, the second the y axis, and
        the third the hue.
    params3: lst
        List of parameters to plot on subplot three. The first element of the␣
 ↪list is the x axis, the second the y axis, and
        the third the hue.
    params4: lst
        List of parameters to plot on subplot four. The first element of the␣
 ↪list is the x axis, the second the y axis, and
```

```
        the third the hue.
    model1: str
        The name of the classifier used for subplot one.
    model2: str
        The name of the classifier used for subplot two.
    model3: str
        The name of the classifier used for subplot three.
    model4: str
        The name of the classifier used for subplot four.
    title: str
        Extension of the title specifying which input data.

    Returns
    -------
    None
    """

    # initialise plots and plot params for each model
    fig, ax = plt.subplots(2, 2, sharey=True, figsize=(10, 10))
    sns.scatterplot(data=df_gs, x=params1[0], y=params1[1], hue=params1[2],␣
↪ax=ax[0][0])
    sns.scatterplot(data=df_gs, x=params2[0], y=params2[1], hue=params2[2],␣
↪ax=ax[0][1])
    sns.scatterplot(data=df_gs, x=params3[0], y=params3[1], hue=params3[2],␣
↪ax=ax[1][0])
    sns.scatterplot(data=df_gs, x=params4[0], y=params4[1], hue=params4[2],␣
↪ax=ax[1][1])

    # rescale axis to display C at regular intervals
    ax[0][0].set_xscale('log')
    ax[0][1].set_xscale('log')
    ax[0][0].set_title(model1)
    ax[0][1].set_title(model2)
    ax[1][0].set_title(model3)
    ax[1][1].set_title(model4)
    ax[1][0].set_xticks([3, 5, 7, 9])
    ax[1][1].set_xticks([2, 5, 10, 20])
    fig.suptitle('Precision Scores for Different Models and Parameters with␣
↪'+title)
    fig.tight_layout()
    plt.show()
```

[25]:
```
import pandas as pd

# convert grid search results to df, and rename columns for plotting
df_gs = pd.DataFrame(gs.cv_results_)
```

```
df_gs.rename(columns={'param_classifier__C':'C', 'mean_test_score':'Precision␣
 ↪score',
                      'param_classifier__solver':'Solver',
                      'param_classifier': 'Model', 'param_classifier__kernel':
 ↪'Kernel',
                      'param_classifier__n_neighbors': 'Nearest neighbours',␣
 ↪'param_classifier__algorithm': 'Algorithm',
                      'param_classifier__max_depth': 'Max depth',␣
 ↪'param_classifier__min_samples_split': 'Min samples to split'}, inplace=True)

plot_gs_results(df_gs, ['C', 'Precision score', 'Solver'], ['C', 'Precision␣
 ↪score', 'Kernel'],
                ['Nearest neighbours', 'Precision score', 'Algorithm'], ['Min␣
 ↪samples to split', 'Precision score', 'Max depth'],
                'Logistic Regression', 'SVC', 'KNN', 'Decision Tree',␣
 ↪'Augmented Data')
```

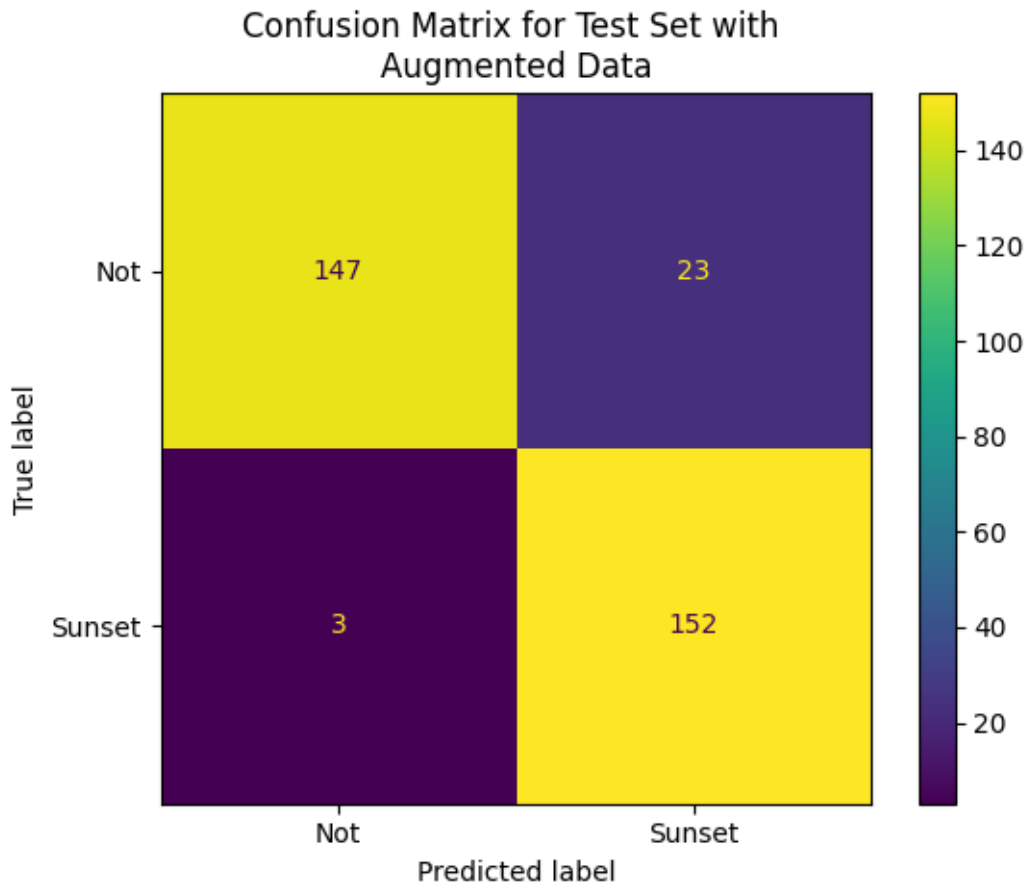


[26]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import roc_auc_score, precision_score, accuracy_score

def confusion_metrics(y_pred, title):
    """
    Computes precision, auc, and accuracy scores for the input model's␣
 ↪predictions,
    and plots a confusion matrix between the test set classifications and␣
 ↪predictions.

    Parameters
    ----------
    y_pred : arr
```

```
        Predicted target values.
    title : str
        The title end of the plot.

    Returns
    -------
    None
    """
    # precision, auc, and accuracy scores for model
    print('\033[1mScores for '+title+'\033[0m')
    print('Precision score for test set:', round(precision_score(y_test,␣
↪y_pred), 3))
    print('AUC score for test set:', round(roc_auc_score(y_test, y_pred), 3))
    print('Accuracy score for test set:', round(accuracy_score(y_test, y_pred),␣
↪3))

    # plot confusion matrix between test set classifications and predictions
    cmd = ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred),␣
↪display_labels = ['Not', 'Sunset'])
    cmd.plot()
    plt.title('Confusion Matrix for Test Set with \n'+title)
    plt.show()
```

[27]:
```
# make predictions from optimal grid search model
num_images = np.array(X_test).shape[0]
X_test_reshaped = np.array(X_test).reshape(num_images, -1)
y_pred = gs.predict(X_test_reshaped)
confusion_metrics(y_pred, 'Augmented Data')
```

```
Scores for Augmented Data
Precision score for test set: 0.869
AUC score for test set: 0.923
Accuracy score for test set: 0.92
```



[28]:
```
def plot_incorrect_classifications(x_test, y_test, y_pred):
    """
    Plots incorrect classifications made by a model.

    Parameters
    ----------
    x_test : array
        The test set images.
    y_test : array
        The true labels of the test set.
    y_pred : array
        The predicted labels of the test set.
    """
    fig, axes = plt.subplots(nrows=3, ncols=7, figsize=(15, 10))
    count = 0
    for i in range(len(y_test)):

        # only plot up to the number of subplots available in axes
```

```
        if y_test[i] != y_pred[i] and count < axes.size:
            row = count // 7
            col = count % 7

            # convert colours from cv to matplotlib
            axes[row, col].imshow(cv2.cvtColor(x_test[i], cv2.COLOR_BGR2RGB))

            # print true and predicted labels for comparison
            axes[row, col].set_title(f"True: {y_test[i]}, Predicted:␣
↪{y_pred[i]}")
            axes[row, col].axis('off')
            count += 1
    fig.subplots_adjust(wspace=0.05, hspace=0.005)
    plt.show()
```

[29]:
```
plot_incorrect_classifications(np.array(X_test), y_test, y_pred)
```