DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Master's Thesis

Study program/specialization: Applied Data Science	Spring semester 2023 Open
Authors: Daniel Grønner and Elisabeth Eik	
Supervisor: Antorweep Chakravorty	
Title of Master's Thesis: Deep learning-based automated detection and clustering of potholes using variational autoencoder for efficient road maintenance	
ECTS: 30	
Subject headings: Support Vector Machine, Variational Autoencoder	Pages: 29 + attachments: Code included in PDF: Stavanger 15. Juni 2023

Abstract

In this study we explore the possibility of using unsupervised learning for pothole detection. We will use images of roads from Brazil where both potholes and cracks can be present in the images, as well as clean images with no damage on the road. This will be done using a variational autoencoder (VAE) and clustering. The study will also explore a supervised method, support vector machine (SVM), to compare the performance of supervised model vs. unsupervised model. The goal for this study is to correctly cluster images containing potholes from images that do not contain potholes.

Acknowledgements

We would like to thank our supervisor, Antorweep Chakravorty, for all help and insightful knowledge during the writing of this thesis.

List of Figures

3.1 Images of roads with potholes 6
3.2 Images of roads with no pothole 7

4.1 Overview of varitional autoencoder architecture 15

5.1 Images of roads with cracks 20
5.2 Steps of localization of potholes 21
5.3 Example of crack image 22
5.4 Example of shadow image 22

6.1 Mask image of pothole vs localization of pothole using VAE model 24
6.2 Mask image of pothole vs localization of crack using VAE model 24

List of Tables

4.1 Confusion matrix 9

5.1 Confusion matrix for SVM model 18
5.2 Results table after 10 runs with VAE model 20
5.3 Confusion matrix for VAE model 21

Contents

Abstract **ii**

Acknowledgements **iii**

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 Problem statement 2

2 Literature **3**
2.1 Related work 3
2.1.1 Convolutional neural networks based potholes detection using thermal imaging 3
2.1.2 Real-time machine learning-based approach for pothole detection . . 4
2.1.3 Pothole detection in asphalt pavement images 4
2.1.4 Pothole Detection Using Deep Learning Classification Method . . . 5
2.1.5 Road Pothole Detection using Deep Learning Classifiers 5

3 Data **6**
3.1 Training and testing 7
3.2 Pre-processing 7
3.2.1 Resizing 8
3.2.2 Grayscale 8
3.2.3 Random flip 8
3.2.4 Flatten data 8

4 Methods **9**
4.1 Support Vector Machine 10

4.1.1 Rational for choosing support vector machine 10
4.1.2 Model explanation 10
4.2 Variational Autoencoder 12
4.2.1 Rationale Behind the VAE Training and Clustering Approach 12
4.2.2 Custom Loss 12
4.2.3 VAE Architecture 15
4.2.4 Hyperparameter Tuning 16

5 Results **18**
5.1 Support Vector Machine 18
5.2 Variational Autoencoder 20
5.2.1 Localize pothole 21

6 Discussion **23**

7 Conclusion **26**
7.1 Future work 27

Support Vector Machine Python Code **30**

Variational Autoencoder Python Code **33**

Chapter 1

Introduction

Road maintenance is an essential aspect of urban planning and infrastructure management. Potholes, in particular, pose significant safety risks to drivers, cyclists, and pedestrians while also causing accelerated wear and tear on vehicles. Identifying and repairing potholes in a timely manner is vital to ensure the longevity of road infrastructure and reduce the risk of accidents.

Traditionally, the process of identifying and analyzing potholes has been a labor intensive task, relying on manual inspections and subjective assessments. [1] With the advent of computer vision and deep learning techniques, it is now possible to develop automated systems that can efficiently detect and assess road defects. One such approach is to employ unsupervised methods, like variational autoencoders (VAEs), to cluster images containing potholes, which can reveal underlying patterns in the data and provide valuable insights for road maintenance and repair strategies.

In this work, we introduce the concept of using VAEs as a potential solution for clustering images of potholes, with a data set of 2235 images in consideration. We will discuss the motivation behind this approach, outline its potential benefits, and provide an overview of the problem and its significance.

The thesis contains 7 section, and is structured as follows; chapter 2 contains related work that is relevant to this thesis. In chapter 3 we go into detail about the data and the pre-processing of it, before moving on to chapter 4 where the experimental methods of this thesis will be. Further on chapter 5 will contain the results for the different methods in the previous chapter and chapter 6 discusses the performance of the models. Lastly will be chapter 7, which will include conclusion and future work.

1.1 Background and Motivation

Cracks and potholes in roads are a significant concern for transportation infrastructure. These defects can cause damage to vehicles, reduce road safety, and increase maintenance

costs. Manual inspection of roads is time-consuming, labor-intensive, and often unreliable, which has led to the development of automated systems for detecting and classifying cracks and potholes. One of the critical challenges in the automated crack and pothole detection is the wide range of variability in the shape, size, and appearance of cracks and potholes. Recent advances in image processing and machine learning have led to the development of more robust techniques for object recognition, such as template matching and anomaly clustering. [2]

Anomaly clustering is a technique used to identify patterns or clusters in a data set that do not conform to the expected patterns. For example, this technique helps identify unusual or unexpected patterns in the distribution of road cracks and potholes, which can help infrastructure maintenance crews prioritize repairs and allocate resources more effectively.[3]

The primary motivation behind utilizing VAEs for clustering pothole images lies in their ability to learn meaningful latent space representations in an unsupervised manner. By capturing the essential features of the images, VAEs can facilitate the identification of common characteristics indicative of specific types of potholes or related to particular road conditions. This information can be valuable for planning targeted road maintenance efforts and optimizing repair strategies.

Moreover, VAEs offer several advantages over traditional autoencoders, including their probabilistic nature, which allows for the generation of new samples and the discovery of meaningful latent space representations. This is particularly relevant in the context of pothole detection, as it enables the model to adapt to new, unseen instances, making it more robust and versatile. [4]

1.2 Problem statement

Our goal is to explore the potential of VAEs as a solution for effectively clustering a data set of 2235 images, revealing underlying patterns and providing insights into the characteristics of different types of potholes. To achieve this, we will investigate the implementation of VAEs and examine their suitability for the task at hand.

Investigating the application of VAEs for clustering pothole images has the potential to significantly impact the field of road maintenance and infrastructure management. By providing an automated means of identifying and analyzing potholes, VAEs can streamline the decision-making process for road repairs and contribute to the development of more effective maintenance strategies. Ultimately, this could lead to improved road safety, reduced vehicle damage, and more efficient allocation of resources for road infrastructure management.

Chapter 2

Literature

In this chapter we will look at review work on pothole detection algorithms. The main focus in this literature study is finding work related to pothole detection, and will be used to compare our results to later in this thesis.

2.1 Related work

2.1.1 Convolutional neural networks based potholes detection using thermal imaging

Aparna et al. [5] tried to use a convolutional neural network (CNN) to detect potholes by using thermal images. Objects are rarely of the same temperature as other objects around it, and thermal imaging can therefore have an advantage in areas that are not well lit or have poor weather conditions like fog or rain.

A convolutional neural network-based model was created, which takes thermal images of potholes and non-pothole roads as input. After the model had been trained, it would predict if the input image is of a pothole or not.

In this study 500 images of potholes and non-potholes were collected manually and used in the convolutional neural network. There was conducted two main experiments which was:

- Self-built CNN models
- CNN based ResNet models

The self-built CNN-model was run twice with different parameters, where it achieved a testing accuracy of 64.42% and 73.06%. The ResNet model was tested out with different numbers of layers ranging from 18 to 152, different image sizes, and different train-validation split. The accuracies on the two image sizes, 224 and 240, were almost similar, and the highest validation accuracy achieved was 97.08%, which was with image size 224 and the ResNet101 model, having train-validation split of 90:10.

2.1.2 Real-time machine learning-based approach for pothole detection

The study conducted by Egaji et al. [6] compared the performance of five different classification models on pothole detection, Naïve Bayes, logistic regression, support vector machine(SVM), K-nearest neighbor(KNN) and random forest tree. The data was acquired using apps on smartphones which were mounted at the center of a car's dashboard. A passenger was required to hold a second device with a second app and press and hold a button when the vehicle was approaching a pothole to record until the car had driven over the pothole. The data was split into 80% for training, 10% for validation and 10% for testing. A stratified K-fold cross-validation was applied to the training data set with K=10.

For the training data set, SVM, random forest tree and KNN performed best with an average accuracy at 0.8200 ± 0.1598, 0.8071 ± 0.1259 and 0.7879 ± 0.1371, respectively, with a 95% confidence interval. For the validation data set, random forest tree and KNN performs best with a F-score of over 77%. The KNN gets a recall of 1 meaning it finds all the actual potholes, but gets a precision of 0.7778 meaning it classifies non-pothole data as pothole data. On the test data set all models get an accuracy of 83% or higher, except for the logistic regression model. Overall in the test data set random forest tree performed best with accuracy, precision, recall and F-score of 0.8889, 1, 0.7778, and 0.8750, respectively.

2.1.3 Pothole detection in asphalt pavement images

A study performed by Koch and Brilakis [7] created a framework that can detect defects on pavements. The images used in this study are either cropped from available survey videos, or cropped from pavement video data collected by passenger vehicles equipped with a high speed-camera. The data set contains 120 pavement images (50 to train and 70 to test), with a variety of shape and size of potholes, other defects such as cracks, non-defect pavements and diverse lighting conditions. Their proposed model contains mainly three components:

- Image segmentation
- Shape extraction
- Texture extraction and comparison

They used a histogram shape-based thresholding algorithm to segment the image into defect and non-defect regions. By using morphological thinning and elliptic regression, the potential pothole shape is approximated. The texture inside a potential pothole is then extracted and compared with the surroundings to determine if the region is an actual pothole. This is an important step in order to distinguish between false candidates and true potholes. The pothole detection method was implemented in Matlab. After testing the method, the model reaches an overall accuracy of 86%, with 82% precision and 86% recall. These results are promising, but their approach is vision based and therefore relies on normal lighting.

2.1.4 Pothole Detection Using Deep Learning Classification Method

Chemikala Saisree and Dr. Kumaran U [1] utilized a deep leaning algorithm to classify images in a data set on whether the roads were plain, or had potholes. They are comparing images of muddy roads together with images of highway roads. The images were converted to a readable format and resized before they were used as input to the three different models, ResNet50, InceptionResNetV2 and VGG19. The data set was split into training and testing, in order to train and evaluate the model respectively. For the highway roads the training set is set to 65% and 35% for testing, whereas for the muddy roads the training is set to 60% and 40% for testing.

For the highway road data set the VGG19 performed best with an accuracy of 97.91%, and a validation accuracy of 97.79%. The validation accuracy for the ResNet50 and InceptionResNetV2 model was 92.82% and 94.48% respectively. The loss was also lowest for the VGG19 model, having a value of 4.41%, whereas ResNet50 had the highest loss with 17.95%.

For the muddy road data set the VGG19 performed best again with 98.17% accuracy and 96.36% on validation accuracy. ResNet50 and InceptionResNetV2 model got a validation accuracy of 95.91% and 90.45%.

2.1.5 Road Pothole Detection using Deep Learning Classifiers

Arjapure et al. [8] tried using a convolutional network based approach for detecting potholes from road images, the images used were raw road-pothole and clean images of roads. The pothole images obtained were in various shapes, sizes and lightning conditions, and non-defect road images from the local road of Mumbai, nearby highways and the internet source Kaggle. They had a total of 838 images with image size 224x224, with a train:validation split of 722:116 images. 7 different pre-trained models were used to compare the results of the self-built CNN. The pre-trained models were ResNet50, ResNet152, ResNet50V2, ResNet152V2, InceptionV3, InceptionResNetV2 and DenseNet201. All models accept image size of 224x224, exept InceptionV3 and InceptionResNetV2 which accepts input images of size 299x299.

Their self-built CNN consisted of 5 convolutiotion layers with ReLU activation function. They used precision, recall and accuracy to measure the performance of the CNN model. The average training accuracy of the CNN model with 25 epochs came to 97.65%, and average validation accuracy 80.17%.The model achieved a precision of 71.64% and recall of 92.3%. From the pre-trained models, ResNet50 perfomed best on the training set with a training accuracy of a 100%. For the validation set InceptionResNetV2 and DenseNet201 performed best with a validation accuracy of 89.66%.

Chapter 3

Data

The data set being used in this thesis contains 2235 samples of road images, where each image has three masks that shows the road path, pothole and cracks in the road. The images were captured between 2014 and 2017 and was made available by Brazilian National Department of Transport Infrastructure (NDTI). The images are from highways in the states of Espírito Santo, Rio Grande do Sul and the Federal District in Brazil. The images were selected manually having the four criteria listed below. [9]

- To count as an image with damaged asphalt, cracks or potholes must be present
- Do not contain vehicles in same lane
- Do not contain people in image
- No problems due to capture, as defects in colors or in the image

The masks for each image are only considering the right lane of the road.

Before we can begin to develop any model for pothole detection, the data needs to be pre-processed, which is addressed in the following subchapter.

Some of the images are exemplified in figure 3.1 and figure 3.2

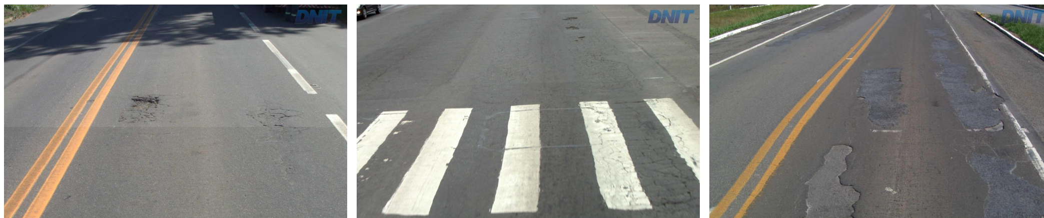

Figure 3.1: Images of roads with potholes

In figure 3.1 we can see images of roads with potholes. Some of the images have more severe and bigger potholes, while others are smaller. We can also see that the potholes are in different locations for each image, and that the road shape differ as can be seen in the middle picture where there is no yellow line separating the two lanes.

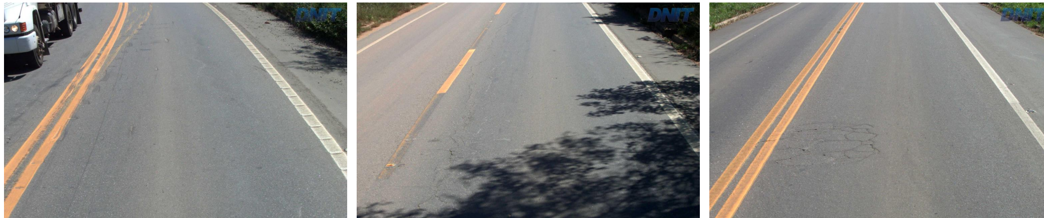

Figure 3.2: Images of roads with no pothole

In figure 3.2 the images vary with shadows and cars in the opposite lane. Although there are no potholes present in these images, there might be cracks as can be seen in the images.

As these are just 6 examples of images from the data set we can see that it varies in both road shape, shadows and severity of pothole and cracks.

We will now look further on splitting the data into training and testing, and then how to pre-process the data before we can use it in the model.

3.1 Training and testing

The data set was split into training and testing before using it in the models. It was split on a 80/20 percentage of the data set, before it was further pre-processed with resizing, grayscale, and flipping, which will be further explained in the next chapters. After the split the training data set contains 1791 images and test set 445 images.

The data being used in the different models all contain the same data for training and testing so the training and testing is equal for all models. The images split into training and testing all contains the same percentage of smaller and bigger potholes.

The training set is also split into train and validation to use in the variational autoencoder model.

3.2 Pre-processing

In this subchapter we will look at the pre-processing of the data. This is an important step as pre-processing improves accuracy and reliability. The below sections will be different forms of pre-processing.

3.2.1 Resizing

Each image have a size of 1024x640, meaning each image contains 655 360 pixels. When multiplying with the amounts of images we have available there are a total amount of 1 464 729 600 pixels. By using this many pixels it will severely slow down the time it takes for the model to train. To solve this we will resize the image to 128x128 giving us 16 384 pixels for each image, and a total of 36 618 240 pixels for all images. Python has a variety of functions for this, but we will use Python's OpenCV tool "resize". [10]

3.2.2 Grayscale

Another important factor of reducing data from the images is to convert them to grayscale. Color images are digitally stored as an array of red, blue and green values. The computer mixes these images to produce color outputs. Each image then have three layers, one for each of the colors. By converting the images to grayscale, we are left with only one layer, with values ranging between 0 (white) and 255 (black). [11]

By doing this we convert the images from the three layers of RGB (red, green blue) to one grayscale layer.

3.2.3 Random flip

To accommodate for overfitting, random horizontal flip is used on the training data set before using it in the different models. The probability for flipping the images is set to 30%, meaning 30 % of the images in the training data set will be flipped horizontally before using the images to train the model.

3.2.4 Flatten data

For the supervised model the input needs to be an one-dimensional array. We accommodate this by flattening the data. To do this we use Python's reshape function. [12]

Chapter 4

Methods

In this chapter we will explain the different models that have been trained and tested on the data set, and measure the accuracy for each of the models. The two models that have been chosen for this thesis is support vector machine and variational autoencoder. The support vector machine is the baseline model for the thesis and is used to compare the variational autoencoder. The literature study conducted in chapter 2 will also be used to compare how the model perform, but this will be discussed in chapter 6.

By using the confusion matrix we can calculate accuracy, recall and precision. Accuracy will lie between 0 and 100, and precision and recall between 0 and 1. Precision is the proportion of positive identifications that are correctly classified, and recall are the proportions of actual positives that are correctly identified. The confusion matrix and formulas are given below. [13]

		Actual values	
		0	1
Predicted values	0	True negative (TN)	False Negative (FN)
	1	False Positive (FP)	True positive (TP)

Table 4.1: Confusion matrix

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

4.1 Support Vector Machine

Support vector machine is a supervised machine learning technique that can be used for both classification and regression tasks. Since our task is trying to classify pothole images from non-pothole images we will focus on SVM for classification. The algorithm constructs a hyperplane in multidimensional space to separate the different classes. It generates the optimal hyperplane in an iterative manner, where the idea is to find a maximum marginal hyperplane that best divides the data set into classes. [14] [15]

4.1.1 Rational for choosing support vector machine

We want to have a baseline model to compare our results from other models. From [6] we saw that support vector machine were one of the models that got one of the best accuracy on both training and testing on pothole detection. This is why we hope the support vector machine can be a good fit for our data set and achieve good results for pothole detection. Other literature also found support vector machine as a good model for pothole detection. [16] As our approach is to find a good unsupervised model for pothole detection, we want to compare it to a supervised, which is an additional reason for choosing to use support vector machine as a pothole detection baseline model.

4.1.2 Model explanation

To start off we read the data from the train and test data set from chapter 3 and do the necessary pre-processing steps explained in chapter 3.2.

To create the SVM model we will use scikit learn's support vector machine library [17] and use GridSearchCV [18] for hyperparameter tuning.

Hyperparameter tunning

By using GridSearchCV the hyperparameter tuning takes far less time to compute versus doing it manually. [19] We do hyperparameter tuning to find the best parameters for the SVM model. To do this we need to try all possible combinations of values to know which parameters are optimal for the model. We do this by passing a grid of parameters, here called ”param_grid”, which consists of the following parameters.

- C: 0.001, 0.01, 0.1, 1, 10, 100
- gamma: 1, 0.1, 0.01, 0.001, 0.0001
- kernel: linear, rbf

C, gamma and kernel are some of the hyperparameters for an SVM model. These parameters will be run for each combination, total of 40 candidates, with 10 folds each. In total there will be 400 fits to run through to find the best hyperparameters.

C is a hyperparameter in SVM to control error, meaning a low value for C means low error and a high value for C means high error. Even though a low value means low error does not mean it equals a good model and will depend on the data set.

Gamma is a hyperparameter that decides how much curvature we want in the decision boundary. A high gamma means more curvature and a low gamma means less curvature. Since gamma decides the curve, it is only used when the kernel is rbf (Radial basis function). If the kernel is linear or poly only the hyperparameter C is needed. The value of gamma depends on the data set as it does with the value of C. By doing hyperparameter tuning we can find the most optimal parameters from this. [20] Even though gamma is only needed when the kernel is rbf, there is not a way to specify this when doing hyperparameter tuning, so the hyperparameter tuning runs through all combinations regardless.

All of the hyperparameters are set before training the data.

After running the hyperparameter tuning with GridSearchCV, the algorithm prints out the best combination of the parameter grid. For the our data set the best parameters were 10, 0.01, and rbf for C, gamma and the kernel, respectively, and this will be used in the SVM model fit for training.

Support vector machine algorithm

As previously mentioned we will use scikit learn's support vector machine library to create our SVM model with the best hyperparameters found from hyperparameter tuning. To begin with we create an instance of the support vector machine, and use model.fit [21] to train the model in a specified number of iterations. In the SVM model we do one iteration.

After fitting the model, we save it as a .joblib file. Now that the model is created it can be tested on the test data set which we will discuss further in chapter 6. The whole model for the support vector machine can be found in appendix 7.1

4.2 Variational Autoencoder

4.2.1 Rationale Behind the VAE Training and Clustering Approach

This approach trains a variational autoencoder (VAE) model to analyze and cluster images containing potholes in an unsupervised manner. The main motivation behind choosing a VAE is its ability to learn a compact and continuous representation of the input data in the latent space. This enables the model to capture the underlying structure of the data, which is particularly useful when working with complex and high-dimensional inputs like images. [22]

The VAE model consists of two main components: an encoder and a decoder. The encoder maps input images to the latent space, while the decoder reconstructs the input images from the latent space representations. In VAEs, the latent space is represented by two parameters: mean (μ) and log-variance ($\log \sigma^2$), which define a Gaussian distribution. The model learns to generate a latent vector z by sampling from this distribution during training. This adds a stochastic component to the model, which helps in capturing the inherent variability in the data and making the latent space more robust and informative. [22] [4]

The main advantage of using the μ latent space over the z latent space is that μ represents the Gaussian distribution's center, providing a more stable and reliable representation of the data. In contrast, the z latent space is obtained by sampling from the distribution, and therefore, it can have higher variability. By clustering in the μ latent space, we can obtain more consistent and meaningful clusters that better reflect the underlying structure of the data.

The VAE model uses the training, validation, and testing data set from chapter 3 and is then trained on the training set using a custom loss function that combines reconstruction loss, KL divergence, and a pothole-specific loss term. This encourages the model to learn a more meaningful latent space representation that captures the relevant features of pothole images.

After training, the model is evaluated on the test set by extracting the latent space representations (μ) and applying clustering algorithms such as Agglomerative Clustering [23]. The resulting clusters can be used to group similar pothole images together and analyze the different types of potholes present in the data set.

The chosen VAE-based approach effectively captures the underlying structure of the pothole images in a continuous and compact latent space. By clustering in the μ latent space, we obtain more consistent and meaningful groupings of the images, which can be helpful for further analysis and decision-making.

4.2.2 Custom Loss

The primary goal of the customized loss function is to make the variational autoencoder (VAE) focus on anomaly detection. This loss function integrates the concepts of Mean

Squared Error (MSE) loss, Kullback-Leibler Divergence (KLD) loss, and a local MSE loss. These components enhance the model's capacity to identify, prioritize, and locate anomalies.

Components of CustomLoss

The 'CustomLoss' function comprises three crucial components: the global MSE loss, the KLD loss, and the local MSE loss. Each of these components plays a unique role in guiding the model's learning and anomaly detection abilities.

Mean Squared Error (MSE) Loss The MSE loss measures the difference between the input (original) and output (reconstructed) images. The global MSE loss, calculated over the whole image, ensures that the VAE is adept at accurate reconstruction, a prerequisite for effective anomaly detection.[4]

Local MSE Loss Local MSE loss is designed to lend additional weightage to the anomalies in the image. A mask isolates the pixels where the MSE loss exceeds the mean MSE loss over the image, thereby identifying potential anomalies. These regions are then subjected to local MSE loss calculation. The 'local_weight' hyperparameter controls this component's contribution, ensuring that the model can be fine-tuned to focus more intently on these regions during training.

Kullback-Leibler Divergence (KLD) The KLD loss ensures that the learned latent variables align with a standard normal distribution, achieving a well-structured latent space. This is crucial for efficiently learning representations that are more sensitive to anomalies.[4]

The 'CustomLoss' function imparts several advantages to the VAE. First, with the local MSE loss and a high 'local_weight', the model's attention is directed toward anomalies in the image. This enables effective anomaly localization in the reconstruction phase. The KLD loss ensures a structured latent space by making the learned latent variables conform to a standard normal distribution. This contributes to a more focused latent representation of anomalies in the input space. The global MSE loss guarantees the model's proficiency in accurate reconstruction. After training, the difference between the original and reconstructed image in regions of potential anomalies can be used to visualize and locate the anomaly. The hyperparameters 'alpha', 'beta', and 'local_weight' offer flexibility in balancing the emphasis on accurate reconstruction (MSE loss), sensitivity to anomalies (local MSE loss), and structuring the latent space (KLD loss).

The 'CustomLoss' function provides a powerful and versatile mechanism for training a VAE for anomaly detection. It equips the model to reconstruct images accurately, efficiently

focus on anomalies in the latent space and makes it possible to cluster the images on the latent space.

Description of the CustomLoss Class

The 'CustomLoss' class is initialized with three hyperparameters: 'alpha', 'beta', and 'local_weight'. 'alpha' and 'beta' control the contribution of the global MSE loss and the KLD loss, respectively, while 'local_weight' controls the contribution of the local MSE loss.

The forward method of the class first calculates the MSE loss per pixel. This results in a tensor of the same shape as the inputs, containing the MSE loss for each pixel. The global MSE loss is then calculated by summing up these losses.

A mask is created by identifying pixels where the MSE loss per pixel exceeds the mean MSE loss over the entire image. This mask effectively identifies regions where the variational autoencoder's reconstructions are less accurate, which can be considered potential anomalies. The local MSE loss is calculated by summing the MSE losses of these pixels and then multiplying by the 'local_weight'.

The KLD loss is calculated using the standard formula for the Kullback-Leibler Divergence in VAEs, and is weighted by 'beta'. The total loss is then calculated as the sum of the global MSE loss weighted by 'alpha', the KLD loss, and the local MSE loss. This total loss is what the model aims to minimize during training.

The 'CustomLoss' class provides a simple, flexible, and powerful interface for defining the loss function of a VAE, making it easier to train models that focus on detecting and clustering anomalies in images.

The total loss can be expressed as:

$$L_{total} = \alpha \cdot L_{MSEglobal} + \beta \cdot L_{KLD} + \gamma \cdot L_{MSElocal} \tag{4.1}$$

Where:

1. $L_{MSEglobal}$ is the global mean square error, defined as:

$$L_{MSEglobal} = \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 \tag{4.2}$$

2. L_{KLD} is the Kullback-Leibler divergence, defined as:

$$L_{KLD} = -\frac{1}{2} \sum_{i=1}^{n} (1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2) \tag{4.3}$$

3. $L_{MSElocal}$ is the local mean square error, calculated only for the regions of the image where the global MSE exceeds its mean:

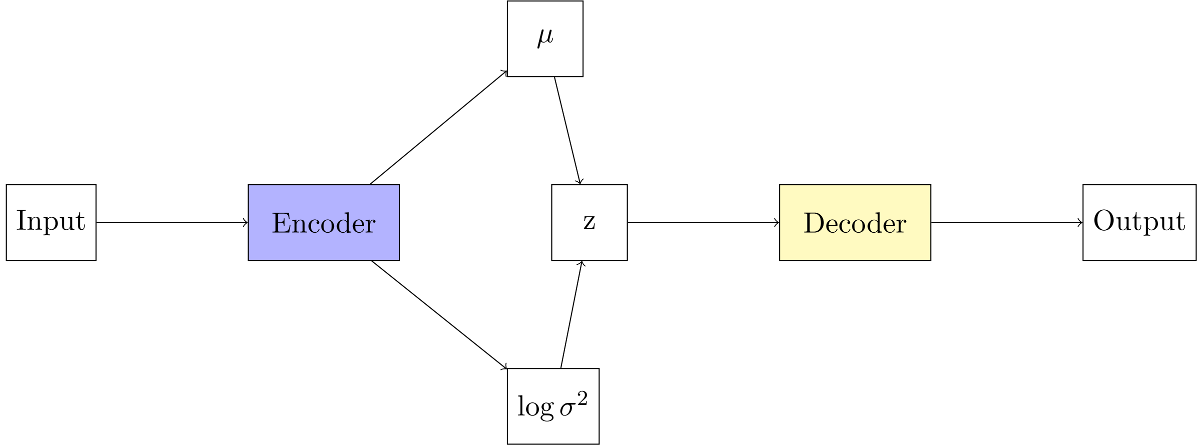

Figure 4.1: Overview of varitional autoencoder architecture

$$L_{MSElocal} = lw \cdot \sum_{i=1}^{n} (x_i - \hat{x}_i)^2 \cdot I((x_i - \hat{x}_i)^2 > \mu((x - \hat{x})^2)) \tag{4.4}$$

In the last equation, $I(\cdot)$ is an indicator function that is 1 if the condition is satisfied and 0 otherwise.

4.2.3 VAE Architecture

The variational autoencoder (VAE) architecture consists of two main parts: the encoder and the decoder. The encoder compresses the input image into a lower-dimensional latent space while the decoder reconstructs the image from the latent representation. The architecture is designed to optimize the reconstruction quality and the latent space distribution by minimizing a custom loss function.

The encoder is a series of convolutional layers with batch normalization and Leaky ReLU activation functions. Max-pooling layers are used for downsampling the feature maps. The output of the encoder is flattened and fed into two fully connected layers that predict the mean and log-variance of the latent variables. The reparameterization trick is applied to sample latent variables from the predicted distributions, ensuring that the model can be trained using gradient-based optimization methods.

The decoder is a series of transposed convolutional layers with ReLU activation functions, responsible for upsampling the feature maps and reconstructing the input image. The final layer uses a sigmoid activation function to output pixel values in the range [0, 1].

The custom loss function used in this VAE implementation is designed to handle the specific problem of detecting potholes in images. It combines the reconstruction loss (measuring the difference between the input image and the reconstructed image), the Kullback-

Leibler divergence (encouraging the latent space to follow a Gaussian unit distribution), and a pothole-aware loss term that is sensitive to the presence of potholes in the input images. This custom loss function aims to balance reconstruction quality and the ability to detect potholes effectively.

Encoder: The encoder is a series of convolutional layers followed by batch normalization and LeakyReLU activation functions. The role of the encoder is to extract features from the input image and produce a lower-dimensional latent space representation. The encoder employs max-pooling layers for downsampling and spatial compression.

Reparameterization trick: To enforce the latent space to follow a Gaussian distribution, the model uses the reparameterization trick [24]. This involves learning the mean (μ) and log-variance (log_var) of the Gaussian distribution and sampling from this distribution using the reparameterization formula: z = $\mu + \epsilon$ * std, where ϵ is a random sample from a standard normal distribution.

Decoder: The decoder is a series of transposed convolutional layers followed by ReLU activation functions, except for the last layer, which uses a sigmoid activation function. The role of the decoder is to reconstruct the input image from the latent space representation. The decoder upsamples the latent representation to produce an output with the exact dimensions as the original input.

Loss function: The model employs a custom loss function, which combines the reconstruction loss, the KL-divergence loss, and the local MSE loss. The custom loss is described in subchapter 4.2.2.

Training and validation: The model is trained using the provided data set and validated using a separate validation set. An early stopping mechanism is implemented to prevent overfitting. The best model is saved based on the lowest validation loss.

4.2.4 Hyperparameter Tuning

The efficacy of the VAE model is enhanced through hyperparameter optimization, implemented using the Optuna library [25]. The hyperparameters under consideration include the dimension of the latent space (z_{dim}), the learning rate (lr), and the local weight factor in the custom loss function ($local_weight$). The search intervals for these parameters are defined as follows:

- z_{dim}: Integer values within the range of 10 to 100
- lr: Log-uniform distribution between 1e-6 and 1e-3
- $local_weight$: Floating point values from 5 to 1000

The model is trained with the selected hyperparameters in each trial, and the F1-score for clustering on the validation set is computed as the objective function that needs to be maximized. Utilize the agglomerative clustering algorithm to analyze the latent representations of the images.

The script in Appendix 8 was used to determine the labels in the data set. The script scans through the provided binary images. Each binary image represents the original picture, with potholes, if present, highlighted in white against a black background. A white region indicates a pothole if the image is not entirely black and the image name is stored in a set. This set thus contains the names of all images featuring potholes. The script then writes these names into a .txt file. This file serves as a record of images in the data set representing potholes, and it is used to assign labels to the images when computing the F1-score.
The Optuna study is configured to conduct 100 trials, the objective being to find the best combination of hyperparameters that maximizes the F1-score. The optimal hyperparameters identified through the study are subsequently reported.
Fine-tuning the hyperparameters z_{dim}, lr, and $local_weight$ is vital. Proper adjustment of these hyperparameters enables the VAE to learn a concise representation of the input data and a discriminative feature space for detecting potholes. The F1-score is selected as the evaluation metric for hyperparameter tuning as it provides a balanced assessment of the model's performance. Since the data set contains 2235 images and only 564 contain potholes, the data set is imbalanced. This imbalance could lead to a model favoring the majority class (non-pothole images), performing inadequately on the minority class (pothole images). In such situations, relying on accuracy alone can be misleading. The F1-score, however, considers both precision and recall, making it a more robust metric for imbalanced data sets, ensuring the model's proficiency in identifying both pothole and non-pothole images.

Chapter 5

Results

In this chapter we will present the results for the models described in chapter 4 with the data and pre-processing from chapter 3. We will look at the different forms of measuring how well the model performs, with accuracy, precision and recall for the different models.

5.1 Support Vector Machine

We will start by looking at the results for the support vector machine model. As mentioned in chapter 4.1 we found the best parameters by doing hyperparameter testing. From doing 10 runs for each combination we got the values 10, 0.01 and rbf for C, gamma and the kernel, respectively. These parameters were used to train the model with the training set, and then tested on the test set. The confusion matrix is displayed below after using the best parameters on the test set.

		Actual values	
		No pothole	Pothole
Predicted values	No pothole	317	56
	Pothole	20	52

Table 5.1: Confusion matrix for SVM model

The no pothole section for both predicted and actual values are relatively larger than the other categories, which makes sense as there is a non-even spread of pothole images vs non-pothole images in the data set. In total there are 337 images without potholes vs 108 images with potholes.

The actual images of potholes are split roughly 50/50 on the predicted no pothole and pothole cluster. From the confusion matrix above we can calculate the different measure-

ments from chapter 4.

Accuracy: (317 + 52)/(317 + 20 + 56 + 52) = 82.9 %
Precision: 52/(52 + 56) = 48.1%
Recall: 52/(52+20) = 72.2%

As we can see the support vector machine gets a quite high accuracy of 82.9%, which is about the same as the literature study from chapter 2.1.2 where the SVM model got an accuracy of 83%. This is a good accuracy, but we also need to look at the precision and recall measurements before we can say if it is a good model or not.

As mentioned previously, there are 108 images of potholes in the test set, and we want these images in the same cluster. From the confusion matrix there are 52 images of potholes correctly classified, and 56 images of potholes incorrectly classified. By looking at this accuracy, only 48% of the potholes are correctly classified, in other words the precision of the model. When the model predicts an image as a pothole, it is correct 48.1% of the time.

Precision measures the accuracy of positive predictions, whereas recall measures the completeness of positive predictions. It is desirable to have both high precision and recall. [26] The recall measurement is at a decent percentage, but the precision is not sufficient for it to be a good model.

Now let's look at the results for the variational autoencoder model.

5.2 Variational Autoencoder

For the next model we have a results table of the 10 runs on the test set and the average test results at the end.

Run ID	**Agglomerative Correct Percentage**	**Agglomerative Elements in Class**	**Total number in Class**
1	80.6 %	87	341
2	80.6 %	87	339
3	80.6 %	87	340
4	80.6 %	87	340
5	78.7 %	85	330
6	80.6 %	87	341
7	80.6 %	87	340
8	80.6 %	87	341
9	76.9 %	83	332
10	80.6 %	87	339
Average	80 %	86	338

Table 5.2: Results table after 10 runs with VAE model

After 10 runs the amount of correctly classified potholes is an average of 80%, meaning an average of 86 images of potholes are classified to the same cluster. The total number of images in that class is an average of 338, where 252 images without potholes gets clustered together with the 86 images with potholes. Among the 86 images of potholes, the cluster also contains images with cracks. Some examples are shown in figure 5.1

Figure 5.1: Images of roads with cracks

We will also take a look at the confusion matrix with the different measurements like we did with the support vector machine.

		Actual values	
		No pothole	Pothole
Predicted values	No pothole	171	22
	Pothole	252	86

Table 5.3: Confusion matrix for VAE model

From the actual values in the pothole column we can see that 86 pothole images gets clustered together, which is the same from table 5.2. Now we can calculate accuracy, precision and recall for the VAE model:

Accuracy: (171 + 86)/(171 + 22 + 252 + 86) = 48.4 %
Precision: 86/(86 + 22) = 79.6%
Recall: 86/(86 + 252) = 25.4%

In the VAE model the overall accuracy is 48.4% and recall measuring 25.4%. Precision gets a better measurement of 79.6%, which is the same as the measurements from table 5.3

5.2.1 Localize pothole

The image series shown in figures 5.2, 5.3 and 5.4 presents three scenarios demonstrating the strengths and weaknesses of our road damage identification model.

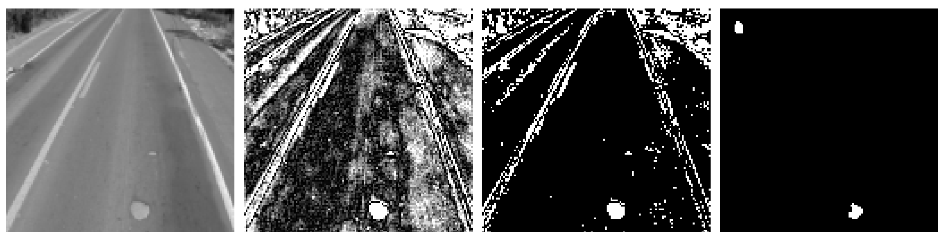

Figure 5.2: Steps of localization of potholes

We see an ideal case in figure 5.2 where the model performs optimally. The initial image from the left presents the original road surface in grayscale. The second image shows the outcome after applying an absolute difference operation between the original image and the model's reconstructed image, highlighting the discrepancies that represent potential damage areas. The third image presents the binary version, produced by applying a threshold operation to the absolute difference image, thereby emphasizing the distinct

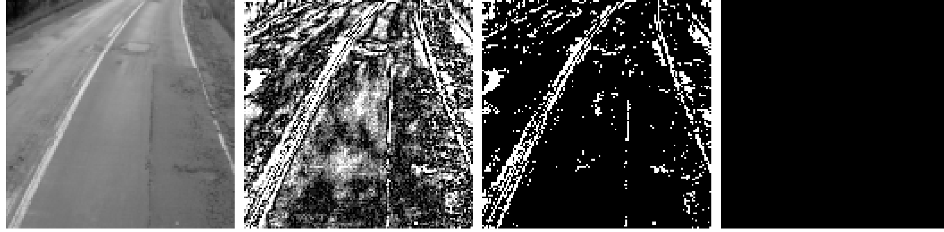

Figure 5.3: Example of crack image

Figure 5.4: Example of shadow image

differences. The final image shows the result after removing noise and eliminating lane lines from the binary image, providing a clear representation of potential road damage areas.

In figure 5.3 we see that the model is able to find some cracks in the image, but struggles more on this compared to the pothole image. Figure 5.4 showcases the model's challenge: the impact of shadows on the road. With the same sequence of transformations as described in the first set, it becomes clear that shadows can create additional complexity and lead to potential misinterpretations by the model. Despite the exact steps of absolute difference operation, threshold, and noise and line removal, shadows can introduce discrepancies, influencing the model's accuracy in identifying actual areas of road damage. This comparison between the two sets highlights the model's potential, while emphasizing areas for future improvement.

Now that we have looked more closely on the results of the two models, the next chapter will contain the discussion of both model, and use these results as a comparison among other interesting finds.

Chapter 6

Discussion

This thesis main focus is to correctly classify pothole images in the same group in an unsupervised manner, and we will discuss the two different methods mostly on how accurately they are able to do this. In addition we will also include other findings that may be relevant or interesting to this approach.

From the results in chapter 5 we saw that the support vector machine got the highest accuracy of 82.9 % compared to the variational autoencoder that achieved an accuracy of 48.4%. By looking more closely at the numbers we saw that the variational autoencoder correctly predicted 80% of the pothole images, whereas the SVM model only got 48% of the pothole images correctly classified.

As previously mentioned the reason for choosing support vector machine as the baseline model was due to its high ability to correctly classify potholes from chapter 2.1.2 and to have a supervised model to compare other methods to. Based on our results the model did not perform as well as hoped on the pothole data set with the pre-processing methods that were used. It may perform better with other pre-processing techniques or feature extractions, but this is a discussion for future work in chapter 7.1.

From the results for the variational autoencoder we saw that it outperformed the support vector machine by correctly classifying 80% of the pothole images to the same cluster. As mentioned in the previous chapter, the pothole cluster also contained 252 other images from the test data set where many of these contained other road damage as cracks. Even though the goal of the model is to find potholes, other road damage is also a severe risk for cars and people. The model is therefore ideal to find road damage such as cracks, before it turns into potholes.

A main advantage for using the variational autoencoder is the ability to localize potholes as seen in chapter 5.2.1. Figure 6.1 shows an instance of the mask for the pothole on the left, together with the binary image in the middle and noise-filtering image on the right. By comparing the three images in figure 6.1 they all look similar to one another.

The reason to why the three images are in different shapes are because of the resizing

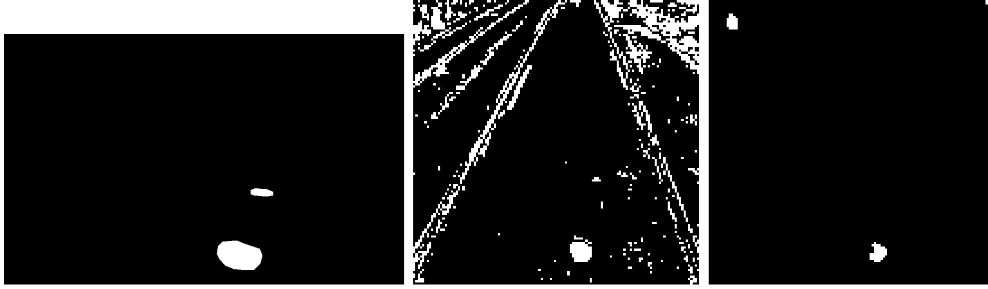

Figure 6.1: Mask image of pothole vs localization of pothole using VAE model

that was done on the raw image in the pre-processing step in chapter 3. From the three images we can clearly see the pothole in the same spot in all of them.

Figure 6.2 showcase the same sequence of images as in figure 6.1, but here we have used an image with a crack present in the road instead of a pothole.

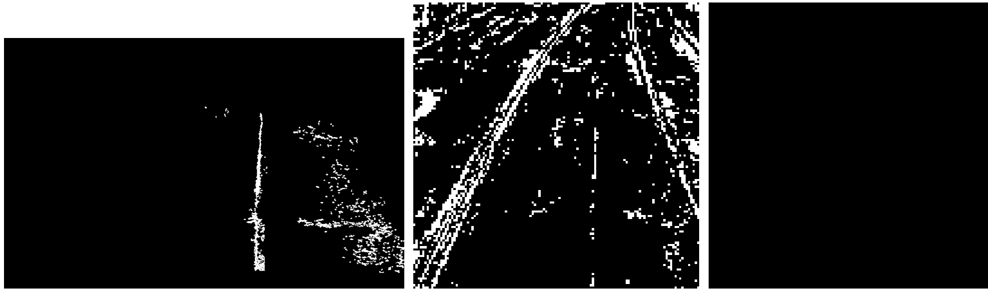

Figure 6.2: Mask image of pothole vs localization of crack using VAE model

The left image shows the cracks present in the road, which is the mask image. Even though there is a lot of noise in the middle image in figure 5.3, we can clearly see the crack line matching up on the left and middle image. When we try to remove the noise in the right image, everything is removed and we are left with a black image. This technique is not suitable for images containing cracks.

By comparing the two figures 6.1 and 6.2, the model does a good job on localizing potholes, but needs more work on the cracks. A drawback for the variational autoencoder are shadows in the images, which can be seen in figure 5.4. Even though this is not desirable, it is a small drawback for the model which can be further optimized.

To summarize the support vector machine does not perform well on our data set splitting the pothole images 50/50 in each cluster, whereas the variational autoencoder is able to cluster 80% of the pothole images together. The VAE model is also able to localize

potholes, but is struggling when shadows are present in the image. This can be addressed as discussed in future work in chapter 7.1.

Chapter 7

Conclusion

Despite the support vector machine (SVM) model's overall high accuracy, it exhibited a limited ability to classify pothole images, resulting in distributing them evenly across its two clusters. The variational autoencoder (VAE) significantly outperformed the SVM regarding classifying pothole images, by successfully grouping 80% of pothole images within the same cluster. The VAE did exhibit a higher misclassification rate on non-pothole images, but there is a compelling reason behind this. Though not featuring explicit potholes, many misclassified images contained considerable road damage, such as substantial cracks. In this light, the VAE's 'misclassifications' can be seen more as an extended focus on road conditions, effectively identifying and clustering images of roads that require maintenance - whether they contain potholes or not. Even though not all of the images in the pothole cluster showcase potholes, many contain road damage in need of repair. This can justify the larger size of the pothole cluster. Therefore, the VAE's ability to cluster images of damaged roads, whether due to potholes or extensive cracking, is advantageous, as all these conditions require attention and remediation. Despite its lower general accuracy, the VAE provides more valuable insights for road damage identification and classification in this context.

7.1 Future work

For future work, other models should be tested on the data set, both supervised and unsupervised. There also exists other pre-processing and feature extraction techniques that are possible to test on the data set, to check for accuracy improvement. In this final section we will go through various methods that can be interesting to try for further improvement.

The data we have worked on does not contain many clean images, i.e. images without cracks or potholes. From the mask provided in the data set, only 16 images are completely clean from potholes or cracks, which made it hard for the VAE model to cluster images with potholes from images without potholes as many images with cracks were placed in the cluster with potholes. For further work more clean images without potholes or cracks should be included in the data set to check for improvement of clustering pothole images vs. non-pothole images.

From chapter 2 we saw that there was other supervised models like convolutional neural network, random forest and KNN that also performed well on pothole detection. These could be interesting to implement and test on the data set. For convolutional neural network there exists multiple pre-trained models that performed well on other pothole data sets, like VGG19 from 2.1.4. This model got an accuracy of 97% on highway roads, which are quite good results. The VGG19 is also possible to use as a feature extraction technique, together with multiple other feature extraction techniques that can be found in [27].

For unsupervised models, K-means and anomaly detection, both outlier and novelty, could be models worth testing as there has not been found any related work which have used these methods. These methods can be tested out together with the feature extraction techniques for example see if this improves the model.

As highlighted in section 5.2.1, our model demonstrates potential in localizing potholes, but there is a clear opportunity for enhancing its performance in visualizing road cracks. The technique currently employed involves calculating absolute differences between the original images and their reconstructed counterparts to identify areas of road damage. This approach has proven effective in exposing significant anomalies, such as potholes. However, it also introduces a certain degree of noise into the image. The subsequent noise filtration process, while crucial for sharpening the clarity of our results, unfortunately, tends to eliminate the finer details, such as cracks from the binary images. Therefore, the focus will be to devise a more nuanced strategy for noise reduction that preserves these minor defects. The ultimate objective is to fine-tune the model to efficiently filter out the noise without obscuring the cracks, allowing both potholes and cracks to be accurately localized within the images. This refinement should substantially enhance the comprehensive utility of our model in road damage detection and localization.

Bibliography

[1] Saisree and Kumaran. Pothole detection using deep learning classification method, 2023. Last accessed 29th May 2023.

[2] Ralph Caubalejo. Image processing — template matching, 2018. Last accessed 12th May 2023.

[3] Angsuman Dutta. All you need to know about anomaly detection. Last accessed 12th May 2023.

[4] Irhum Shafkat. Intuitively understanding variational autoencoders. Last accessed 5th June 2023.

[5] Aparna et al. Convolutional neural networks based potholes detection using thermal imaging, 2022. Last accessed 28h February 2023.

[6] Egaji et al. Real-time machine learning-based approach for pothole detection, 2021. Last accessed 1st March 2023.

[7] Koch and Brilakis. Pothole detection in asphalt pavement images, 2011. Last accessed 8th March 2023.

[8] Arjapure et al. Road pothole detection using deep learning classifiers, 2020. Last accessed 1st June 2023.

[9] National Department of Transport Infrastructure. Cracks-and-potholes-in-road-images-dataset, 2020. Last accessed 1st March 2023.

[10] OpenCV. Opencv resize image using cv2.resize(). Last accessed 11th May 2023.

[11] Joseph Nelson. When to use grayscale as a preprocessing step, 2020. Last accessed 14th May 2023.

[12] NumPy. numpy.reshape, 2022. Last accessed 10th June 2023.

[13] Sarang Narkhede. Understanding confusion matrix, 2018. Last accessed 10th June 2023.

[14] anuragrazarwal. Image classification using support vector machine (svm) in python, 2023. Last accessed 11th May 2023.

[15] Bahauddin Taha. Build an image classifier with svm!, 2021. Last accessed 11th May 2023.

[16] Fox et al. Crowdsourcing undersampled vehicular sensor data for pothole detection, 2023. Last accessed 24th May 2023.

[17] Scikit learn. Support vector machines. Last accessed 14th May 2023.

[18] Scikit learn. sklearn.model_selection.gridsearchcv. Last accessed 14th May 2023.

[19] Great learning team. Hyperparameter tuning with gridsearchcv. Last accessed 22nd May 2023.

[20] A Man Kumar. C and gamma in svm. Last accessed 22th May 2023.

[21] Keras. Model training apis. Last accessed 25th May 2023.

[22] Will Badr. Uncovering anomalies with variational autoencoders (vae): A deep dive into the world of unsupervised learning. Last accessed 12th april 2023.

[23] Satyam Kumar. "agglomerative clustering and dendrograms — explained". Last accessed 14nd march 2023.

[24] Sayak Paul. "reparameterization" trick in variational autoencoders. Last accessed 2nd march 2023.

[25] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[26] Purva Huilgol. Precision and recall | essential metrics for data analysis. Last accessed 24th May 2023.

[27] Keras. Keras applications. Last accessed 24th May 2023.

Support Vector Machine Python Code

```
import os
os.environ['CUDA_VISIBLE_DEVICES'] = '2,5'
import pandas as pd
import numpy as np
from sklearn import svm
from sklearn.model_selection import GridSearchCV
import cv2
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline
import joblib


def get_pothole_image_names(file_path):
    with open(file_path, "r") as f:
        return [line.strip() + "_RAW.png" for line in f]

pothole_name = get_pothole_image_names('pothole.txt')

image_dir = {
        "train": "../images/raw",
        "test": "../images/test"
}

image_tuple_list_train = [(cv2.imread(os.path.join(image_dir['train'], image_name), 0), 1 if image_name in pothole_name else 0) for image_name in os.listdir(image_dir['train'])]
image_tuple_list_test = [(cv2.imread(os.path.join(image_dir['test'], image_name), 0), 1 if image_name in pothole_name else 0) for image_name in os.listdir(image_dir['test'])]

def preprocess_images(image_tuple_list, img_size=(128, 128)):
    images, labels = zip(*image_tuple_list)
    images_resized = [cv2.resize(img, img_size, interpolation=cv2.INTER_AREA) for img in images]
    return np.array(images_resized), np.array(labels)
```

```
31
32 def random_horizontal_flip(images, labels, p=0.3):
33     flipped_images, flipped_labels = [], []
34     for image, label in zip(images, labels):
35         if np.random.rand() < p:
36             flipped_image = np.fliplr(image)
37             flipped_images.append(flipped_image)
38             flipped_labels.append(label)
39         else:
40             flipped_images.append(image)
41             flipped_labels.append(label)
42     return np.array(flipped_images), np.array(flipped_labels)
43
44 X_train, y_train = preprocess_images(image_tuple_list_train)
45
46 X_test, y_test = preprocess_images(image_tuple_list_test)
47
48 X_train, y_train = random_horizontal_flip(X_train, y_train, p=0.3)
49
50 X_train_flat, X_test_flat = X_train.reshape(X_train.shape[0], -1), X_test.
       reshape(X_test.shape[0], -1)
51
52 param_grid = {
53     'svc__C': [0.01, 0.1, 1, 10, 100],
54     'svc__gamma': [0.001, 0.01, 0.1, 1],
55     'svc__kernel': ['linear', 'rbf']
56 }
57
58 svm_pipeline = make_pipeline(
59     MinMaxScaler(),
60     svm.SVC()
61 )
62
63 grid_search = GridSearchCV(svm_pipeline, param_grid, cv=10, verbose=2)
64
65 print('fitting')
66 grid_search.fit(X_train_flat, y_train)
67
68 best_params = grid_search.best_params_
69 print(f"Best parameters: {best_params}")
70
71 best_svm = grid_search.best_estimator_
72 best_svm.fit(X_train_flat, y_train)
73
74 model_filename = "svm_classifier.joblib"
75 joblib.dump(best_svm, model_filename)
76 print(f"Model saved to {model_filename}")
77
78
79 loaded_model = joblib.load(model_filename)
```

```python
80 test_predictions = loaded_model.predict(X_test_flat)
81 
82 test_image_names = [image_name for image_name in os.listdir(image_dir['test'])]
83 test_results_df = pd.DataFrame({'ImageName': test_image_names, 'PredictedLabel'
       : test_predictions})
84 test_results_df.to_csv('test_results.csv', index=False)
```

Listing 1: Support Vector Machine Python code

Variational Autoencoder Python Code

```
1 import os
2 from PIL import Image
3 from torch.utils.data import Dataset
4 
5 class CustomImageDataset(Dataset):
6     def __init__(self, root_dir, transform=None):
7         self.root_dir = root_dir
8         self.transform = transform
9         self.image_files = sorted(os.listdir(root_dir))
10 
11     def __len__(self):
12         return len(self.image_files)
13 
14     def __getitem__(self, idx):
15         img_name = self.image_files[idx]
16         img_path = os.path.join(self.root_dir, img_name)
17         try:
18             image = Image.open(img_path).convert('RGB')
19         except (IOError, SyntaxError) as e:
20             print(f'Could not read image {img_path}. Error: {e}')
21             return None, img_name, idx
22 
23         if self.transform:
24             image = self.transform(image)
25 
26         return image, img_name, idx
```

Listing 2: Dataset

```python
import torch
import torch.nn as nn

class CustomPotholeLoss(nn.Module):
    def __init__(self, alpha=0, beta=0, local_weight=0):
        super(CustomPotholeLoss, self).__init__()
        self.alpha = alpha
        self.beta = beta
        self.local_weight = local_weight
        self.mse_loss = nn.MSELoss(reduction='none')

    def forward(self, reconstructions, inputs, mu, log_var):
        mse_loss_per_pixel = self.mse_loss(reconstructions, inputs)
        global_mse_loss = torch.sum(mse_loss_per_pixel)

        high_diff_mask = mse_loss_per_pixel > (torch.mean(mse_loss_per_pixel))
        local_mse_loss = torch.sum(mse_loss_per_pixel * high_diff_mask.float())
 * self.local_weight

        kld_loss = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

        total_loss = self.alpha * global_mse_loss + self.beta * kld_loss +
    local_mse_loss
        return total_loss
```

Listing 3: Custom Loss

```
1 import torch
2 import torch.nn as nn
3 from CustomLoss import CustomPotholeLoss
4
5 DIM = 4
6
7 class VAE(nn.Module):
8     def __init__(self, in_channels, latent_dim, alpha, beta,local_weight,device
    ):
9         super(VAE, self).__init__()
10
11        self.input_channels = in_channels
12        self.z_dim = latent_dim
13        self.model_path = "model_path.pth"
14        self.custom_loss=CustomPotholeLoss(alpha=alpha, beta=beta,local_weight=
    local_weight)
15        self.device = device
16
17        self.encoder = nn.Sequential(
18            nn.Conv2d(self.input_channels, 32, 4, stride=1, padding=2),
19            nn.BatchNorm2d(32),
20            nn.LeakyReLU(0.1),
21            nn.MaxPool2d(2, stride=2),
22            nn.Conv2d(32, 64, 4, stride=1, padding=2),
23            nn.BatchNorm2d(64),
24            nn.LeakyReLU(0.1),
25            nn.MaxPool2d(2, stride=2),
26            nn.Conv2d(64, 128, 4, stride=1, padding=2),
27            nn.BatchNorm2d(128),
28            nn.LeakyReLU(0.1),
29            nn.MaxPool2d(2, stride=2),
30            nn.Conv2d(128, 256, 4, stride=1, padding=2),
31            nn.BatchNorm2d(256),
32            nn.LeakyReLU(0.1),
33            nn.MaxPool2d(2, stride=2),
34            nn.Conv2d(256, 512, 4, stride=1, padding=2),
35            nn.BatchNorm2d(512),
36            nn.LeakyReLU(0.1),
37            nn.MaxPool2d(2, stride=2),
38        )
39
40        self.fc_mean = nn.Linear(512 * DIM * DIM, self.z_dim)
41        self.fc_log_var = nn.Linear(512 * DIM * DIM, self.z_dim)
42        self.fc_decode = nn.Linear(self.z_dim, 512 * DIM * DIM)
43
44        self.decoder = nn.Sequential(
45            nn.ConvTranspose2d(512, 256, 4, stride=2, padding=1),
46            nn.ReLU(),
47            nn.ConvTranspose2d(256, 128, 4, stride=2, padding=1),
48            nn.ReLU(),
```

```
49             nn.ConvTranspose2d(128, 64, 4, stride=2, padding=1),
50             nn.ReLU(),
51             nn.ConvTranspose2d(64, 32, 4, stride=2, padding=1),
52             nn.ReLU(),
53             nn.ConvTranspose2d(32, self.input_channels, 4, stride=2, padding=1)
    ,
54             nn.Sigmoid()
55         )
56 
57     def reparameterize(self, mu, log_var):
58         std = log_var.mul(0.5).exp_()
59         eps = torch.randn_like(std)
60         return mu + eps * std
61 
62     def decoding(self,z):
63         return self.decoder(self.fc_decode(z).view(-1, 512, DIM, DIM))
64 
65     def encoding(self, x):
66         x = self.encoder(x)
67         x = x.view(-1, 512 * 4 * 4)
68         mu = self.fc_mean(x)
69         log_var = self.fc_log_var(x)
70         z = self.reparameterize(mu, log_var)
71         return z, mu, log_var
72 
73     def forward(self, x):
74         z, mu, log_var = self.encoding(x)
75         x_recon = self.decoding(z)
76         return z,x_recon, mu, log_var
77 
78 
79     def fit(self, dataloaders, optimizer, scheduler, num_epochs=10,
    early_stopping=25, verbose=True):
80         best_loss = float('inf')
81         epochs_without_improvement = 0
82 
83         for epoch in range(num_epochs):
84             if verbose:
85                 print("\n")
86                 print(f"Epoch {epoch + 1}/{num_epochs}")
87                 print("-" * 50)
88 
89             for phase in ['train', 'val']:
90                 if phase == 'train':
91                     self.train()
92                 else:
93                     self.eval()
94 
95                 running_loss = 0.0
96 
```

```
97                  for inputs, _, _ in dataloaders[phase]:
98                      inputs = inputs.to(self.device)
99  
100                     optimizer.zero_grad()
101 
102                     with torch.set_grad_enabled(phase == 'train'):
103                         _, reconstructed, mu, log_var = self(inputs)
104                         loss = self.custom_loss(reconstructed, inputs, mu,
    log_var)
105 
106                         if phase == 'train':
107                             loss.backward()
108                             optimizer.step()
109 
110                     running_loss += loss.item() * inputs.size(0)
111 
112                 epoch_loss = running_loss / len(dataloaders[phase].dataset)
113 
114                 if phase == 'train':
115                     scheduler.step(epoch_loss)
116 
117                 if verbose:
118                     print(f"{phase.capitalize()} Loss: {epoch_loss:.4f}")
119 
120                 if phase == 'val':
121                     if epoch_loss < best_loss:
122                         if verbose:
123                             print(f"New best model found! Saving the model to {
    self.model_path}")
124 
125                         best_loss = epoch_loss
126                         torch.save(self.state_dict(), self.model_path)
127                         epochs_without_improvement = 0
128                     else:
129                         epochs_without_improvement += 1
130 
131                     if epochs_without_improvement >= early_stopping:
132                         if verbose:
133                             print(f"Early stopping triggered")
134                         break
135 
136                 if verbose:
137                     print("-" * 50)
138 
139             if epochs_without_improvement >= early_stopping:
140                 break
141 
142         if verbose:
```

```
143                 print("Best val loss: {:4f}".format(best_loss))
```

Listing 4: Model

```python
import torch
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
from torchvision import transforms
from model import VAE
from dataset import CustomImageDataset
from config import Config
from utils import compare_classes_with_txt, clustering_and_f1_score,
    reconstruct_images
from diff import process_images
import pandas as pd
import os

def main():

    config = Config()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    DATA_TRANSFORMS = {
        'train': transforms.Compose([
            transforms.Resize((config.img_dim, config.img_dim)),
            transforms.Grayscale(num_output_channels=1),
            transforms.RandomApply([transforms.RandomHorizontalFlip()], p=0.3),
            transforms.ToTensor(),

        ]),
        'val': transforms.Compose([
            transforms.Resize((config.img_dim, config.img_dim)),
            transforms.Grayscale(num_output_channels=1),
            transforms.ToTensor(),

        ]),
    }

    IMAGE_DIR = {
        "raw": "../images/raw",
        "test": "../images/test"
    }

    dataset = CustomImageDataset(root_dir=IMAGE_DIR['raw'], transform=
    DATA_TRANSFORMS['train'])
    test_dataset = CustomImageDataset(root_dir=IMAGE_DIR['test'], transform=
    DATA_TRANSFORMS['val'])

    train_size = len(dataset)
    val_size = int(train_size * 0.1)
    train_size = train_size - val_size

    train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
    val_dataset.dataset.transform = DATA_TRANSFORMS['val']
```

```
48
49      dataloaders = {
50          'train': DataLoader(train_dataset, batch_size=config.param['batch_size'
        ], shuffle=True, num_workers=4, pin_memory=True),
51          'val': DataLoader(val_dataset, batch_size=config.param['batch_size'],
        shuffle=True, num_workers=4, pin_memory=True),
52          'test': DataLoader(test_dataset, batch_size=config.param['batch_size'],
         shuffle=False, num_workers=4, pin_memory=True),
53      }
54
55      model = VAE(
56          in_channels=config.in_channels,
57          latent_dim=config.param['z_dim'],
58          alpha=config.param['alpha'],
59          beta=config.param['beta'],
60          local_weight=config.param['local_weight'],
61          device=device
62      ).to(device)
63
64      optimizer = optim.Adam(model.parameters(), lr=config.param['lr'])
65      scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
66                                                             mode='min',
67                                                             patience=3,
68                                                             factor=0.1,
69                                                             verbose=config.
        verbose)
70
71      model.fit(
72          dataloaders=dataloaders,
73          optimizer=optimizer,
74          scheduler=scheduler,
75          num_epochs=config.num_epochs,
76          early_stopping=config.early_stopping,
77          verbose=config.verbose
78      )
79
80      print('Training complete.')
81      torch.cuda.empty_cache()
82
83      state_dict = torch.load(model.model_path)
84      model.load_state_dict(state_dict)
85
86      predicted_labels, test_image_names, latent_vectors =
        clustering_and_f1_score(model, test_dataset)
87
88      results_df = pd.DataFrame(
89          data={
90              "Image Name": test_image_names,
91              "Cluster Label": predicted_labels,
92          }
```

```
93     )
94 
95     results_df.to_csv(f'results/clustering_results.csv', index=False)
96 
97     max_common_class, _, _, _ = compare_classes_with_txt(test_image_names,
       predicted_labels)
98 
99     reconstruct_images(model,
100                         test_dataset,
101                         test_image_names,
102                         predicted_labels,
103                         max_common_class,
104                         latent_vectors)
105 
106     process_images(input_folder,
107                     output_folder,
108                     abs_diff_folder,
109                     binary_diff_folder,
110                     removed_noise_diff_folder)
111 
112 
113 
114 if __name__ == "__main__":
115     input_folder = f'output/original'
116     output_folder = f'output/reconstructed'
117     abs_diff_folder = f'output/abs_diff'
118     binary_diff_folder = f'output/binary_diff'
119     removed_noise_diff_folder = f'output/removed_noise_diff'
120 
121     if not os.path.exists('results'):
122         os.makedirs('results')
123 
124     main()
```

Listing 5: main

```python

class Config:
    def __init__(self):
        self.num_epochs = 1
        self.in_channels = 1
        self.img_dim = 128
        self.early_stopping = 25
        self.verbose = True


        self.param = {
            "z_dim": 75,
            "lr": 1e-05,
            "batch_size": 64,
            "alpha": 1,
            "beta": 1,
            "local_weight": 100.0
        }
```

Listing 6: config

```python
import torch
import os
from typing import List, Tuple, Any
from torch import Tensor
from sklearn.cluster import AgglomerativeClustering
from torchvision.utils import save_image
import numpy as np


def compare_classes_with_txt(image_names: List[str], predicted_labels: List[int
    ]) -> int:
    """
    Compares classes with .txt file

    Args:
        image_names: List of image names
        predicted_labels: List of predicted labels
        csv_file_name: .txt file to compare with

    Returns:
        String representing the maximum common class
    """
    class_dict = {}

    for img_name, pred_label in zip(image_names, predicted_labels):
        class_label = str(pred_label)
        if class_label not in class_dict:
            class_dict[class_label] = []
        class_dict[class_label].append(img_name)

    txt_files = get_pothole_image_names("pothole.txt")

    common_counts = {}
    total_common = 0

    for class_label, file_names in class_dict.items():
        common_count = len(set(txt_files).intersection(file_names))
        common_counts[class_label] = common_count
        total_common += common_count

    max_common_class = max(common_counts, key=common_counts.get)
    max_common_count = common_counts[max_common_class]
    print(f'max common class is {max_common_class}, withch have {
    max_common_count}')
    print(f'Number in max class : {len(class_dict[max_common_class])}')
    print(f'total pothole :{total_common}')

    return int(max_common_class), max_common_count, total_common, len(
    class_dict[max_common_class])

```

```python
48 def clustering_and_f1_score(vae: Any, dataset: List[Tensor]) -> Tuple[List[int
    ], AgglomerativeClustering, List[str], List[float]]:
49     """
50     Performs clustering and calculates the F1 score
51 
52     Args:
53         vae: The Variational Autoencoder model
54         dataset: The dataset of images
55 
56     Returns:
57         A tuple containing predicted labels, image names and latent space
58     """
59     latent_space = []
60     image_names = []
61 
62     vae.eval()
63     with torch.no_grad():
64         for inputs, img_names, _ in dataset:
65             inputs = inputs.unsqueeze(0).to(vae.device)
66             z, mu, _ = vae.encoding(inputs)
67             latent_space.append(mu.cpu().numpy())
68             image_names.append(img_names)
69 
70     latent_space = np.vstack(latent_space)
71     clustering = AgglomerativeClustering(n_clusters=2)
72     predicted_labels = clustering.fit_predict(latent_space)
73 
74     return predicted_labels, image_names, latent_space
75 
76 
77 def reconstruct_images(vae: Any, dataset: List[Tensor], image_names: List[str],
     predicted_labels: List[int], max_common_class: str, latent_space: List[
    float], output_dir: str = "output"):
78     """
79     Reconstructs images using the VAE
80 
81     Args:
82         vae: The Variational Autoencoder model
83         dataset: The dataset of images
84         image_names: List of image names
85         predicted_labels: List of predicted labels
86         max_common_class: The class with the maximum number of common elements
87         latent_space: List of latent spaces
88         output_dir: The directory to save the images
89     """
90     os.makedirs(os.path.join(output_dir, "original"), exist_ok=True)
91     os.makedirs(os.path.join(output_dir, "reconstructed"), exist_ok=True)
92     os.makedirs(os.path.join(output_dir, "filtered_out"), exist_ok=True)
93 
94     vae.eval()
```

```
95      with torch.no_grad():
96          for idx, (inputs, img_name, _) in enumerate(dataset):
97              if img_name in image_names:
98                  if predicted_labels[idx] == max_common_class:
99                      latent = torch.tensor(latent_space[idx]).unsqueeze(0).to(
    vae.device)
100                     outputs = vae.decoding(latent)
101 
102                     save_image(inputs.unsqueeze(0).cpu(), os.path.join(
    output_dir, "original", img_name))
103                     save_image(outputs.cpu(), os.path.join(output_dir, "
    reconstructed", img_name))
104             else:
105                 save_image(inputs.unsqueeze(0).cpu(), os.path.join(
    output_dir, "filtered_out", img_name))
106 
107 def get_pothole_image_names(file_path: str) -> List[str]:
108     """
109     Reads image names from a .txt file
110 
111     Args:
112         file_path: The path of the .txt file
113 
114     Returns:
115         A list of image names
116     """
117     with open(file_path, "r") as f:
118         return [line.strip() + "_RAW.png" for line in f]
```

Listing 7: utils

```python
import os
from PIL import Image
import numpy as np

binary_images_folder = '../images/binary'

binary_image_names = []
for filename in os.listdir(binary_images_folder):
    img_path = os.path.join(binary_images_folder, filename)
    img = Image.open(img_path)
    img_np = np.array(img)

    if np.any(img_np):
        binary_image_names.append(filename.replace('_POTHOLE.png', ''))

binary_image_names = set(binary_image_names)

def save_image_names_to_txt(names: set, file_path: str):
    """
    Writes image names to a .txt file

    Args:
        names: A set of image names
        file_path: The path of the .txt file
    """
    with open(file_path, "w") as f:
        for name in names:
            f.write(name + "\n")

save_image_names_to_txt(binary_image_names, "pothole.txt")
```

Listing 8: Creating Labels

```python
import torch
from model import VAE
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
from torchvision import transforms
import numpy as np
import optuna
from sklearn.cluster import AgglomerativeClustering
from dataset import CustomImageDataset
from sklearn.metrics import f1_score
from config import Config
import os

def latent_representations(model, dataloader):
    model.eval()
    latent_representations = []

    with torch.no_grad():
        for inputs, _, _ in dataloader:
            input_images = inputs.to(model.device)
            z,_, mu, _ = model(input_images)
            latent_representations.append(mu.cpu().numpy())

    return np.vstack(latent_representations)

def calculate_f1_score(vae, dataloader, pothole_image_names):
    latent_vectors = latent_representations(vae, dataloader)
    true_labels = []
    image_names = []

    for _, image_paths, _ in dataloader:
        for raw_image_path in image_paths:
            image_names.append(os.path.basename(raw_image_path))

    image_names = np.array(image_names)

    clustering_Agglomerative = AgglomerativeClustering(n_clusters=2)
    predicted_labels = clustering_Agglomerative.fit_predict(latent_vectors)

    pothole_image_names_set = set(pothole_image_names)

    common_counts = {}
    for label in set(predicted_labels):
        common_count = len(set(pothole_image_names_set).intersection(
    image_names[predicted_labels.tolist() == label]))
        common_counts[label] = common_count

    max_common_label = max(common_counts, key=common_counts.get)

    predicted_labels[predicted_labels == max_common_label] = 1
```

```python
50      predicted_labels[predicted_labels != 1] = 0
51
52      for image_name in image_names:
53          true_labels.append(1 if image_name in pothole_image_names else 0)
54
55      f1 = f1_score(true_labels, predicted_labels, average='weighted')
56      return f1
57
58
59  config = Config()
60
61  DATA_TRANSFORMS = {
62      'train': transforms.Compose([
63          transforms.Resize((config.img_dim, config.img_dim)),
64          transforms.Grayscale(num_output_channels=1),
65          transforms.RandomApply([transforms.RandomHorizontalFlip()], p=0.3),
66          transforms.ToTensor(),
67
68      ]),
69      'val': transforms.Compose([
70          transforms.Resize((config.img_dim, config.img_dim)),
71          transforms.Grayscale(num_output_channels=1),
72          transforms.ToTensor(),
73
74      ]),
75  }
76
77  IMAGE_DIR = {
78  "raw": "../images/raw",
79  "test": "../images/test",
80  }
81
82  dataset = CustomImageDataset(root_dir=IMAGE_DIR['raw'], transform=
        DATA_TRANSFORMS['train'])
83  test_dataset = CustomImageDataset(root_dir=IMAGE_DIR['test'], transform=
        DATA_TRANSFORMS['val'])
84
85  train_size = len(dataset)
86  val_size = int(train_size * 0.1)
87  train_size = train_size - val_size
88
89  train_dataset, val_dataset = random_split(dataset, [train_size, val_size])
90  val_dataset.dataset.transform = DATA_TRANSFORMS['val']
91
92
93  raw_image_names = os.listdir(IMAGE_DIR['raw'])
94
95  pothole_image_names = []
96  with open("pothole.txt", "r") as f:
97      for line in f:
```

```python
98          pothole_image_names.append(line.strip() + "_RAW.png")
99
100 dataloaders = {
101     'train': DataLoader(train_dataset, batch_size=config.param['batch_size'],
        shuffle=True, num_workers=4, pin_memory=True),
102     'val': DataLoader(val_dataset, batch_size=config.param['batch_size'],
        shuffle=True, num_workers=4, pin_memory=True),
103     'test': DataLoader(test_dataset, batch_size=config.param['batch_size'],
        shuffle=False, num_workers=4, pin_memory=True),
104 }
105
106 def objective(trial):
107     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
108
109     z_dim = trial.suggest_int("z_dim", 10, 100)
110     lr = trial.suggest_float("lr", 1e-6, 1e-3,log=True)
111     local_weight = trial.suggest_float("local_weight", 5, 1000)
112
113     model = VAE(
114         in_channels=config.in_channels,
115         latent_dim=z_dim,
116         alpha=config.param['alpha'],
117         beta=config.param['beta'],
118         local_weight=local_weight,
119         device=device
120     ).to(device)
121
122     model.model_path = "hyper.pth"
123
124     optimizer = optim.Adam(model.parameters(), lr=lr,)
125     scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min
        ', patience=3, factor=0.1, verbose=False)
126
127     model.fit(dataloaders=dataloaders,
128               optimizer=optimizer,
129               scheduler=scheduler,
130               num_epochs=config.num_epochs,
131               early_stopping=config.early_stopping,
132               verbose=config.verbose)
133
134
135     f1 = calculate_f1_score(model, dataloaders['val'],pothole_image_names)
136     return f1
137
138 study = optuna.create_study(direction='maximize')
139 study.optimize(objective, n_trials=100)
140
141 best_trial = study.best_trial
142 print("Best hyperparameters found:")
```

```
143 print(best_trial.params)
```

Listing 9: Hyperparameter tuning

```
1  import cv2
2  import numpy as np
3  import os
4
5  DIFF_FILTER_KERNEL = np.array([[-1, -1, -1], [-1, 23, -1], [-1, -1, -1]], dtype
       =float)
6  DILATION_KERNEL = np.ones((5, 5), np.uint8)
7  MIN_LINE_LENGTH = 50
8  MAX_LINE_GAP = 2
9  THRESHOLD_MIN = 250
10 MIN_AREA = 2
11 MIN_CIRCULARITY = 0.3
12
13
14 def calculate_absolute_difference(input_img, output_img, filter_kernel=
       DIFF_FILTER_KERNEL, median_kernel_size=3, bilateral_diameter=50,
       bilateral_sigma_color=5, bilateral_sigma_space=5):
15     """
16     Calculate absolute difference between two images, apply a filter and median
        blur, and bilateral filter to the result.
17     """
18     diff = cv2.absdiff(input_img, output_img)
19     diff = cv2.filter2D(diff, -1, filter_kernel)
20     diff = cv2.bilateralFilter(diff, bilateral_diameter, bilateral_sigma_color,
        bilateral_sigma_space)
21
22     return diff
23
24 def calculate_binary_difference(diff):
25     _, diff = cv2.threshold(diff, THRESHOLD_MIN, 255, cv2.THRESH_BINARY)
26     return diff
27
28 def calculate_noise_removed_difference(diff):
29     diff = remove_lane_lines(diff)
30     diff = keep_circular_shapes(diff)
31
32     return diff
33
34 def remove_lane_lines(img):
35     edges = cv2.Canny(img, 100, 200, apertureSize=3)
36     edges = cv2.dilate(edges, DILATION_KERNEL, iterations=2)
37     lines = cv2.HoughLinesP(edges, 1, np.pi / 180, 30, minLineLength=
       MIN_LINE_LENGTH, maxLineGap=MAX_LINE_GAP)
38
39     if lines is not None:
40         for line in lines:
41             x1, y1, x2, y2 = line[0]
42             length = np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
43
44             if length > 5 :
```

```
45                 cv2.line(img, (x1, y1), (x2, y2), (0, 0, 0), 2)
46
47     return img
48
49 def keep_circular_shapes(img):
50     contours, _ = cv2.findContours(img, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
51     circular_shapes_img = np.zeros_like(img)
52
53     for contour in contours:
54         area = cv2.contourArea(contour)
55         perimeter = cv2.arcLength(contour, True)
56
57         if perimeter > 0:
58             circularity = 4 * np.pi * area / (perimeter * perimeter)
59
60             if area > MIN_AREA and circularity > MIN_CIRCULARITY:
61                 cv2.drawContours(circular_shapes_img, [contour], -1, 255, -1)
62
63     return circular_shapes_img
64
65 def process_images(input_folder, output_folder, abs_diff_folder,
    binary_diff_folder, removed_noise_diff_folder):
66     os.makedirs(abs_diff_folder, exist_ok=True)
67     os.makedirs(binary_diff_folder, exist_ok=True)
68     os.makedirs(removed_noise_diff_folder, exist_ok=True)
69
70     input_files = sorted(os.listdir(input_folder))
71     output_files = sorted(os.listdir(output_folder))
72
73     for idx, (input_file, output_file) in enumerate(zip(input_files,
    output_files)):
74         input_img = cv2.imread(os.path.join(input_folder, input_file), cv2.
    IMREAD_GRAYSCALE)
75         output_img = cv2.imread(os.path.join(output_folder, output_file), cv2.
    IMREAD_GRAYSCALE)
76
77         abs_diff = calculate_absolute_difference(input_img, output_img)
78         binary_diff = calculate_binary_difference(abs_diff.copy())
79         removed_noise_diff = calculate_noise_removed_difference(binary_diff.
    copy())
80
81         cv2.imwrite(os.path.join(abs_diff_folder, input_file), abs_diff)
82         cv2.imwrite(os.path.join(binary_diff_folder, input_file), binary_diff)
83         cv2.imwrite(os.path.join(removed_noise_diff_folder, input_file),
    removed_noise_diff)
84
85 def main():
86     input_folder = f'output/original'
87     output_folder = f'output/reconstructed'
```

```
88     abs_diff_folder = f'output/abs_diff'
89     binary_diff_folder = f'output/binary_diff'
90     removed_noise_diff_folder = f'output/removed_noise_diff'
91     process_images(input_folder, output_folder, abs_diff_folder,
       binary_diff_folder, removed_noise_diff_folder)
92 
93 if __name__ == "__main__":
94     main()
```

Listing 10: Find difference

```python
1 import torch
2 import torch.optim as optim
3 from torch.utils.data import DataLoader, random_split
4 from torchvision import transforms
5 from model import VAE
6 from dataset import CustomImageDataset
7 from config import Config
8 from utils import compare_classes_with_txt, clustering_and_f1_score
9 import pandas as pd
10 import os
11
12 def main():
13
14     metrics_df = pd.DataFrame(columns=['Run', 'Max Common Class', 'Max Common
    Count', 'Number in Max Class', 'Total Pothole'])
15
16     if not os.path.exists('results'):
17         os.makedirs('results')
18
19     for run in range(1):
20         print(f'Run: {run}')
21         config = Config()
22         device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
23
24         DATA_TRANSFORMS = {
25             'train': transforms.Compose([
26                 transforms.Resize((config.img_dim, config.img_dim)),
27                 transforms.Grayscale(num_output_channels=1),
28                 transforms.RandomApply([transforms.RandomHorizontalFlip()], p
    =0.3),
29                 transforms.ToTensor(),
30
31             ]),
32             'val': transforms.Compose([
33                 transforms.Resize((config.img_dim, config.img_dim)),
34                 transforms.Grayscale(num_output_channels=1),
35                 transforms.ToTensor(),
36
37             ]),
38         }
39
40         IMAGE_DIR = {
41             "raw": "../images/raw",
42             "test": "../images/test"
43         }
44
45         dataset = CustomImageDataset(root_dir=IMAGE_DIR['raw'], transform=
    DATA_TRANSFORMS['train'])
46         test_dataset = CustomImageDataset(root_dir=IMAGE_DIR['test'], transform
    =DATA_TRANSFORMS['val'])
```

```
47
48        train_size = len(dataset)
49        val_size = int(train_size * 0.1)
50        train_size = train_size - val_size
51
52        train_dataset, val_dataset = random_split(dataset, [train_size,
    val_size])
53        val_dataset.dataset.transform = DATA_TRANSFORMS['val']
54
55        dataloaders = {
56            'train': DataLoader(train_dataset, batch_size=config.param['
    batch_size'], shuffle=True, num_workers=4, pin_memory=True),
57            'val': DataLoader(val_dataset, batch_size=config.param['batch_size'
    ], shuffle=True, num_workers=4, pin_memory=True),
58            'test': DataLoader(test_dataset, batch_size=config.param['
    batch_size'], shuffle=False, num_workers=4, pin_memory=True),
59        }
60
61        model = VAE(
62            in_channels=config.in_channels,
63            latent_dim=config.param['z_dim'],
64            alpha=config.param['alpha'],
65            beta=config.param['beta'],
66            local_weight=config.param['local_weight'],
67            device=device
68        ).to(device)
69
70        optimizer = optim.Adam(model.parameters(), lr=config.param['lr'])
71        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
72                                                               mode='min',
73                                                               patience=3,
74                                                               factor=0.1,
75                                                               verbose=config.
    verbose)
76
77        model.fit(
78            dataloaders=dataloaders,
79            optimizer=optimizer,
80            scheduler=scheduler,
81            num_epochs=config.num_epochs,
82            early_stopping=config.early_stopping,
83            verbose=config.verbose
84        )
85
86        print('Training complete.')
87        torch.cuda.empty_cache()
88
89        state_dict = torch.load(model.model_path)
90        model.load_state_dict(state_dict)
91
```

```
92         predicted_labels, test_image_names, latent_vectors =
    clustering_and_f1_score(model, test_dataset)
93 
94         results_df = pd.DataFrame(
95             data={
96                 "Image Name": test_image_names,
97                 "Cluster Label": predicted_labels,
98             }
99         )
100 
101        results_df.to_csv(f'results/clustering_results_run_{run+1}.csv', index=
    False)
102 
103        max_common_class, max_common_count, total_common, num_in_max_class =
    compare_classes_with_txt(test_image_names, predicted_labels)
104 
105        metrics_df = pd.concat([metrics_df, pd.DataFrame({
106        'Run': [run+1],
107        'Max Common Class': [max_common_class],
108        'Max Common Count': [max_common_count],
109        'Number in Max Class': [num_in_max_class],
110        'Total Pothole': [total_common]
111    })], ignore_index=True)
112 
113    metrics_df.to_csv(f'results/clustering_metrics_all_runs.csv', index=False)
114 
115    average_metrics_df = pd.DataFrame(metrics_df.mean()).transpose()
116    average_metrics_df.columns = metrics_df.columns
117    average_metrics_df.to_csv(f'results/clustering_metrics_average.csv', index=
    False)
118 
119 
120 if __name__ == "__main__":
121 
122    main()
```

Listing 11: Finding average