POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Anomaly Detection in GUIs applied to websites

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Davide Volta**

Student ID: 968027
Advisor: Prof. Piero Fraternali
Academic Year: 2022-23

Anomaly Detection in GUIs applied to websites

Davide Volta

April 2024

Contents

1 Abstract

This thesis presents a novel approach to anomaly detection in Graphical User Interfaces (GUIs) of websites, with a focus on detecting visual anomalies that deviate from the GUI's expected appearance. Utilizing a combination of image segmentation and anomaly detection models, the study specifically targets the dynamic and complex dashboard page of ABB's Energy Manager web application. By segmenting the webpage into core components and applying anomaly detection models like PaDiM, the research demonstrates enhanced anomaly detection performance compared to traditional methods. The dataset, created through automated processes, includes screenshots of both normal and artificially induced anomalous states, facilitating the training and validation of the models. The findings highlight the potential of applying image-based anomaly detection techniques to improve the reliability and user experience of web applications by automatically identifying and addressing visual inconsistencies.

2 Introduction

The idea behind this work was born from a simple question: is there a way to automatically detect visual anomalies on a website? We should first define what we mean by *visual anomaly*. A *visual anomaly* is a visible difference in a *GUI (Graphical User Interface)* when compared to its usual appearance recorded over a certain amount of time, which could mean a few minutes or even a month, depending on the amount of non-anomalous variation that the GUI can exhibit (i.e. how dynamic its content is).

The main target for this work was a web application developed by ABB called *ABB Ability Energy Manager*, Energy Manager for short from now on. ABB Ltd. is a Swedish-Swiss multinational corporation whose main industry is electrical equipment such as breakers; Energy Manager is a web-based tool that allows users to monitor the status of their electrical systems, including charts and real-time data. We focused on its *Dashboard* page which is the most dynamic and complex in terms of layout: it is made of widgets that a user can add, remove, and shuffle freely.

Our visual anomaly detection method consists of segmenting the web page into its core components (in this specific case, that would be mainly individual widgets and buttons) via a simple heuristic and then training known anomaly detection models (such as PaDiM) on pictures of those components. This leads to better results compared to running the anomaly detection model directly on the whole screenshot.

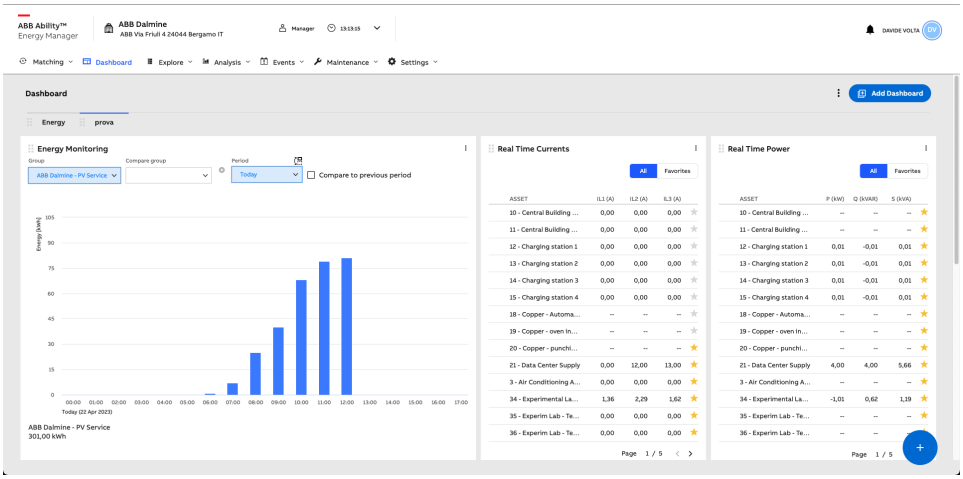

Figure 1: Screenshot of a dashboard in Energy Manager

3 Related work

The first thing we did was to look into the existing literature on the topic: the keywords were *GUI testing*, *one-class classification*, and *anomaly detection in images*. The findings are presented in the following subsections.

We feel like we should emphasize that during our research, we also stumbled upon a product called *Applitools Eyes* (`https://applitools.com/platform/eyes/`): a free trial allowed us to attest that it indeed seems to do what we need (i.e. you can upload screenshots, mark them as anomalous or non-anomalous, and it will learn to detect anomalies), but it is pretty costly and there is no clear documentation on how exactly it works. We want to make it clear that in no way, shape, or form did we plagiarize their product nor perform any kind of reverse engineering, as we wanted to devise an original approach to the problem.

3.1 GUI testing

GUI testing refers to the act of interacting with GUIs to check whether they behave as expected. The main (and also most recent) source of info on this topic that I could find is *"Visual GUI testing in practice: An extended industrial case study", Garousi et al.* [16]. In this paper, the authors recap the history of automated GUI testing tools, for which they identify 3 distinct generations:

1. 1st generation tools: these rely on the XY coordinates of elements on the screen to locate and test GUI components.

2. 2nd generation tools: this generation comprises the most used tools as of today, such as Selenium, which can locate elements by other means than their fixed location (e.g. Selenium can use the *id* of an HTML element, or its *class*).

3. 3rd generation tools: this is where computer vision techniques come into play; these tools can "see" and recognize objects on the screen by their appearance. This is often referred to as "Visual GUI testing" (VGT for short).

As the authors say, VGT has garnered significant attention in recent years, leading to numerous empirical studies: these primarily compare popular tools like Sikuli and JAutomate. Researchers have addressed diverse research questions, including the applicability of VGT in industrial contexts, the advantages and disadvantages of VGT for regression testing, and potential enhancements to VGT tools. While positive experiences have been reported, studies have also highlighted challenges, limitations, and issues faced.

The following sections contain a brief introduction to Sikuli and JAutomate; even though neither of the two completely solves our problem, an integration of these tools with our methods might result in an even better GUI testing method (see Conclusions).

3.1.1 Sikuli

Sikuli "allows users to take a screenshot of a GUI element (such as a toolbar button, icon, or dialog box) and query a help system using the screenshot instead of the element's name" (*"Sikuli: Using GUI Screenshots for Search and Automation", Yeh, Chang, and Miller* [47]).

Sikuli presents the user with two services:

1. Sikuli Search: this allows the user to run a search for a screenshot of a GUI element against a database of screenshots

2. Sikuli Script: users can write Python scripts that use screenshots as if they were variable (e.g.: you can write something like

```
find(<screenshot>).inside().find(<other screenshot>)
```

to check whether a given element is inside another one.

While this tool may sometimes be more immediate to use than something like Selenium, when content is expected to be highly dynamic and using methods based on the code structure to find elements might not always be reliable (e.g. HTML IDs may change at any given time), it is still not a solution to the problem of deciding whether the GUI under exam is anomalous or not in a completely automated manner.

3.1.2 JAutomate

JAutomate is a VGT tool that combines "image recognition with record and replay functionality for high system-level test automation performed through the system under test's graphical user interface." (*"JAutomate: A Tool for System- and Acceptance-test Automation", Alégroth, Nass, and Olsson* [4]).

R&R (Record and Replay) is a very common testing methodology where interactions are first recorded and saved in a script file and later replayed to check for regressions, though, as the authors say, "[...] The recording can be done in several ways, e.g. using references to the SUT's backend or by using exact coordinates on the SUT's GUI. However, both of these methods suffer from limitations that, once again, require high maintenance, e.g. reference-based R&R is fragile to API or even code change, whilst coordinate-based R&R is fragile to GUI layout change. Hence, R&R does not fulfill all of the industry's needs for a robust, flexible, high system-level, automated test technique."

The answer to that is using image recognition to locate GUI elements during the replay step. This "perceivably lowers the automation costs, since scripts can be recorded during regular manual test case execution, but retain all the benefits of the image recognition based playback, e.g. imperviousness to GUI layout change. Changes to the bitmap graphics of the GUI are instead what imposes the most amount of script maintenance. However, this maintenance can be done at a low cost in JAutomate which supports simple swapping of images within the scripts. Thus, JAutomate perceivably fills the gap in the industry for a high-level, cost-effective, flexible, and robust test automation tool."

Compared to Sikuli, the upfront cost is a lot lower (there is no need to write code, even though in some situations that might make more sense), but it is still not completely automated: someone needs to record all the interactions and the related expected outcomes.

3.2 Anomaly detection in images

In Visual Anomaly Detection for Images: A Survey*, Yang et al.* [45], the authors provide a comprehensive survey of the classical and deep learning-based approaches for visual anomaly detection in the literature.

Anomaly detection approaches try to find patterns in the training data and then try to detect deviations from those patterns in newly observed data; in our case, the training data would be normal screenshots, while the newly observed data would be potentially anomalous screenshots. This is usually called *visual anomaly detection* or *image anomaly detection*.

Based on the availability of abnormal samples, anomaly detection approaches can be classified as either *supervised* or *unsupervised*. Supervised approaches rely on abnormal data for the training process: as an example, you could build a classifier that learns to classify images as either anomalous or non-anomalous; unsupervised approaches only learn from normal samples. In the field of image anomaly detection, abnormal samples are often not easy to come by: the amount of possible variation can be so high that it would be unreasonable to generate samples of all possible anomalies, which means that any kind of supervision-based approach would be incomplete; thus, the focus shifts to unsupervised approaches.

Image anomaly detection methods can also be classified based on whether their goal is to classify images (*image-level anomaly detection*) or locate and highlight anomalies (*pixel-level anomaly detection*).

There is also a clear distinction between pre-deep learning approaches, based on feature extraction, statistical analysis, and traditional machine learning, and post-deep learning approaches, which are based on neural networks, more specifically *Convolutional Neural Networks (CNN)*.

3.2.1 Image-level anomaly detection

Density estimation The density estimation method first estimates the probability distribution model of the normal image features and then tests whether the newly observed image is abnormal or normal by comparing its features against the established distribution. This is based on the assumption that anomalous images will show features that don't fit the learned distribution.

Density estimation techniques traditionally encompass parametric methods like the Gaussian model and Gaussian mixture model. Non-parametric methods, such as the nearest neighbor and kernel density estimation, are also used. However, to accurately estimate a density, a significant volume

of training samples is necessary. This becomes especially challenging when dealing with high-dimensional data like images. Furthermore, these classical models often lack scalability.

Deep generative models have emerged as promising tools to determine the probability distributions of high-dimensional data. But, like their traditional counterparts, they require vast amounts of training data. Their robustness is also in question, making their performance for anomaly detection inconsistent. Through extensive testing, the researchers in [28] discovered that prevalent deep generative models, including the Variational Autoencoder (VAE) and the Glow flow model, struggle with even basic image anomaly detection tasks.

We briefly experimented with this method, hoping it would be effective for our problem. However, employing a Variational Autoencoder (VAE) model, yielded disappointing results. Our findings mirrored those in [28], where these models underperformed in simple image anomaly detection. Given the short duration of our trial, we concluded that this approach might not be the best fit for our needs.

Image reconstruction Image reconstruction approaches in anomaly detection involve mapping images to low-dimensional vector representations (latent space) and finding an inverse mapping to reconstruct the original image. Autoencoders, a type of neural network introduced by Hinton [22], play a crucial role in these methods, compressing input data through a narrow hidden layer and regenerating it while retaining non-redundant information. Autoencoders were first applied to anomaly detection by Japkowicz et al. [23], who recognized that redundant information in normal data might not be redundant in abnormal data and vice versa. Subsequently, deep autoencoders were introduced for high-dimensional data anomaly detection by Sakurada et al. [35], and Monte Carlo sampling was proposed to estimate reconstruction probabilities for scoring anomalies [5].

To enhance anomaly detection performance, some methods leverage both the distribution in the latent space and reconstruction errors of deep autoencoders. Gaussian mixture models are used to estimate the distribution of autoencoder latent codes in [52], while auto-regressive neural networks model the probability distribution of latent codes in [1]. Memory units for the autoencoder's latent code are introduced in [18] to represent the latent distribution.

Another approach suggests increasing the difficulty of image reconstruction by applying transformations to input images, such as color removal or

geometric transformations, before training the autoencoder to reconstruct the original input image with the transformed input [46], [36]. This method effectively raises the reconstruction difficulty for abnormal images, leading to larger reconstruction errors and improved anomaly detection performance.

Schlegl et al. [37] pioneered the use of Generative Adversarial Networks (GANs) for visual anomaly detection. This approach involves training a GAN model on normal images and detecting anomalies by calculating the difference between a test image and the closest normal image, determined through an iterative optimization process. However, this iterative search process can be inefficient.

Some approaches combine image reconstruction with adversarial training to enhance the efficiency and performance of visual anomaly detection. Sabokrou et al. [34] used a model comprising a generator (convolutional autoencoder) and a discriminator, optimizing it with mean square error loss and adversarial loss. Akćay et al. [3] introduced an encoder to measure the difference between the latent code of the reconstructed image and the input image's latent code as an anomaly indicator. Zenati et al. [51] employed BiGAN for image anomaly detection, while [31] proposed regularization techniques for the latent code's feature distribution in the joint model of autoencoder and adversarial network.

To address the robustness issue with using the discriminator's output as the anomaly score in GAN-based models, a new training strategy was proposed in [49] to improve the stability of the discriminator's probability output.

One-class classification A classification problem is a very common task in the realm of machine learning: the goal is to tell which class(es) an object belongs to. One-class classification is a variant where there is only one class and the training data set only contains instances of objects that do belong to that class. This comes up

Going back to the topic of this paper, if we simplify our goal from "detecting anomalies on a GUI" to "deciding whether a GUI is anomalous or not", it indeed becomes a one-class classification problem: the class is "normal screenshots" and the training data set would contain screenshots of the GUI under exam when it is not displaying any kind of visual anomaly. Combining one-class classification with our segmentation technique may also allow us to locate anomalies up to a certain degree of precision.

In *"Learning Deep Features for One-Class Classification", Perera and Patel* [32] the authors present "a novel deep-learning based approach for

one-class transfer learning in which labeled data from an unrelated task is used for feature learning in one-class classification".

Self-supervised classification In visual anomaly detection using self-supervised classification, the key factor is the strong capability of self-supervised learning in representing visual features. Self-supervised learning involves using auxiliary tasks to extract supervision information from large-scale unsupervised data. This information is then used to train deep convolutional neural networks to learn visual representations, which can be applied to various tasks, including image classification, object detection, and anomaly detection.

The concept behind self-supervised classification for anomaly detection is that models trained in a self-supervised manner can capture unique and significant characteristics of normal samples. These learned representations not only encompass low-level features like color and texture but also higher-level attributes such as location, shape, position, and direction. By exclusively learning from normal samples, these models become proficient at identifying abnormal samples lacking these characteristics.

One approach, presented by Golan et al., is RotateNet [17], which teaches a neural model to distinguish different geometric transformations applied to normal images, making it the first self-supervised classification model for image anomaly detection.

Dan et al. proposed a more challenging self-supervised classification task, where the model not only distinguishes image rotations but also classifies image translations [21].

Tack et al. introduced contrastive learning, a type of self-supervised classification method, for image anomaly detection, showing superior performance compared to other self-supervised methods [39].

Sehwag et al. further enhanced and extended contrastive learning for anomaly detection, achieving state-of-the-art performance in image-level anomaly detection [38].

3.2.2 Pixel-level anomaly detection

Unsupervised pixel-level anomaly detection approaches can be broadly categorized into two groups: image reconstruction and feature modeling. It's worth noting that all methods designed for image-level anomaly detection can typically be applied to pixel-level anomaly detection by segmenting the entire image into multiple patches and then performing anomaly detection

at the patch level. However, image-level anomaly detection techniques primarily focus on the semantic information of the entire image. The semantic information within individual image patches may be relatively weak, casting doubt on the suitability of image-level detection for pixel-level tasks.

Image reconstruction Image reconstruction is a common method for detecting anomalies, typically involving the use of deep convolutional autoencoders. These autoencoders first learn to reconstruct normal images, and potential anomalies are identified by assessing the pixel-level differences between the input image and its reconstructed version. Metrics like pixel-level L2-distance [26] and Structural Similarity Measure (SSIM) [20] are used to quantify these disparities. However, reconstruction-based methods assume that autoencoders trained on normal images cannot generalize well to abnormal ones.

Deep generative models, such as Variational Autoencoders (VAEs) [24] and Generative Adversarial Networks (GANs) [19], can also serve as reconstruction models. For instance, VAE-GAN [6] and similar models use GANs for adversarial training to enhance reconstructed image quality. During testing, pixel-level L1 distance is employed to score abnormality. Additionally, deep generative models-based anomaly detection methods can leverage reconstruction probability [7], [5] or likelihood score [34], [40] as supplementary anomaly measures.

Some approaches differ by not comparing input and reconstructed images but by calculating differences between the input image and its nearest normal counterpart. AnoGAN [37] and similar methods first train a GAN model with normal images and then detect anomalies by estimating differences between a test image and its nearest normal image, determined through an iterative optimization process. However, these methods can be inefficient in practical applications due to their iterative search process.

Furthermore, certain methods aim to make reconstruction more challenging by degrading input images before having the autoencoder reconstruct them. This information degradation process increases the difficulty of reconstructing abnormal images, thereby enhancing the anomaly score and improving detection performance [43], [50].

While image reconstruction is an intuitive approach for pixel-level anomaly detection, it faces challenges in generating high-quality images for comparison. Issues like regenerating sharp edges and complex textures often result in large reconstruction errors in these areas, leading to false abnormal alarms.

Feature modeling Feature-based methods for image anomaly detection focus on detecting anomalies in the feature space rather than the image space. These methods aim to construct effective representations of local image regions using either handcrafted features or features learned by neural networks [42], [10], [11], [13], [12], [29], [9], [44]. Machine learning models like sparse coding, Gaussian mixed models, and K-means clustering are then used to model the feature distribution of normal images. When testing for anomalies, if the features of a local region in the test image deviate from the modeled feature distribution, that region is labeled as abnormal.

To enhance detection performance, these methods often employ a multi-scale model ensemble strategy that combines results from models trained on different image region sizes [42], [13]. However, this approach involves dividing the image into multiple small patches for modeling and detection, which can be time-consuming, especially when deep neural networks are used to extract features. Additionally, since each local region is evaluated independently, these methods may not effectively leverage the global context information of the image.

A benchmark dataset for evaluating unsupervised image anomaly detection algorithms, MVtec AD, was recently introduced by Bergmann et al. [8]. This dataset contains various texture and object categories with more than 70 different types of anomalies. Bergmann et al. evaluated multiple methods based on image reconstruction and feature modeling on this dataset, highlighting the need for improvement. They later proposed an unsupervised anomaly detection method using a student-teacher distillation framework, achieving promising results by leveraging transferred deep convolution features and feature regression.

Some recent feature-based methods aim to encode spatial context information using pre-trained deep hierarchical convolutional features, showing potential for pixel-level anomaly detection and segmentation [44], [48], [14]. These methods make use of hierarchical features from networks like VGG19 and ResNet18, employing mechanisms such as feature reconstruction and feature matching for anomaly detection.

Another research direction in visual anomaly detection involves using gradient-based attention mechanisms like Grad-CAM and interpretable deep generative models [25], [41]. These approaches focus on locating abnormal regions in images using attention mechanisms and gradient-based visual interpretation methods.

3.2.3 PaDiM

PaDiM [15], which stands for Patch Distribution Modeling, is an innovative approach for anomaly detection and localization in images, with a particular focus on applications in industrial inspections. The primary objective of PaDiM is to automatically identify and pinpoint anomalies or unexpected patterns in images, enabling constant quality control without the need for manual intervention.

What sets PaDiM apart from other methods is its use of a pre-trained Convolutional Neural Network (CNN) for feature extraction. It leverages this CNN to describe each position in an image patch as a multivariate Gaussian distribution. Additionally, PaDiM takes into account the correlations between different semantic levels of the CNN, enhancing its ability to detect and localize anomalies effectively.

PaDiM has proven to be highly efficient and effective in anomaly detection and localization tasks, surpassing existing state-of-the-art methods on datasets like MVTec AD and ShanghaiTech Campus. Importantly, PaDiM's efficiency is notable, as it maintains low time and space complexity during testing, making it well-suited for practical industrial applications.

The following paragraphs briefly present how PaDiM works.

Embedding Extraction: In this step, pre-trained Convolutional Neural Networks (CNNs) are utilized to extract relevant features for anomaly detection. Instead of optimizing a neural network from scratch, the method employs a pre-trained CNN to generate patch embedding vectors. The process of generating these patch embeddings is similar to that used in SPADE. During training, each patch from normal images is associated with spatially corresponding activation vectors from the pre-trained CNN activation maps. These activation vectors come from different layers of the CNN and are concatenated to create embedding vectors. This combination of activations at different layers allows the method to capture fine-grained and global context information. Since activation maps have a lower resolution than the input image, multiple pixels share the same embeddings, forming pixel patches with no overlap in the original image resolution. Thus, the input image is divided into a grid of $(i, j))$ positions, each associated with an embedding vector (x_{ij}) computed as described. To enhance efficiency, the method explores the possibility of reducing the size of these patch embedding vectors, finding that random dimensionality reduction is more efficient than classic *Principal Component Analysis (PCA)*. This reduction significantly decreases the complexity of the model for both training and testing while maintaining high performance.

Learning of the Normality: In this step, the method learns the normal image characteristics at different positions. It achieves this by assuming that the set of patch embedding vectors at each position (i, j) follows a multivariate Gaussian distribution $N(\mu_{ij}, \Sigma_{ij})$. Here, μ_{ij} represents the sample mean of the patch embedding vectors, and Σ_{ij} is the sample covariance matrix. To ensure that Σ_{ij} is full rank and invertible, a regularization term ϵI is introduced. This modeling is applied to each possible patch position, effectively creating a matrix of Gaussian parameters. Importantly, since the patch embedding vectors capture information from different semantic levels, each estimated multivariate Gaussian distribution $N(\mu_{ij}, \Sigma_{ij})$ captures information from different levels as well, including inter-level correlations. This modeling contributes to improved anomaly localization performance.

Inference: Computation of the Anomaly Map: In the inference phase, the method uses the Mahalanobis distance to calculate an anomaly score for each patch in a test image at position (i, j). The Mahalanobis distance measures the distance between a test patch embedding and the learned distribution $N(\mu_{ij}, \Sigma_{ij})$. Specifically, the Mahalanobis distance $(M(x_{ij}))$ is computed for each position (i, j) in the test image. This process generates a matrix of Mahalanobis distances (M) that forms an anomaly map. Regions with high scores in this map indicate anomalous areas in the test image. The final anomaly score for the entire test image is determined by selecting the maximum value from the anomaly map. Notably, this method does not suffer from scalability issues associated with K-NN-based methods, as it doesn't require the computation and sorting of a large number of distance values for each patch, ensuring computational efficiency.

3.3 Anomaly detection in GUIs

Matila Noblia has explored the feasibility of automated visual inspection, specifically concentrating on graphical user interfaces (GUIs) [30]. This research focuses on detecting two types of anomalies within GUIs: missing GUI components and misplaced GUI components, driven by insights from prior research.

Their method employs a Convolutional Neural Network (CNN) architecture based on a previously established design [27]. The architecture consists of convolutional layers with ReLU activation functions, max-pooling layers, and fully connected layers. It utilizes a learning rate update schedule to fine-tune the network's performance over epochs.

The network's hyperparameters include a dropout rate of 0.5, a batch size of 64, and an initial learning rate $(\epsilon 0)$ of 0.001. The learning rate

decreases based on the number of epochs and a parameter (i_c), as defined in Equation (3.1). Batch normalization is applied after each convolutional layer.

The dataset used for training and evaluation comprises screenshots from various apps that share a consistent underlying design, despite variations in color and content. Both correct GUI looks and anomalous instances are included. Anomalies are introduced through image manipulation, resulting in two types: missing GUI components (type 1) and misplaced GUI components (type 2).

To address the class imbalance issue inherent in anomaly detection, an anomalous dataset is manually generated, ensuring the network learns from anomaly examples. The final dataset is split into training and evaluation sets, with the training set further divided into training and validation subsets.

The findings of this report suggest that the network demonstrates the ability to distinguish between the correct GUI appearance and individual types of anomalies, such as missing GUI components or misplaced GUI components. However, the network's performance decreases when it attempts to differentiate between the two types of anomalies, as well as the correct GUI appearance simultaneously. This decline in performance may be attributed to the subtle definition of anomaly type 2, which encompasses a broad range of anomalies, including misalignments and components placed elsewhere on the screen. This complexity appears to affect the network's performance when trained and evaluated on all three classes.

The nature of the dataset, which was manufactured by introducing graphical anomalies manually, presents certain limitations. The choice of anomalies was based on research into common visual anomalies in open software GUIs. However, the dataset's class imbalance and potential bias resulting from manual anomaly introduction raise concerns about its representativeness of real-world scenarios. Gathering real-world data examples for training and evaluation could improve the generalizability of the results but may pose challenges due to the relative rarity of graphical errors in GUIs.

Our goal instead is to not rely on anomalous images for training in order to overcome one of the issues in their work.

4 Methods

4.1 Overview of the testing process

We divided the whole process into 3 different steps:

1. Acquire images (*acquire*)
2. Train models (*train*)
3. Check the web page for anomalies (*verify*)

4.1.1 Acquire images

The *acquire* step consists of collecting screenshots of the web page and the individual components, together with binary masks that highlight the anomalous regions (masks and related problems are covered in detail in *Ground truth generation*. Normal images are called *normal* and images where an error has been artificially generated are called *abnormal*.

The data acquisition program runs the following steps for each web page:

1. Check if we have reached the target dataset size. If yes, stop.
2. Load the webpage and wait for it to load completely.
3. Take a screenshot.
4. Generate a blank binary anomaly mask the size of the webpage.
5. Find elements of interest and, for each element:
   (a) Take a screenshot
   (b) Randomly alter it in some way.
   (c) Generate a difference mask between the pre-alteration version and the post-alteration version.
   (d) If the amount of different pixels is lower than the set threshold, discard this element and go to the next one.
   (e) Binarize the difference mask.
   (f) Store the normal screenshot, the altered one, and the binary anomaly mask.
   (g) Paste the generated binary anomaly mask on the existing full-page mask.

6. Take a screenshot of the page after all elements of interest have been altered.

7. Store the pre-alteration full-page screenshot, the altered full-page screenshot, and the resulting composite binary anomaly mask.

Figure 2 shows the generated directory structure. Simply put, a directory is generated for each website, which will contain directories for all the analyzed pages. Every page directory is made of a *fullpage* directory (which holds screenshots and masks for the complete page) and several other directories for the detected components.

An issue that arose during the data acquisition step is the following: how can we store our models in such a way that when running verification we can match components to the corresponding models?

Our solution was quite brutal, in a way, but also effective: all of our processing was performed at a fixed resolution of 1920x2160 so that components could be identified by their X and Y coordinates. Of course, this means that if anything changes its position our program can no longer work and needs to be retrained.

We thought about possible alternative approaches to this problem and came up with 3 leads to follow for future research:

1. **Hash/fingerprint generation:** we searched for methods that, given the HTML markup, could generate some sort of hash and/or fingerprint that we could then use as an identifier. Normal hashing algorithms such as SHA256 do not fit our purpose, since the code for a web page can, of course, vary, even when visible content stays the same. *Fuzzy hashing* could be a solution though. Fuzzy hashing, also known as similarity hashing or approximate hashing, is a technique used to determine the similarity or likeness between two data sets or objects, even when they are not exact duplicates. It's particularly useful for identifying similar content or detecting near-duplicate files in large datasets. Fuzzy hashing works by generating a fixed-size hash value (usually a numerical or hexadecimal representation) for each input, and similar inputs produce hash values that are close to each other.

2. **Fuzzy string matching:** Fuzzy string matching, often called approximate string matching, is a technique employed to compare two strings for their similarity or dissimilarity. It is particularly useful when dealing with data that may contain errors, typos, or variations. This approach assigns a similarity score or distance metric to quantify

the degree of similarity between two strings, making it valuable in a range of applications.

In fuzzy string matching, two strings are compared, typically referred to as the "target" string (the reference) and the "query" string (the one to be compared). The objective is to assess how closely the query string resembles the target string.

To achieve this, a numerical score or distance metric is computed, reflecting the similarity between the two strings. This score indicates the number of operations required to transform one string into the other, encompassing actions like insertions, deletions, substitutions, or transpositions.

For decision-making purposes, a threshold value is established. If the computed score falls below this threshold, the strings are deemed similar or matching. Conversely, if it surpasses the threshold, they are considered dissimilar. Our main problem with any fuzzy string matching method was establishing the correct threshold, which would probably require its own machine learning model to take into account all possible cases where the HTML markup changes but the visual result remains identical.

3. **Object detection:** Object detection is a computer vision task used to identify and locate objects in images or video. It's essential for applications like autonomous driving and surveillance.

   The process involves extracting image features, determining object locations with bounding boxes, classifying objects into predefined categories, and resolving overlapping boxes. These steps collectively enable accurate object detection.

   Two types of models are commonly used: one-stage detectors (e.g., YOLO [33]) process the entire image at once, providing real-time performance. In contrast, two-stage detectors (e.g., Faster R-CNN) first propose regions of interest and then classify them, offering higher accuracy. It would be theoretically possible to train an object detection model during our *training* step, and then use that model to locate components when running the anomaly detection model. We tried YOLO on our dataset and it did work, but for the purpose of this paper, we settled on our initial choice of using the XY coordinates to better focus our efforts on the core of our research, which is anomaly detection. Also, this solution would greatly increase training times.

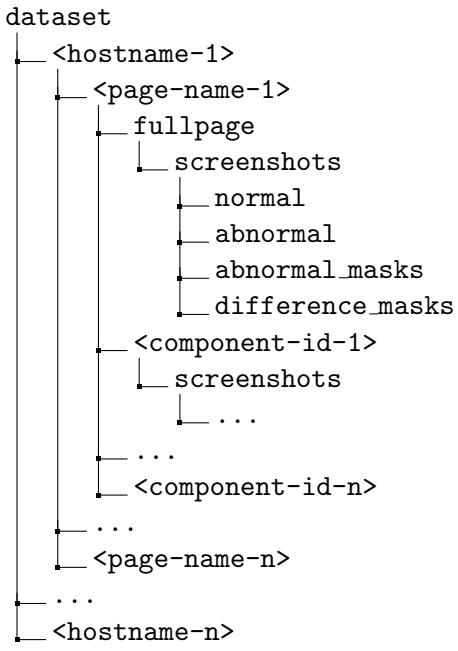

Figure 2: Dataset directory structure

4.1.2 Train models

During the training step, the program trains a model for each component found in the dataset. Some validation samples (*screenshots*) plus a summary .csv file are also generated. The resulting directory structure is illustrated in Figure 3.

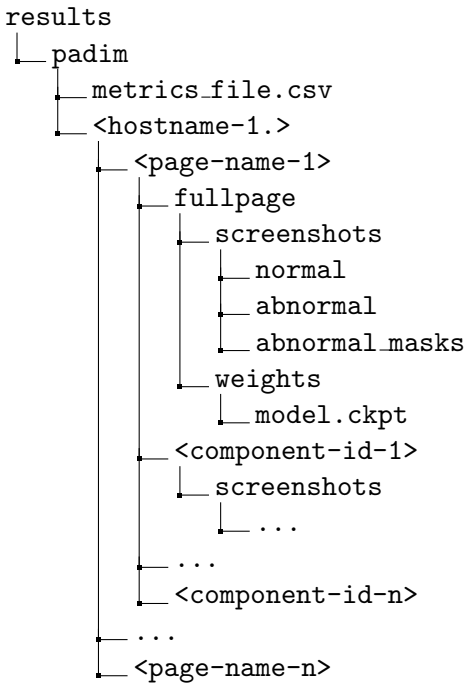

Figure 3: Training output

4.1.3 Monitor webpage

The *verify* step will not be examined in this document, since we are focusing on the correctness of our method and that can be done by looking at the training results.

4.2 Ground truth generation

One of the requirements that we decided on pretty early on was that the whole system needed to be as autonomous and flexible as possible, which meant that manually taking screenshots and drawing masks was out of the question.

In addition to the set of regular images, the training process also requires a set of anomalous images paired with anomaly masks, so that image-wise and pixel-wise performance metrics can be calculated.

Since the anomaly detection models we can use are all unsupervised, the results are not directly influenced by anomalous images and the related anomaly masks, but metrics are. AUROC and F1 scores can get pretty low in 2 cases:

1. Anomalous images are not actually anomalous (see Figure 4)
2. The anomaly mask is not accurate enough (see Figure 5)

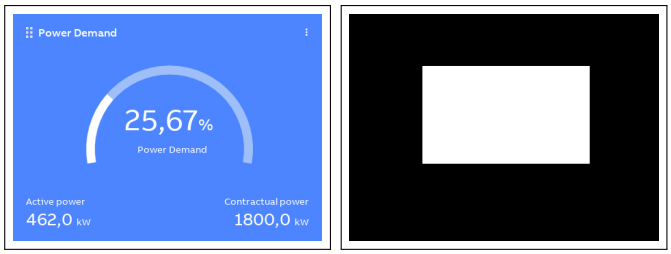

Figure 4: Altered component and the generated anomaly mask (white regions are considered anomalous). This is a case where the result is not anomalous at all. The changed element was probably fully occluded by other elements.

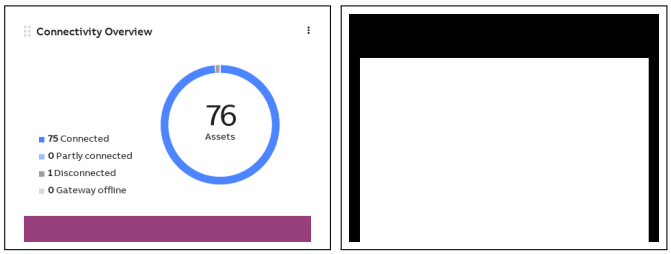

Figure 5: Altered component and the generated anomaly mask (white regions are considered anomalous). In this case, the mask shows an anomalous region bigger than what we can see from the image. The changed element was probably partially occluded by others.

The anomalous images are automatically generated by simply picking an HTML element in the component and changing some of its properties (e.g. *background-color*).

As for anomaly masks, our first approach was to simply draw a rectangle where the changed element was, but this presented a big flaw: it did not work where there was occlusion (i.e.: HTML elements overlapping), which meant that a lot of the collected "anomalous" images fell into one of the two cases presented above (again, see Figures 4 and 5). We therefore revised the anomaly generation process and based it on pixel-wise difference: any

difference greater than 0 results in a white pixel. An example of such a mask is shown in Figure 6.

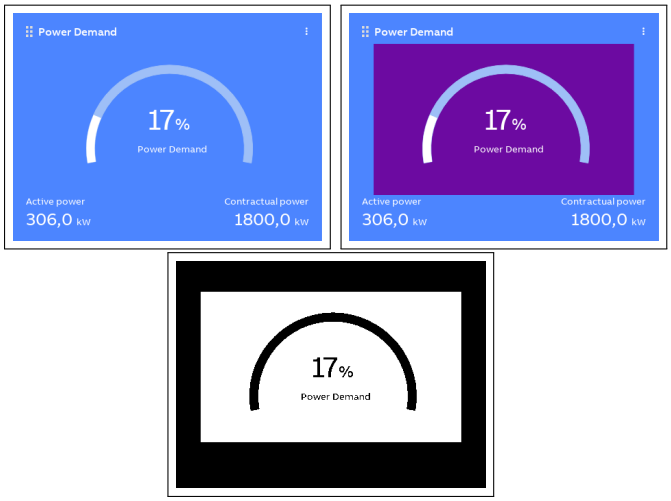

Figure 6: Normal screenshot, altered screenshot, and the resulting difference mask. As you can see, this mask generation method is more accurate and only highlights visible differences (in this case, the bar and the text did not change so they are ignored).

These difference masks solve problem number 2.

Also, we avoid storing abnormal screenshots that have a low difference value (< 0.025), measured as:

$$\frac{\#\text{ of different pixels}}{\#\text{ of pixels}}$$

This solves problem number 1.

4.3 Anomaly detection

For the actual anomaly detection step, we relied on *anomalib* version 0.3.7 (at the time of writing, the latest available version is 0.4.0) [2] (`https://github.com/openvinotoolkit/anomalib`). From the README in their repository:

> Anomalib is a deep learning library that aims to collect state-of-the-art anomaly detection algorithms for benchmarking on both

public and private datasets. Anomalib provides several ready-to-use implementations of anomaly detection algorithms described in the recent literature, as well as a set of tools that facilitate the development and implementation of custom models. The library has a strong focus on image-based anomaly detection, where the goal of the algorithm is to identify anomalous images, or anomalous pixel regions within images in a dataset.

These algorithms were designed for pictures of real-life objects, such as parts coming out of a production line; they were not really made to work with artificial images, as is the case here. Running any of these directly on a screenshot of the full web page gave lackluster results (you can see a comparison with our method in section 5.2.

We chose to use *PaDiM* (*Patch Distribution Modeling*, [15] - see 3.2.3 for a summary of how PaDiM works), which was the simplest and also the fastest to train (since it does not actually need to fine-tune a neural network, as opposed to other algorithms such as *CFLOW-AD*). If you consider that our method may result in dozens of core components being identified, with each of those requiring its own anomaly detection model, this means that we are allowed to use a larger dataset while still keeping training times under a reasonable threshold.

At the end of the training process, anomalib will output the trained model, computed metrics (both on the test and validation sets), plus a visualization of the validation results for normal images as well as abnormal ones, as shown in Figures 7 and 8.

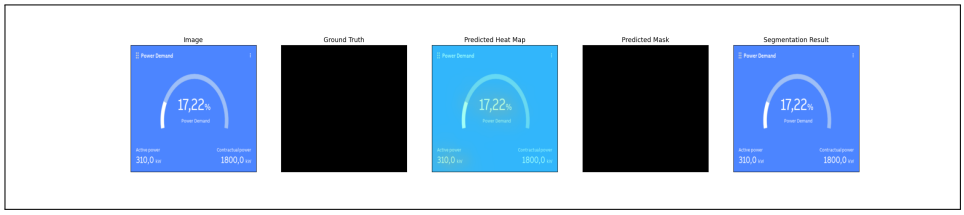

Figure 7: This example shows the algorithm's performance during validation on a normal image. From left to right, the normal image, the ground truth (i.e. our generated anomaly mask), a heat map of detected anomalies, and the anomaly segmentation result.

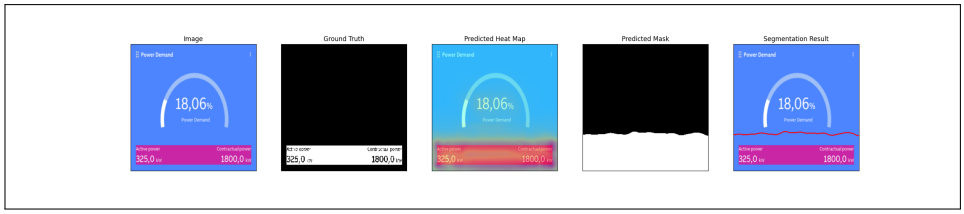

Figure 8: This example shows the algorithm's performance during validation on an anomalous image. From left to right, the anomalous image, the ground truth (i.e. our generated anomaly mask), a heat map of detected anomalies, and the anomaly segmentation result.

Here is the anomalib configuration file that we used in YAML format:

```yaml
dataset:
  name: screenshots
  format: folder
  path: dataset/<hostname>/<page name>/<component ID>/screenshots
  normal_dir: normal # name of the folder containing normal images.
  abnormal_dir: abnormal # name of the folder containing abnormal images.
  normal_test_dir: null # name of the folder containing normal test images.
  mask: difference_masks
  task: segmentation # classification or segmentation
  extensions: null
  split_ratio: 0.2 # ratio of the normal images that will be used to create a test sp
  image_size: 256
  train_batch_size: 4
  test_batch_size: 4
  num_workers: 8
  transform_config:
    train: null
    val: null
  create_validation_set: true
  tiling:
    apply: false
    tile_size: null
    stride: null
    remove_border_count: 0
    use_random_tiling: False
    random_tile_count: 16
```

```yaml
model:
  name: padim
  backbone: resnet18
  pre_trained: true
  layers:
    - layer1
    - layer2
    - layer3
  normalization_method: min_max # options: [none, min_max, cdf]

metrics:
  image:
    - F1Score
    - AUROC
  pixel:
    - F1Score
    - AUROC
  threshold:
    image_default: 3
    pixel_default: 3
    adaptive: true

visualization:
  show_images: False # show images on the screen
  save_images: True # save images to the file system
  log_images: True # log images to the available loggers (if any)
  image_save_path: null # path to which images will be saved
  mode: full # options: ["full", "simple"]

project:
  seed: 42
  path: ./results

logging:
  logger: [] # options: [tensorboard, wandb, csv] or combinations.
  log_graph: true # Logs the model graph to respective logger.

optimization:
  export_mode: null #options: onnx, openvino
```

```yaml
# PL Trainer Args. Don't add extra parameter here.
trainer:
  accelerator: auto # <"cpu", "gpu", "tpu", "ipu", "hpu", "auto">
  accumulate_grad_batches: 1
  amp_backend: native
  auto_lr_find: false
  auto_scale_batch_size: false
  auto_select_gpus: false
  benchmark: false
  check_val_every_n_epoch: 1 # Don't validate before extracting features.
  default_root_dir: null
  detect_anomaly: false
  deterministic: false
  devices: 1
  enable_checkpointing: true
  enable_model_summary: true
  enable_progress_bar: true
  fast_dev_run: false
  gpus: null # Set automatically
  gradient_clip_val: 0
  ipus: null
  limit_predict_batches: 1.0
  limit_test_batches: 1.0
  limit_train_batches: 1.0
  limit_val_batches: 1.0
  log_every_n_steps: 50
  max_epochs: 1
  max_steps: -1
  max_time: null
  min_epochs: null
  min_steps: null
  move_metrics_to_cpu: false
  multiple_trainloader_mode: max_size_cycle
  num_nodes: 1
  num_processes: null
  num_sanity_val_steps: 0
  overfit_batches: 0.0
  plugins: null
  precision: 32
  profiler: null
```

```
reload_dataloaders_every_n_epochs: 0
replace_sampler_ddp: true
sync_batchnorm: false
tpu_cores: null
track_grad_norm: -1
val_check_interval: 1.0 # Don't validate before extracting features.
```

5 Evaluation

5.1 Data set

The dataset we used to conduct the evaluation contained 827 screenshots for each of the 36 significant components identified, plus another 827 for the whole page. Every screenshot also has a corresponding generated anomalous version, plus a binary anomaly mask. The directory structure is presented in Figure 2).

Our program automatically split our dataset into a training portion (80%) and an evaluation one (20%).

5.2 Results

The results are presented in the following tables. The first row refers to the whole page (hence the ID *fullpage*), while the others refer to the identified components (whose ID is generated by concatenating their X and Y coordinates in this case).

5.2.1 Training

Component	Pixel AUROC	Pixel F1	Image AUROC	Image F1
fullpage	0.77	0.32	1.0	1.0
30.20	0.87	0.93	1.0	1.0
1642.20	0.94	0.72	0.87	0.91
17.152.040	1.0	1.0	1.0	1.0
20.83	0.69	0.68	0.86	0.91
150.84	0.71	0.74	0.88	0.91
280.84	0.74	0.73	0.88	0.91
392.84	0.72	0.7	0.86	0.91
507.84	0.7	0.68	0.85	0.91
615.84	0.75	0.74	0.87	0.91
764.84	0.71	0.69	0.85	0.91
202.36	0.98	0.97	1.0	1.0
232.29	0.95	0.91	1.0	1.0
452.040	1.0	1.0	1.0	1.0
3.092.040	1.0	1.0	1.0	1.0
4.262.040	1.0	1.0	1.0	1.0
5.462.040	1.0	1.0	1.0	1.0
7.572.040	1.0	1.0	1.0	1.0
8.472.040	1.0	1.0	1.0	1.0
9.852.040	1.0	1.0	1.0	1.0
11.662.040	1.0	1.0	1.0	1.0
493.39	0.96	0.76	1.0	1.0
644.39	0.95	0.79	1.0	1.0
744.41	0.97	0.98	1.0	1.0
18.052.070	1.0	1.0	1.0	1.0
1.697.160	0.72	0.76	0.88	0.91
1.662.163	0.69	0.77	0.91	0.91
48.217	0.97	0.78	1.0	1.0
35.264	0.65	0.32	0.62	0.91
497.264	0.8	0.52	0.77	0.91
1.420.948	0.82	0.54	0.73	0.91
958.264	0.87	0.49	0.98	0.98
1.420.264	0.89	0.46	0.98	0.98
35.606	0.73	0.3	0.65	0.91
958.948	0.89	0.46	0.94	0.94
351.290	0.8	0.51	0.75	0.91
4.971.290	0.87	0.62	0.8	0.91

5.2.2 Evaluation

Component	Pixel AUROC	Pixel F1	Image AUROC	Image F1
fullpage	0.77	0.34	1.0	1.0
30.20	0.87	0.93	1.0	1.0
1642.20	0.93	0.71	0.85	0.91
17.152.040	1.0	1.0	1.0	1.0
20.83	0.75	0.75	0.89	0.91
150.84	0.73	0.76	0.89	0.91
280.84	0.73	0.71	0.87	0.91
392.84	0.7	0.68	0.87	0.91
507.84	0.7	0.68	0.85	0.91
615.84	0.7	0.69	0.85	0.91
764.84	0.69	0.67	0.86	0.91
202.36	0.98	0.97	1.0	1.0
232.29	0.95	0.91	1.0	0.99
452.040	1.0	1.0	1.0	1.0
3.092.040	1.0	1.0	1.0	1.0
4.262.040	1.0	1.0	1.0	1.0
5.462.040	1.0	1.0	1.0	1.0
7.572.040	1.0	1.0	1.0	1.0
8.472.040	1.0	1.0	1.0	1.0
9.852.040	1.0	1.0	1.0	1.0
11.662.040	1.0	1.0	1.0	1.0
493.39	0.96	0.75	1.0	1.0
644.39	0.95	0.78	1.0	1.0
744.41	0.96	0.98	1.0	1.0
18.052.070	1.0	1.0	1.0	1.0
1.697.160	0.72	0.75	0.88	0.91
1.662.163	0.65	0.77	0.89	0.91
48.217	0.97	0.78	1.0	1.0
35.264	0.66	0.31	0.61	0.91
497.264	0.82	0.6	0.77	0.91
1.420.948	0.84	0.63	0.78	0.91
958.264	0.88	0.45	0.99	0.97
1.420.264	0.82	0.43	0.98	0.97
35.606	0.73	0.32	0.6	0.91
958.948	0.77	0.37	0.96	0.94
351.290	0.82	0.55	0.77	0.91
4.971.290	0.91	0.66	0.79	0.91

5.3 Discussion

We will first focus on *Image AUROC* and *Image F1*. These two values tell us how good our models are at classifying the image as either anomalous or non-anomalous. If our goal is to simply classify the page as "contains errors" or "does not contain errors", then the numbers tell us that there is no significant improvement obtained from working on individual components compared to the full web page: we already have a 1 in both metrics.

Things look different though when you observe *Pixel AUROC* and *Pixel F1* scores. These measure the ability to locate anomalies in the image. As you can see, the *fullpage* scored 0.77 and 0.32 respectively, but individual components scored mostly much better values, with some exceptions. This means that our method has a higher chance to successfully highlight anomalies in a web page; this is due to having separate models for each component, which reduces variability in the input and allows each model to be much more precise.

5.4 Limits to validity

Let's examine one of the components that didn't do too well in our evaluation: *35.606*.

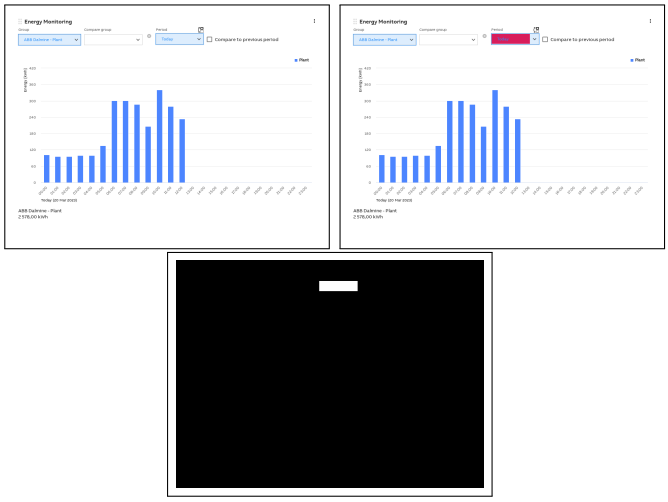

Figure 9: Sample data for component 35.606.

This component is made by some drop-down menus and, more prominently, a histogram, which generates a lot of variability in the output, thus

requiring the model to see more data when compared to other simpler and more static components. We therefore reasonably expect performance to improve as the dataset size increases. Also, we need to consider that the Energy Manager platform we ran our training against was experiencing some issues that caused long loading times and sometimes graphical bugs; while we put in place some mitigation measures to avoid capturing data that is clearly wrong, some of the screenshots still present issues that most definitely impact the accuracy of the result. The training process would ideally need to see only correct and valid images, but this is sometimes not technically feasible when using automation. A manual screening would certainly help, but we could not afford to do it for our research effort. At the same time, we pondered over this issue and thought that, if the test target behaves abnormally during training even with mitigation measures put in place, then it would be unreasonable to expect our system to work and a stable condition would need to be achieved before performing any kind of training.

6 Conclusions and future work

In conclusion, this method, which builds upon the foundation of Anomalib and PaDiM, has demonstrated its effectiveness in anomaly detection. It leverages the strengths of these existing techniques while addressing specific challenges and making what we believe are notable contributions to the field. However, as with any evolving technology, certain areas warrant further attention in future research endeavors.

Firstly, one crucial area for improvement lies in handling variable window sizes. The whole training is run at a specific resolution and using the resulting models on elements that differ in size might result in inaccuracies. Also, we rely on fixed locations to match an element to the corresponding model, which means that the test resolution needs to be the same as the training one. Developing mechanisms to dynamically adapt to different window sizes or effectively processing images with varying resolutions would enhance the method's adaptability and applicability across diverse scenarios.

Secondly, the method can benefit from ongoing efforts to dynamically match elements in test images to the learned models. As we just said, our current method of matching an element to its model via its page coordinates is very unreliable: any change in resolution and/or position of the detected elements would break it immediately. We believe this could be implemented by something rather rudimentary like cosine similarity or, maybe, another deep learning model dedicated to computing a similarity score between an

image and a given set of images. Pre-existing tools such as Sikuli and JAutomate could also help.

Lastly, leveraging provisional data obtained during training to filter out invalid content in test images presents a valuable avenue for future research. The data acquisition step currently runs based on the assumption that no error will occur and all screenshots taken are considered valid. For example, we could think about acquiring 100-200 images per element, perform training on this initial dataset, and then use the learned models to acquire more screenshots while filtering errors that may occur during the acquisition process, so that the dataset is as clean as possible.

In summary, while we believe we have achieved interesting results, addressing these challenges and exploring these avenues for future work could result in a more robust method that might see some adoption from people who want to monitor their website's appearance.

Bibliography

[1] Davide Abati et al. "Latent Space Autoregression for Novelty Detection". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019.

[2] Samet Akcay et al. *Anomalib: A Deep Learning Library for Anomaly Detection*. 2022. arXiv: 2202.08341 [cs.CV].

[3] Samet Akçay, Amir Atapour-Abarghouei, and Toby P Breckon. "Skip-ganomaly: Skip connected and adversarially trained encoder-decoder anomaly detection". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019, pp. 1–8.

[4] Emil Alégroth, Michel Nass, and Helena H. Olsson. "JAutomate: A Tool for System- and Acceptance-test Automation". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. 2013, pp. 439–446. DOI: 10.1109/ICST.2013.61.

[5] Jinwon An and Sungzoon Cho. "Variational Autoencoder based Anomaly Detection using Reconstruction Probability". In: 2015. URL: https://api.semanticscholar.org/CorpusID:36663713.

[6] Christoph Baur et al. "Deep Autoencoding Models for Unsupervised Anomaly Segmentation in Brain MR Images". In: *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*. Springer International Publishing, 2019, pp. 161–169. DOI: 10.1007/978-3-030-11723-8_16. URL: https://doi.org/10.1007%2F978-3-030-11723-8_16.

[7] Paul Bergmann et al. "Improving Unsupervised Defect Segmentation by Applying Structural Similarity to Autoencoders". In: *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. SCITEPRESS - Science and Technology Publications, 2019. DOI: 10.5220/0007364503720380. URL: https://doi.org/10.5220%2F0007364503720380.

[8] Paul Bergmann et al. "MVTec AD - A Comprehensive Real-World Dataset for Unsupervised Anomaly Detection". In: June 2019. DOI: 10.1109/CVPR.2019.00982.

[9] Paul Bergmann et al. "Uninformed Students: Student-Teacher Anomaly Detection With Discriminative Latent Embeddings". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2020. DOI: 10.1109/cvpr42600.2020.00424. URL: https://doi.org/10.1109%2Fcvpr42600.2020.00424.

[10] Tobias Böttger and Markus Ulrich. "Real-time Texture Error Detection on Textured Surfaces with Compressed Sensing". In: vol. 26. Dec. 2014. DOI: 10.1134/S1054661816010053.

[11] Diego Carrera et al. "Defect Detection in SEM Images of Nanofibrous Materials". In: *IEEE Transactions on Industrial Informatics* 13 (2017), pp. 551–561. URL: https://api.semanticscholar.org/CorpusID:13356813.

[12] Diego Carrera et al. "Detecting anomalous structures by convolutional sparse models". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280790.

[13] Diego Carrera et al. "Scale-invariant anomaly detection with multiscale group-sparse models". In: *2016 IEEE International Conference on Image Processing (ICIP)* (2016), pp. 3892–3896. URL: https://api.semanticscholar.org/CorpusID:217954261.

[14] Niv Cohen and Yedid Hoshen. *Sub-Image Anomaly Detection with Deep Pyramid Correspondences*. 2021. arXiv: 2005.02357 [cs.CV].

[15] Thomas Defard et al. *PaDiM: a Patch Distribution Modeling Framework for Anomaly Detection and Localization*. 2020. arXiv: 2011.08785 [cs.CV].

[16] Vahid Garousi et al. "Visual GUI testing in practice: An extended industrial case study". In: *CoRR* abs/2005.09303 (2020). arXiv: 2005.09303. URL: https://arxiv.org/abs/2005.09303.

[17] Izhak Golan and Ran El-Yaniv. *Deep Anomaly Detection Using Geometric Transformations*. 2018. arXiv: 1805.10917 [cs.LG].

[18] Dong Gong et al. "Memorizing normality to detect anomaly: Memory-augmented deep autoencoder for unsupervised anomaly detection". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1705–1714.

[19] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[20] R. Hadsell, S. Chopra, and Y. LeCun. "Dimensionality Reduction by Learning an Invariant Mapping". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100.

[21] Dan Hendrycks et al. *Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty*. 2019. arXiv: 1906.12340 [cs.LG].

[22] Geoffrey E. Hinton. "Connectionist learning procedures". In: *Artificial Intelligence* 40.1 (1989), pp. 185–234. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(89)90049-0. URL: https://www.sciencedirect.com/science/article/pii/0004370289900490.

[23] Nathalie Japkowicz, Catherine Myers, and Mark Gluck. "A Novelty Detection Approach to Classification". In: *Proceedings of the Fourteenth Joint Conference on Artificial Intelligence* (Oct. 1999).

[24] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML].

[25] Wenqian Liu et al. *Towards Visually Explaining Variational Autoencoders*. 2020. arXiv: 1911.07389 [cs.CV].

[26] Shuang Mei, Hua Yang, and Zhouping Yin. "An Unsupervised-Learning-Based Approach for Automated Defect Inspection on Textured Surfaces". In: *IEEE Transactions on Instrumentation and Measurement* 67.6 (2018), pp. 1266–1277. DOI: 10.1109/TIM.2018.2795178.

[27] Kevin Moran et al. *Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps*. 2018. arXiv: 1802.02312 [cs.SE].

[28] Eric Nalisnick et al. *Do Deep Generative Models Know What They Don't Know?* 2019. arXiv: 1810.09136 [stat.ML].

[29] Paolo Napoletano, Flavio Piccoli, and Raimondo Schettini. "Anomaly Detection in Nanofibrous Materials by CNN-Based Self-Similarity". In: *Sensors* 18.1 (2018). ISSN: 1424-8220. DOI: 10.3390/s18010209. URL: https://www.mdpi.com/1424-8220/18/1/209.

[30] Matilda Noblia. *Automatic Anomaly Detection in Graphical User Interfaces Using Deep Neural Networks*. July 2019. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1373586&dswid=4487.

[31] Pramuditha Perera, Ramesh Nallapati, and Bing Xiang. "Ocgan: One-class novelty detection using gans with constrained latent representations". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 2898–2906.

[32] Pramuditha Perera and Vishal M. Patel. "Learning Deep Features for One-Class Classification". In: *CoRR* abs/1801.05365 (2018). arXiv: 1801.05365. URL: http://arxiv.org/abs/1801.05365.

[33] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[34] Mohammad Sabokrou et al. *AVID: Adversarial Visual Irregularity Detection*. 2018. arXiv: 1805.09521 [cs.CV].

[35] Mayu Sakurada and Takehisa Yairi. "Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction". In: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. MLSDA'14. Gold Coast, Australia QLD, Australia: Association for Computing Machinery, 2014, pp. 4–11. ISBN: 9781450331593. DOI: 10.1145/2689746.2689747. URL: https://doi.org/10.1145/2689746.2689747.

[36] Mohammadreza Salehi et al. "Puzzle-ae: Novelty detection in images through solving puzzles". In: *arXiv preprint arXiv:2008.12959* (2020).

[37] Thomas Schlegl et al. *Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery*. 2017. arXiv: 1703.05921 [cs.CV].

[38] Vikash Sehwag, Mung Chiang, and Prateek Mittal. *SSD: A Unified Framework for Self-Supervised Outlier Detection*. 2021. arXiv: 2103.12051 [cs.CV].

[39] Jihoon Tack et al. *CSI: Novelty Detection via Contrastive Learning on Distributionally Shifted Instances*. 2020. arXiv: 2007.08176 [cs.LG].

[40] Yao Tang et al. "Integrating prediction and reconstruction for anomaly detection". In: *Pattern Recognition Letters* 129 (Jan. 2020), pp. 123–130. DOI: 10.1016/j.patrec.2019.11.024.

[41] Shashanka Venkataramanan et al. *Attention Guided Anomaly Localization in Images*. 2020. arXiv: 1911.08616 [cs.CV].

[42] Xianghua Xie and Majid Mirmehdi. "TEXEMS: Texture Exemplars for Defect Detection on Random Textured Surfaces". In: *IEEE transactions on pattern analysis and machine intelligence* 29 (Sept. 2007), pp. 1454–64. DOI: 10.1109/TPAMI.2007.1038.

[43] Xudong Yan et al. "Learning Semantic Context from Normal Samples for Unsupervised Anomaly Detection". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.4 (May 2021), pp. 3110–3118. DOI: 10.1609/aaai.v35i4.16420. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16420.

[44] Jie Yang. "Unsupervised anomaly segmentation via deep feature reconstruction". In: *Neurocomputing* 424 (Dec. 2020). DOI: 10.1016/j.neucom.2020.11.018.

[45] Jie Yang et al. *Visual Anomaly Detection for Images: A Survey*. 2021. arXiv: 2109.13157 [cs.CV].

[46] Fei Ye et al. "Attribute restoration framework for anomaly detection". In: *IEEE Transactions on Multimedia* 24 (2020), pp. 116–127.

[47] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: Using GUI Screenshots for Search and Automation". In: *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*. UIST '09. Victoria, BC, Canada: Association for Computing Machinery, 2009, pp. 183–192. ISBN: 9781605587455. DOI: 10.1145/1622176.1622213. URL: https://doi.org/10.1145/1622176.1622213.

[48] Jihun Yi and Sungroh Yoon. *Patch SVDD: Patch-level SVDD for Anomaly Detection and Segmentation*. 2020. arXiv: 2006.16067 [cs.CV].

[49] Muhammad Zaigham Zaheer et al. "Old is gold: Redefining the adversarially learned one-class classifier training paradigm". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 14183–14193.

[50] Vitjan Zavrtanik, Matej Kristan, and Danijel Skočaj. "Reconstruction by inpainting for visual anomaly detection". In: *Pattern Recognition* 112, 107706 (Apr. 2021), p. 107706. DOI: 10.1016/j.patcog.2020.107706.

[51] Houssam Zenati et al. "Efficient gan-based anomaly detection". In: *arXiv preprint arXiv:1802.06222* (2018).

[52] Bo Zong et al. "Deep Autoencoding Gaussian Mixture Model for Unsupervised Anomaly Detection". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=BJJLHbb0-.