Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports

Yanfu Yan
yyan09@wm.edu
William & Mary
Williamsburg, Virginia, USA

Nathan Cooper
nacooper01@.wm.edu
William & Mary
Williamsburg, Virginia, USA

Oscar Chaparro
oscarch@wm.edu
William & Mary
Williamsburg, Virginia, USA

Kevin Moran
kpmoran@ucf.edu
University of Central Florida
Orlando, Florida, USA

Denys Poshyvanyk
denys@cs.wm.edu
William & Mary
Williamsburg, Virginia, USA

ABSTRACT

Video-based bug reports are increasingly being used to document bugs for programs centered around a graphical user interface (GUI). However, developing automated techniques to manage video-based reports is challenging as it requires identifying and understanding often nuanced visual patterns that capture key information about a reported bug. In this paper, we aim to overcome these challenges by advancing the bug report management task of *duplicate detection* for video-based reports. To this end, we introduce a new approach, called Janus, that adapts the scene-learning capabilities of vision transformers to capture subtle visual and textual patterns that manifest on app UI screens — which is key to differentiating between similar screens for accurate duplicate report detection. Janus also makes use of a video alignment technique capable of adaptive weighting of video frames to account for typical bug manifestation patterns. In a comprehensive evaluation on a benchmark containing 7,290 duplicate detection tasks derived from 270 video-based bug reports from 90 Android app bugs, the best configuration of our approach achieves an overall mRR/mAP of 89.8%/84.7%, and for the large majority of duplicate detection tasks, outperforms prior work by ≈9% to a statistically significant degree. Finally, we qualitatively illustrate how the scene-learning capabilities provided by Janus benefits its performance.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Bug Reporting, GUI Learning, Duplicate Video Retrieval

ACM Reference Format:
Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2024. Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639163

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639163

1 INTRODUCTION

Video-based bug reports are becoming increasingly popular for mobile applications [23, 34, 35, 52]. As mobile app bugs typically manifest visually via the graphical user interface (GUI), recording videos depicting bugs is more natural compared to textual bug reports [23, 35]. App users can easily record app bugs via the recording features of mobile operating systems (*e.g.,* Android [5]) or via third-party recording apps [4]. Additionally, popular issue trackers, such as GitHub [8], offer easy-to-use features for users to submit these videos to app developers. The rapidly increasing use of videos in mobile app issue trackers has been documented by recent studies [34, 52]. Feng *et al.* studied open source apps hosted on FDroid [6], and reported the usage of over 13k video recordings in issue trackers between 2012-2020, with a significant increase in usage during 2018-2020 (*i.e.,* a 15% - 35% increase). Kuramoto *et al.* [52] reported a 13% increase in issues containing videos in 2017-2021 for 289k popular GitHub projects.

While video-based bug reporting offers various advantages (ease of recording and submission, and visual details about app bugs [23, 26, 34, 35, 52]), it also presents several challenges for developers during bug report management tasks, particularly in scenarios where a high volume of bug reports is encountered [23, 34, 35, 52].

One of the most challenging tasks for developers is determining whether video-based bug reports depict the same app bug. This situation arises when multiple users independently report identical problems with the application (*e.g.,* during crowd-sourced app testing [26, 31, 40]). In such scenarios, developers face the challenge of watching, understanding, and assessing incoming and previously submitted video-based bug reports. This task can be extremely challenging since these recordings typically show numerous steps executed rapidly, making it difficult to recognize the bug reproduction scenario from the videos [23, 26, 35]. Additionally, the depicted buggy app behavior may not be apparent in the videos for the various types of bugs that apps can show in their GUI [33]. Developers often need to pause and replay the videos multiple times in order to fully understand the reported problems [23, 35]. The task of duplicate (video-based) bug report detection is crucial during the

bug triage process, as it helps developers avoid excessive redundant effort in investigating and resolving identical issues [26, 31, 40, 71].

This challenge is particularly prominent in crowd-sourced testing of mobile apps [31, 40], wherein software vendors engage a large distributed user base to test applications across diverse operational environments, for example, encompassing various devices, locations, and mobile networks. Crowd-sourced app testing often leads to multiple users encountering and reporting the same app-related issues. In fact, previous research has found that a substantial proportion (80%+) of bug reports submitted by users during crowd-sourced app testing are duplicates [71]. Consequently, developers often spend considerable effort on duplicate detection, which can impede the overall bug resolution process [26, 31, 40, 71].

In this paper, we propose Janus, a novel automated approach designed to assist developers in identifying duplicate video-based bug reports. Janus combines visual representation learning, information retrieval, and sequence-based algorithms, to analyze the visual, textual, and sequential information present in video-based bug reports. Through these methods, Janus establishes the degree of similarity between videos in reporting the same bug, thus enabling the automated detection of duplicate reports.

To model the visual information within videos, Janus leverages the Vision Transformer (ViT) architecture [30] and the self-supervised training scheme DINO [18], which extract rich hierarchical features that explicitly capture scene layout information related to GUI screens. In addition, Janus analyzes the textual content of videos by leveraging the Efficient and Accurate Scene Text Detector (EAST) [81] and a Transformer-based Optical Character Recognition (TrOCR) model [53], which accurately localize and extract text from video frames. By encoding this textual content via an adapted vector space model (VSM) [37], Janus assesses the textual similarity between two videos. Finally, to encode the sequential aspect of videos, Janus incorporates an adapted version of the classical longest common substring algorithm, giving higher weight to subsequent video frames that show the buggy app behaviors even if the videos show distinct bug reproduction scenarios.

We evaluate Janus using a comprehensive benchmark of 7,290 duplicate detection tasks, constructed from 270 video-based bug reports representing 90 unique bugs found in nine Android apps. We created this benchmark by extending an existing dataset that relied mostly on synthetic bugs [26]. Specifically, we extended it by incorporating 90 video-based bug reports pertaining to 30 real bugs of different kinds (*e.g.,* crashes, incorrect app output, and cosmetic issues) from three additional apps, resulting in a more comprehensive, realistic, and diverse benchmark.

Through multiple ablation experiments, we systematically assess the performance of the individual components of Janus as well as various combinations of these components. Our evaluation demonstrates that the most optimal configuration of Janus (when visual, textual, and sequential video information is combined) achieves an overall mRR/mAP of 89.8%/84.7%, surpassing the performance of an existing duplicate detector by ≈9% (with statistical significance). These results suggest that Janus can significantly reduce the effort required to identify duplicate video-based bug reports, as developers would only need to review fewer video reports to assess whether an incoming report depicts a known bug.

Furthermore, we conducted a qualitative analysis to understand the reasons behind Janus' performance compared to prior work. Notably, Janus exhibits an interpretable representation of video frames, effectively capturing nuanced patterns related to GUI component style, composition, and layout, which are crucial in accurately distinguishing duplicate video-based bug reports.

In summary, this paper makes the following main contributions:

- A new approach (Janus) that leverages visual representation learning for the graphical/lexical analysis of video-based bug reports. It also leverages sequential information within these reports for more accurate duplicate detection;
- A comprehensive empirical evaluation that demonstrates state-of-the-art improvements achieved by Janus in detecting duplicate video-based bug reports, when compared to a prior duplicate detector;
- A qualitative analysis into potential reasons for our improved results as well as future research directions on how the results can be further improved; and
- A publicly available benchmark for evaluating duplicate video-based bug report detectors [75], composed of 7,290 duplicate detection tasks created from 120/150 reports about 40/50 real/synthetic bugs—the largest benchmark to date.

2 THE JANUS DUPLICATE DETECTOR

This section describes the architecture and design details behind Janus, our approach for duplicate video-based bug report detection.

2.1 Problem Formulation and Challenges

We formulate the problem of duplicate detection as an information retrieval (IR) problem, as is typically done for textual bug reports [47, 54, 78]. A newly-submitted video-based bug report (the query) is compared against the set of previously submitted video reports in the issue tracker (the corpus) via a retrieval engine (*e.g.,* Janus), which retrieves and ranks the corpus reports by their similarity with the query. The higher a video-based report is ranked, the more likely it is to depict the same bug as the query. A developer then would watch the ranked videos in a top-down manner, marking the new video as duplicate if they find a video depicting the same bug. Video-based bug reports depict the incorrect behavior of an application (*e.g.,* GUI screens showing a crash, layout problems, or functional misbehavior), and the actions performed by the user on the GUI screens that lead to such misbehavior (*i.e.,* the steps to reproduce the bug). Duplicate video-based bug reports are pairs of two reports, *e.g.,* the query report and one corpus report, that depict the same buggy behavior, possibly showing different GUI steps, as multiple sequences of steps can lead to the manifestation of the same buggy behavior. An advantage of an IR formulation over other methods (*e.g.,* binary classification as (non-)duplicates reports [19]) is the fact that a ranked list gives higher flexibility to developers, because multiple bug reports are recommended as possibly showing the same bug.

While the primary goal of a duplicate detector is to identify whether two distinct videos depict the same incorrect app behavior, there are multiple challenges that make this task particularly difficult. For instance, duplicate videos may vary in length and display different reproduction steps, stemming from diverse reproduction

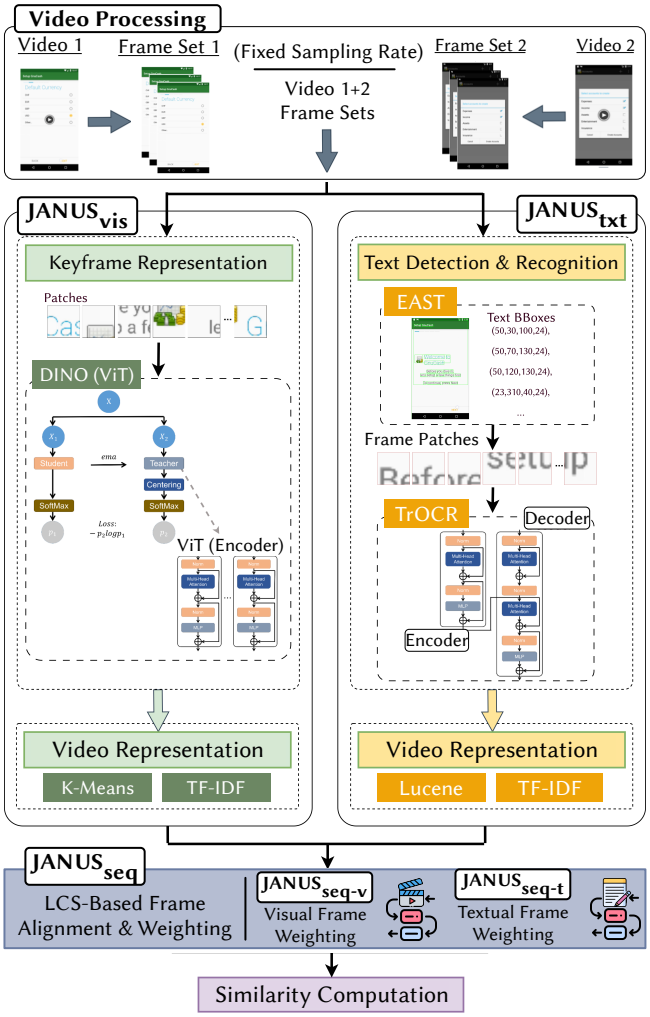

Figure 1: Overview of the JANUS duplicate detector.

scenarios executed by the users or the omission of certain steps during recording. Even if the reproduction steps appear the same or highly similar across videos, users may execute them at varying speeds. Distinguishing between different videos displaying distinct yet similar unexpected app behavior and reproduction steps can pose challenges for detectors. Furthermore, certain applications may exhibit dynamic content. For example, a mobile web browser allows users to navigate websites with varying layouts/content.

2.2 JANUS Overview

An overview of JANUS is shown in Fig. 1. JANUS receives as input two video-based bug reports and outputs a similarity score that indicates how similar they are at depicting the same app bug. JANUS can be used to compute scores between a new video-based bug report and a corpus of videos representing previously-submitted bug reports. The scores allow for ranking the corpus videos as a list of potential duplicate candidates. The goal of JANUS is to rank higher in this list, the actual duplicates for the new video.

Internally, JANUS begins sampling a number of frames from the two videos at a given rate (every sixth frame following the findings of past work [26]) to reduce overhead, given that successive frames tend to be exact or near duplicates of each other. Next, JANUS computes a vector representation of the videos, by processing the visual and/or textual content of the frames. JANUS's visual component, JANUS$_{vis}$, vectorizes each video into a visual TF-IDF representation by discretizing the frames into a Bag of Visual Words (BoVW) [45], using a feature extractor based on a Vision Transformer (ViT) model [30] and the DINO self-supervised training scheme [18]. JANUS's textual component, JANUS$_{txt}$, vectorizes each video into a textual TF-IDF representation by extracting video frame text (via the EAST [81] and TrOCR [53] models) and constructing a document of the concatenated text, represented as a Bag of Words (BoW) [65]. Each pair of visual or textual TF-IDF representations is then compared via cosine similarity. The visual and textual similarities can be used individually to rank duplicate candidates, or they can be combined into a single similarity score to account for both modalities of information, ideally leading to more effective duplicate detection.

To account for the sequential nature of video-based bug reports, which typically show the reproduction steps first and the incorrect app behavior afterward, JANUS can compute an alternative similarity score, based on a customized version of the longest common substring (LCS) algorithm, which matches the vector representation of video frames via cosine similarity and produces an overall similarity score that weights more heavily the later frames in the video than the earlier ones. This similarity is computed by JANUS's sequential component, JANUS$_{seq}$, which operates on the visual (JANUS$_{seq-v}$) and textual (JANUS$_{seq-t}$) vector representations of the frames.

2.3 JANUS$_{vis}$: Visual Representation of Videos

JANUS$_{vis}$ obtains a visual representation of a video in two steps. First, the sampled video frames are resized to 224×224 (*pixels*) and encoded via visual representation learning [49]. Second, these frame embeddings are further processed into a Bag of Visual Words (BoVW) [50], which is used to represent a video as a TF-IDF vector [65]. The goal is to learn useful visual information from app GUI components and layouts shown in the videos to distinguish potential duplicates from non-duplicates.

2.3.1 ***Visual Representation of Video Frames.*** Visual representation learning aims to obtain high-quality visual representations that are helpful for downstream tasks such as image classification [51], object detection [80], or image captioning [43]. This task is typically carried out in an unsupervised, self-supervised, or supervised manner [25, 60]. Most recently, there has been a focus on contrastive [25, 60] and distillation learning methods [18]. A promising technique, known as the Vision Transformer (ViT) [30] has recently been proposed to better learn visual representations. The performance of this architecture has been demonstrated to surpass or, at the very least, match previous models relying on Convolutional Neural Networks (CNNs) for image classification. However, the most significant advantage of ViT lies in its ability to excel beyond CNNs in capturing explicit information concerning the semantic segmentation of an image (*i.e.,* layouts and object boundaries) [30].

We posit that learning object segmentation within an image is particularly useful for app GUI screens, given their structured, component-based nature. Hence, we adopted the ViT architecture

for designing Janus$_{vis}$. The ViT architecture is comprised of a standard Transformer encoder model [29] but instead of lexical tokens, "patches" from images are fed into the network. These patches are treated the same way that tokens are in lexical transformers: they are linearly transformed and have added positional embeddings.

Given that image-level supervision requires labor-intensive annotations and limits the information that can be learned during training to a single concept with a few categories of objects (as is the case of app GUI screens, which contain components and layouts of well-defined kinds), we need to train our ViT model in a self-supervised manner. Janus$_{vis}$ trains its ViT using the self-supervised training methodology DINO [18], which leverages a student-teacher knowledge distillation training scheme [42]. In this scheme, the student network is trained to match the distribution of the teacher network by minimizing the standard cross-entropy loss. Usually, the teacher network is larger than the student network in terms of the number of model parameters. However, the teacher network in DINO is built from the past iterations of the student network with an exponential moving average strategy, whose parameters are frozen over an epoch by applying a stop-gradient operator, given that direct replication of the student weights fails to converge. The outputs of both networks are normalized using a temperature softmax. To adapt the knowledge distillation architecture to self-supervised learning, two global views and several local views are constructed on the basis of data augmentations [38] and the multi-crop strategy [17], with local views passed through the student while only the global views are passed through the teacher network, to encourage local-to-global correspondence. By combining DINO with ViT, we aim to further improve the ability to capture global GUI layouts.

Through this self-supervised training process, the model learns a rich representation of images that emphasize scene layouts and object boundaries. To further refine the DINO model's capabilities to our domain of app GUI screens, we fine-tuned Janus's ViT model, which was pre-trained on ImageNet [28], on a collection of 66k mobile app screenshots from the popular RICO dataset [27]. We directly use the projected output of the *[CLS]* token, a special token that marks the aggregation of all image patch embeddings, from the last block of the ViT model as the representation of video frames.

2.3.2 ***Visual Representation of Videos.*** To represent a video, Janus$_{vis}$ implements a BoVW + TF-IDF approach since it has been shown to be more useful for video retrieval compared to other approaches [50] (*e.g.,* using directly the frame representations for similarity computation or aggregating them into a single vector).

Janus discretizes the frame representations by leveraging a Codebook of visual words [50]. The Codebook represents a catalog of visual words, which are representative vectors found in a corpus of images (in our case, images of app GUIs). The Codebook is constructed via a trained K-Means model that clusters the corpus of image representations into K clusters, the centroids being the visual words. Janus then assigns each video frame representation to its closest cluster centroid (*i.e.,* a visual word) via Euclidean distance. The Codebook is trained by randomly sampling 15k mobile app screenshots from the RICO dataset [27], vectorizing them via our fine-tuned ViT model, and running the K-Means algorithm on the vectors, with $K = 1k$ recommended by prior work [50]. We take a sample rather than using the full RICO dataset due to computational constraints of the K-Means algorithm. The Codebook is trained only once before the TF-IDF representation approach is applied.

Once each frame representation is discretized to its corresponding visual word, Janus computes a TF-IDF vector representation of a video, as similarly done for text retrieval [65]. The term frequency (TF) is the count of each visual word in the video. The inverse document frequency (IDF) is the count of BoVW representations of existing videos where a visual word appears. Since a corpus of existing videos for a particular app may be small and may lack diversity, we consider the set of RICO images as the corpus of existing videos. By considering the diversity of apps in the RICO dataset, we aim to improve the generalization of the TF-IDF video representations.

Janus$_{vis}$ compares the TF-IDF representation of two videos via cosine similarity to establish the likelihood of the videos showing the same app bug. This method is applied to the existing corpus of TF-IDF visual representations for an app to generate a ranked list of candidate duplicate videos for a new video-based bug report.

To address potential biases due to random sampling when creating the Codebook, we adapted Janus$_{vis}$ to use four Codebooks (each trained on 15k RICO images, 60k in total). Specifically, Janus$_{vis}$ uses each Codebook to produce similarity scores for a set of videos. These similarity scores are averaged to produce a final set of similarities and video ranking. More details are given in section 3.3.2.

2.4 Janus$_{txt}$: Textual Representation of Videos

Janus$_{txt}$ creates a textual representation of a video in two steps: (1) it localizes and extracts the text present in video frames via neural text localization and Optical Character Recognition (OCR); and (2), it encodes the extracted text using a standard TF-IDF representation [65]. The goal is to leverage the text from labels, messages, and other sources shown in the frames to compute video similarity.

For the first step, Janus$_{txt}$ has two components: (1) a text localization component that proposes image regions where text is rendered, and (2) a text recognition component that takes those regions and extracts any text present in them. The text localization component implements the Efficient and Accurate Scene Text Detector (EAST) model [81], which has been trained to directly derive region proposals. The text recognition component leverages the TrOCR Transformer model [53], which takes region proposals from EAST and directly predicts the text represented in the proposals. The combination of EAST and TrOCR was adopted over the popular TesseractOCR [1] approach because: (1) such a combination simplifies the overall OCR pipeline since it relies on neural models only, without needing heuristic-based approaches to filter out poor text region candidates (as TesseractOCR does); and (2) such a combination has shown strong performance improvements in detecting scene text as well as handwritten/printed text, which means it is less sensitive to noise in the images. Each video frame is put through this 2-stage pipeline to extract its text.

For the second step, Janus$_{txt}$ concatenates the text from all video frames and pre-processes it via tokenization, lemmatization, and removal of special characters, such as non-ASCII characters, punctuation, or stop words. This resulting text is used to build a Bag of Words (BoW) representation of the video, which is then encoded as a standard textual TF-IDF representation using the popular Lucene

library [37], which implements the standard information retrieval Boolean model and the Vector Space Model (VSM) [65]. We use this textual representation approach over neural text encoding models because it is based on exact text matching, which could lead to more accurate similarity computation of duplicate videos (as they are likely to show the same text on the buggy app screens).

Finally, Janus$_{txt}$ compares the TF-IDF representation of two videos using Lucene's similarity scoring function (based on cosine similarity and document length normalization) [11]. Similarity computation can be applied to a corpus of video-based bug reports to generate a ranked list of potential duplicate videos to the new video.

2.5 Janus$_{seq}$: Sequential Similarity of Videos

Janus$_{vis}$ and Janus$_{txt}$ ignore the sequential order of the videos, as these components are based on Bags Of (Visual) Words. However, the buggy app behavior is typically shown toward the end of a video-based bug report, after the bug reproduction steps have been rendered. To account for the sequential order of the videos, Janus employs a modified version of the longest common substring (LCS) algorithm to compute an alternative similarity score between videos. This approach is coined as Janus$_{seq}$ and operates on both visual (Janus$_{seq-v}$) and textual representations of the videos (Janus$_{seq-t}$).

Janus$_{seq}$ treats a video as a sequence of visual/textual words, based on the vector representation of the video frames, and applies an LCS-based approach for similarity computation. Intuitively, the longer the LCS between videos is, the higher their similarity is. The textual representation of a video frame is the TF-IDF vector of the text extracted from the frame, using the approach described in section 2.4. In the standard word-based LCS algorithm, words are compared using exact text matching. To account for similar, yet different video words (which might be common for textual video representations), we relaxed this matching scheme and instead used cosine similarity between video frame representations. Additionally, the similarity-based matching should weigh more heavily the frames that appear later in the videos as they are more likely to show the buggy app behavior and should give a normalized similarity score between zero and one.

Given these requirements, we defined the following similarity computation for Janus$_{seq}$: $S_{seq} = \frac{\text{w-}LCS}{\max \text{w-}LCS}$, where the numerator, w-*LCS*, represents the amount of overlap between two videos, given by our modified LCS algorithm, which uses the cosine similarity between frames (rather than exact matching) and a weighting scheme that favors later frames in the videos. The weighting scheme is $\frac{i}{m} \times \frac{j}{n}$, where i is the ith frame of a first video, with m being its # of frames, and j is the jth frame of a second video, with n being its # of frames. The denominator, max w-*LCS*, represents the maximum possible overlap if the videos were identical. Since the videos could be of different lengths, we align the end of the shorter video (with length *min*), to the end of the longer video (with length *max*), and calculate the maximum overlap as: $\sum_{i=1}^{min} \frac{i}{min} \times \frac{max-i}{max}$.

2.6 Combining Janus's Components

To design Janus, we explore different combinations of its components. The similarity scores from Janus$_{vis}$ and Janus$_{txt}$ can be linearly combined as $(1-w) \times S_{vis} + w \times S_{txt}$, with $w \in [0, 1]$— the higher w is the more weight it gives to textual information from the videos. We also explore various combinations that replace this similarity calculation with those given by Janus$_{seq}$ (Janus$_{seq-v}$ & Janus$_{seq-t}$), which consider the sequential video information.

3 EVALUATION METHODOLOGY

We investigate the performance of Janus's components (Janus$_{txt}$, Janus$_{vis}$, and Janus$_{seq}$), as well as the performance of various combinations of these components, and compare these to a baseline duplicate detection technique proposed in prior work [26]. Additionally, we aim to understand why we observe various trends in the overall performance of Janus, and qualitatively examine cases where Janus is able to outperform the baseline technique. To that end, we formulate the following research questions (RQs):

RQ$_1$: *What is* Janus$_{vis}$'s *duplicate detection performance?*
RQ$_2$: *What is* Janus$_{txt}$'s *duplicate detection performance?*
RQ$_3$: *What is* Janus$_{seq}$'s *duplicate detection performance?*
RQ$_4$: *What is the performance of Janus's component combinations?*

3.1 Duplicate Detection Dataset

We constructed a comprehensive evaluation dataset by extending a prior dataset that mostly relied on synthetic app bugs [26]. The previous dataset collected 60 distinct bugs (35 crashes and 25 non-crashes) across six Android apps of different sizes and domains (*e.g.,* podcast, finance, and weight management apps). The dataset contains ten confirmed real bugs and 50 bugs injected by the mutation testing tool MutAPK [33], which generates code mutations based on diverse mutant operators that affect various app features. The dataset includes three duplicate videos per bug, for a total of 180 video-based bug reports, and a set of 810 duplicate detection tasks per app, for a total of 4,860 tasks, created from the videos. We refer to this dataset as the *original dataset*. We next describe how we extended this dataset and detail the creation of video-based bug reports and duplicate detection tasks to evaluate Janus.

3.1.1 ***Extended Real Bug Dataset.*** We extended the prior dataset by constructing an evaluation dataset containing *only* real bugs. Wendland *et al.* [74] released the AndroR2 dataset containing 90 manually reproduced bug reports for Android apps. This dataset was later extended through the addition of another 90 reproduced bug reports in the AndroR2+ dataset [46], for a total of 180 real, reproducible reports. For each bug report, AndroR2+ provides a link to the original bug report in the issue tracker, an `apk` of the buggy app version, a reproduction script, and metadata for bug reproduction (device, OS version, *etc.*).

To construct our new dataset of real bugs, we chose the three apps with the largest number of bugs from AndroR2+, while also ensuring the diversity of app categories. We selected: Firefox Focus (FCS) [7], a web browser; PDF Converter (ITP) [10], an image-to-PDF converter; and GPSTest (GPS) [9], a GPS testing app. FCS is the only app that renders dynamic content on the screen. For these apps in Andror2+, we found ten bug reports for FCS, nine reports for GPS, and eight reports for ITP. We further manually checked each app's issue tracker and collected one more bug for GPS and two more bugs for ITP to have the same number of bugs per app. To find the `apk` files of the correct buggy version of the apps for these three bugs, we chose the app version closest to the date the issue

was created and confirmed that the `apk` allowed for the successful reproduction of the bug. Based on the AndroR2+ metadata and the three bug reports we collected, there are seven different OS versions used to reproduce the bugs, namely, Android version 4.4.4, 6.0.1, 7, 7.1, 8, 8.1, & 9.

3.1.2 ***Duplicate Video Recording.*** The paper authors and external participants recorded videos replicating the collected 30 real bugs from the three AndroR2+ apps, following prior work [26].

We rewrote the descriptions of steps to reproduce (S2R), expected behaviors, and observed behaviors for these bugs to ensure they are clear and easy for participants to reproduce from an end-user perspective. Although the AndroR2+ bugs were reproducible on a Pixel 2 emulator, we chose Nexus 5X to maintain the same device configuration as the previous dataset [26], since the bugs were also reproducible on the Nexus 5X. This ensures a consistent resolution of the videos across the benchmarks. Additionally, we minimized the different OS versions to three (6, 8.1, and 9) to reduce participants' effort by finding the closest OS versions to their original ones while ensuring the bugs were still reproducible. Also, having these additional OSes in our video reproductions of these bugs has the added benefit of being more realistic—the prior dataset only used Android 7.0. While AndroR2+ provides automated bug reproduction scripts, we avoided using them for two reasons: (i) we found that certain scripts led to errors that did not properly reproduce the bug, and (ii) we wanted to capture video-based reports depicting real human actions, to ensure the most realistic setting possible.

Video-based reports were created by the paper authors for all 30 bugs according to the S2R. To maintain three duplicate videos per bug, in line with the previous dataset, two authors (who previously did not record any videos) along with two Ph.D. students were asked to record the additional 60 videos, each responsible for reproducing 15 distinct bugs with only the descriptions of expected and observed behaviors, to ensure diversity of reproduction steps. Unlike the prior dataset, the recorded videos do not show the Android touch indicator when the user taps on the screen.

In total, our new dataset consists of 90 video-based bug reports corresponding with three duplicates of 30 real bugs from three apps. It contains two crashes and 28 non-crashes, comprising 270 reproduction steps in total (249 taps, six gestures, and 15 input entry actions) and ≈35-second videos, on average. There are six videos for Android 6, nine for Android 9, and 75 for Android 8.1.

3.1.3 ***Duplicate Detection Tasks.*** In line with the prior dataset, we construct *duplicate detection tasks* for each app to be as realistic as possible. We define a duplicate detection task as having: (1) a *query* video that represents a newly reported video-based bug report, and (2) a *corpus* of 13 existing video-based reports. The query must be compared against the corpus in order to determine whether the incoming report is a duplicate of an existing report. Each task contains videos of the 10 bugs for an app. The corpus contains two duplicate videos of the query (*i.e.,* they show the same bug). The remaining eleven videos are non-duplicates: three of them are duplicates of each other but not of the query (*i.e.,* they show a bug different from the query bug), and eight videos show distinct bugs. Each task simulates a situation that is similar to crowd-sourced app testing, where duplicates of the query, of other bugs, and unique video-based reports exist together on the issue tracker for an app.

Using different combinations of bugs and videos, we created a total of 810 tasks per app or 2,430 tasks across all apps. Combining both the prior and new datasets, there are 7,920 tasks in our extended evaluation benchmark to evaluate Janus.

3.2 Baseline Duplicate Detector

We compare Janus against the Tango duplicate detector introduced by Cooper *et al.* [26]. Tango also leverages multi-modal information to detect duplicate video-based reports, using less-sophisticated methods as compared to Janus. It extracts visual features from video frames using a contrastive learning method called SimCLR, which uses a ResNet-50 CNN to learn local features of app GUIs [25]. It also analyzes text displayed on GUI screens using an approach that combines LSTM-based language models and heuristics, relying on TesseracOCR to extract video frame text [1]. Finally, Tango performs limited alignment of video frames: only for its extracted visual SimCLR features. Tango's evaluation found the best performing configuration is when the visual and textual components are combined, hence we compare Janus against this configuration while also performing ablation comparisons between their individual components.

3.3 Metrics and Experimental Settings

3.3.1 ***Evaluation Metrics.*** We use standard metrics used in prior work on duplicate bug report detection evaluations [26, 47, 54, 78]:

- **Mean Reciprocal Rank (mRR)**: it gives a measure of the average ranking of the first duplicate video found in the candidate list of videos given by a duplicate detector. It is calculated as: $mRR = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{rank_i}$, for N duplicate detection tasks ($rank_i$ is the rank of the first duplicate video for task i).
- **Mean Average Precision (mAP)**: it gives a measure of the average ranking of all the duplicate videos for a query video. It is computed as: $mAP = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{DV}\sum_{v=1}^{DV}P_i(rank_v)$, where DV is the set of duplicate videos for task i, $rank_v$ is the rank of the duplicate video v, and $P_i(k) = \frac{duplicates}{k}$ is the number of duplicates in the top-k candidates.

All metrics give a normalized score in [0, 1]—the higher the score, the higher the duplicate detection performance. We executed different configurations of Janus and the baseline on the 7,920 tasks and computed/compared the metrics between these approaches.

3.3.2 ***Model Configurations.*** We compared Janus$_{vis}$ against Tango's visual component by experimenting with two ViT models: ViT-Small (ViT-S) and ViT-Base (ViT-B), which have six and 12 self-attention heads respectively. ViT-S has a similar size to RestNet-50's size (used by Tango's SimCLR): ≈23M parameters. To evaluate the differences between the SimCLR (contrastive) and DINO (distillation) training schemes, we implemented Janus$_{vis}$ with DINO + RestNet-50. We also experimented with the following patch sizes for ViT: 16x16 (/16) and 8x8 (/8) pixels, as the patch size can affect Janus$_{vis}$'s performance [30]. In total, we executed four DINO models: DINO (ResNet), DINO (ViT-S/16), DINO (ViT-S/8), & DINO (ViT-B/16). ViT-B/8 was not included in the experimentation for Janus$_{vis}$ due to its substantial computation overhead.

To account for potential biases from random image selection when constructing the Janus$_{vis}$'s Codebook, we used four distinct

Table 1: The network configurations and fine-tuning hyperparameters for Janus$_{vis}$ compared with SimCLR used by Tango

model	dim	# params	batch size	w-temp	temp
SimCLR	2,048	23M	1,792	–	–
DINO (ResNet)	2,048	23M	96	0.03	0.03 (0)
DINO (ViT-S/16)	384	21M	96	0.03	0.03 (0)
DINO (ViT-S/8)	384	21M	18	0.04	0.05 (30)
DINO (ViT-B/16)	768	85M	64	0.05	0.07 (50)

Codebooks, each trained on 15k distinct RICO images (60K images in total). With each Codebook, Janus$_{vis}$ generates similarity scores for a set of videos. These similarities are averaged across the four Codebooks to produce final scores used for ranking. To perform a fair comparison with Tango's visual component, we implemented the same Codebook generation strategy on Tango, using its publicly released implementation [26]. The recomputed Tango results on the prior dataset are slightly higher than those reported in the original paper (76.2 vs 75.3 mRR and 69.8 vs 67.8 mAP).

We compared Janus$_{txt}$ against Tango's textual component by experimenting with different configurations for the EAST and TrOCR models. For EAST, we used three different resolution thresholds to filter out small text regions: 5×5 (EAST-5), 40×20 (EAST-40), and 80 × 40 (EAST-80). The 5 × 5 threshold is used by default in EAST. We did not test larger resolutions than 80 × 40 to ensure that each textual document created for the video has at least one valid detection. 40 × 20 was included as a middle ground to understand the impact of the threshold size on the video similarity calculation. For TrOCR, we used its large version with BEiT Large [14] as the encoder and RoBERTa Large [55] as the decoder. Two fine-tuned TrOCR-Large models are used, namely TrOCR-p (fine-tuned on the printed text dataset SROIE [44]) and TrOCR-s (finetuned on the synthetic scene text datasets such as ICDAR15 [48] and SVT [72]).

3.3.3 ***Model Training.*** All visual models were fine-tuned on the 66k mobile app screenshots from the RICO dataset [27] for 100 epochs using model checkpoints trained on ImageNet [28], except for DINO (ViT-B/16), to fairly compare it with the Tango$_{vis}$'s SimCLR model. After examining preliminary results showing the advantages of DINO with ViT, we decided to train DINO (ViT-B/16) for 400 epochs [18]. Fine-tuning was carried out on three NVIDIA T4 Tesla GPUs with 16GB of memory each. Because DINO does not use contrastive learning, we were able to use a much smaller batch size as compared to the SimCLR model used in Tango: 96 *vs* 1,792 for ViT-S/16 and ResNet-50. For the ViT-B/16 and ViT-S/8 models, we used a batch size of 64 and 16 due to memory constraints. Table 1 shows the network configurations and three fine-tuning hyperparameters, where *dim* is the representation dimension of the output, *# params* is the total number of model parameters. *"temp" and "w-temp"* represent the teacher temperature and the warm-up teacher temperature respectively, and the numbers in parentheses are the # epochs used for warm-up. Model training was not required for Janus$_{txt}$ as we directly use pre-trained EAST and TrCOR models for GUI text localization and recognition [53, 81].

4 EVALUATION RESULTS AND DISCUSSION

Table 2 shows Janus's duplicate detection performance compared to the baseline Tango, for their individual components: visual, textual, and sequential. Table 3 shows the performance of different combinations of Janus components, compared to the baseline.

Cells shaded green in these tables indicate a statistically significant (via Wilcoxon's paired test at the $p < 0.05$ level) higher effectiveness when comparing a given Janus configuration/component to a given Tango configuration/component. Yellow shaded cells indicate a higher performance, but without statistical significance. We present the results for each app of the *original* (mostly synthetic bugs) and *extended* (real bugs) *datasets* and the overall results accounting for all the apps in both sets, separately and combined.

While we computed the performance of four Janus$_{vis}$ DINO models (*i.e.,* DINO with ResNet, ViT-S/16, ViT-S/8, and ViT-B/16), we present (in tables 2 and 3) the best-performing model for Janus$_{vis}$: DINO with ViT-B. Likewise, we report here the results of the best performing model configuration for Janus$_{txt}$, namely EAST-80 (EAST that filters out region proposals smaller than 80*x*40) combined with TrOCR-s (TrOCR fine-tuned on real-world scenes, *e.g.,* street scenes, instead of text found in printed and handwritten documents). The results for all the DINO, EAST, and TrOCR configurations can be found in our replication package [75].

Tables 2 and 3 show a consistent trend: the performance achieved by any duplicate detector (*i.e.,* any configuration) is lower for the original dataset than for the extended dataset. After investigating the minimal set of ground-truth reproduction steps of the bugs used in the datasets, we found this trend is explained by the number of overlapping steps across distinct bugs in an app. We observed that distinct bugs for a given app in the original dataset have a larger step overlap than distinct bugs in the extended dataset. It is more challenging for a duplicate detector to distinguish between duplicate and non-duplicate videos if there is a larger step overlap across bugs (hence, across videos). Recall that in a duplicate detection task, the videos in the corpus are for distinct bugs; if there is a larger overlap among them, particularly between duplicates and non-duplicates, a detector would struggle to discern the differences.

4.1 RQ1: Janus$_{vis}$'s Performance

Table 2 shows the duplicate detection effectiveness of Janus$_{vis}$ (DINO with ViT-B) compared to visual Tango (SimCLR).

Before discussing the table results, we briefly discuss the results of comparing the training schemes (distillation via DINO *vs* contrastive via SimCLR, both using the same pre-trained ResNet weights). We found that SimCLR outperforms DINO for six of nine apps by a relatively small margin (by 3.5% mRR and 4.2% mAP, on average), but DINO outperforms SimCLR for the remaining three apps (APOD, GNU, DINO) by a larger margin (7.5% mRR and 6% mAP, on avg.). Overall, across all the apps, we found a similar performance between these two approaches (less than 1.1% mRR/mAP improvement), which indicates the training scheme does not have a large impact on duplicate detection performance.

Furthermore, both ViT-S/16 and ViT-S/8 used by Janus$_{vis}$'s DINO exhibit superior performance compared to ResNet-50 used by visual Tango's SimCLR. Specifically, although ViT-S/16 and ViT-S/8 have a similar model size to RestNet-50, they outperform ResNet-50 by 2.91% and 2.92% respectively, in terms of mRR on average, with statistical significance. This highlights the effectiveness of ViT over ResNet for duplicate video-based bug report detection.

Table 2: Performance of the individual components of Janus and the baseline Tango

App	Visual				Textual				Sequential (visual)				Seq. (textual)	
	Tango		Janus$_{vis}$		Tango		Janus$_{txt}$		Tango		Janus$_{seq-v}$		Janus$_{seq-t}$	
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
APOD	77.19	69.98	87.32	79.79	80.80	75.30	79.76	73.09	55.01	44.85	84.45	71.11	73.40	68.09
DROID	68.43	58.82	80.77	71.44	67.90	64.70	78.88	72.52	46.54	37.91	61.49	50.88	76.19	73.64
GNU	81.53	75.83	81.83	75.54	84.50	82.30	89.53	81.28	55.91	43.37	71.41	58.82	52.79	43.81
GROW	83.53	78.60	87.46	80.33	76.80	69.00	82.65	77.38	74.57	64.46	92.14	84.57	77.45	76.24
TIME	70.26	65.35	73.76	69.46	47.40	37.70	64.80	56.67	50.85	43.62	63.14	56.18	69.34	66.16
TOK	76.03	70.37	81.11	71.33	61.30	53.30	53.95	44.48	38.13	33.36	53.39	43.22	54.00	47.83
Original	**76.16**	**69.83**	**82.04**	**74.65**	**69.80**	**63.70**	**74.93**	**67.57**	**53.50**	**44.59**	**71.00**	**60.80**	**67.20**	**62.63**
FCS	91.09	85.82	86.88	82.69	85.12	79.12	86.17	84.88	65.23	55.42	90.20	85.91	90.53	88.21
GPS	95.99	92.15	98.09	95.70	92.11	84.82	97.51	96.10	68.34	60.63	93.72	86.64	57.83	53.33
ITP	81.93	73.92	93.29	84.08	89.73	86.34	96.77	89.83	69.50	54.37	90.56	78.20	54.74	46.87
Extended	**89.67**	**83.96**	**92.75**	**87.49**	**88.98**	**85.74**	**93.48**	**90.27**	**67.69**	**56.81**	**91.50**	**83.58**	**67.70**	**62.80**
Overall	**80.66**	**74.54**	**85.61**	**78.93**	**76.14**	**70.06**	**81.11**	**75.14**	**58.23**	**48.67**	**77.84**	**68.39**	**67.36**	**62.69**

Table 2 shows that Janus$_{vis}$ (DINO with ViT-B) significantly outperforms the baseline on both datasets. We observe an overall improvement of (85.61 - 80.66)/80.66 = 6.1% mRR and (78.93 - 74.54)/74.54 = 5.9% mAP, with statistical significance. This overall performance comes from an improvement in eight out of nine apps compared to the baseline (seven with statistical significance), with Tango only having a substantial improvement over Janus$_{vis}$ for the FCS app. These FCS results are due to the nature of the app and the underlying models. Specifically, FCS is a web browser and the video-based bug reports produced for this app show users navigating to different websites. The app produces dynamic content: the navigated websites have different layouts and visual characteristics. Janus$_{vis}$'s ViT is prone to focusing more on the structure of the GUIs, extracting global features about the layouts, while the baseline's ResNet tends to focus on local visual features of the GUIs, not necessarily on general screen layouts, which are more beneficial to detect duplicates. Compared to ResNet, ViT's emphasis on GUI layouts leads to a more substantial dissimilarity between duplicates when sequential visual information is not taken into account.

4.2 RQ2: Janus$_{txt}$'s Performance

Table 2 shows that Janus$_{txt}$ is substantially more effective than textual Tango for seven of nine apps (with statistical significance), especially for DROID (improvement of 16.2% mRR and 12.1% mAP) and TIME (improvement of 36.7% mRR and 50.3% mAP). Only for the apps APOD and TOK, Tango is higher, resulting in Janus$_{txt}$'s overall superiority on both the original and the real bug datasets (overall, by 6.5%/7.3% mRR/mAP). The reason why Janus$_{txt}$ does not perform better for APOD and TOK is that these apps usually contain short or small pieces of text (*e.g.,* due to small fonts) on many of their screens, and EAST fails to identify them because these pieces fit in smaller regions than 80×40 pixels. Indeed, when reducing the threshold to 40×20, Janus$_{txt}$ outperforms Tango for APOD and TOK (by 1.5%/1.1% and 8.8%/5.2% mRR/mAP respectively).

Janus$_{txt}$'s performance is slightly higher than Janus$_{vis}$'s for the extended dataset (improvement of 0.8%/3.1% mRR/mAP overall), but lower for the original dataset (by 9.5%/10.5% mRR/mAP). The lower improvements come from the TOK app, which does not contain enough textual information to accurately detect duplicates [26].

4.3 RQ3: Janus$_{seq}$'s Performance

Table 2 shows that Janus$_{seq-v}$ is substantially more effective in detecting duplicates than sequential Tango, when using visual frame representations. Janus$_{seq-v}$ outperforms the baseline for every app in the original dataset (by 32.7% mRR and 36.4% mAP overall) and in the extended dataset (by 35.2% mRR and 47.1% mAP overall). The high improvements can be attributed to the power of Janus's ViT in learning the global structure of GUI screens, while Tango's ResNet focuses on learning local GUI features. Global GUI structure representations are more useful to measure the sequential overlap between video frames even when there are small variations in the frames because of slightly different reproduction steps. Also, we note that the improvements for the FCS app are substantial (38%/55% mRR/mAP). Since we observed highly different GUI layouts in video frames for this app (because users navigated to different websites), these results are indicative of the effectiveness of the sequential similarity approach of Janus in combination with ViT-based frame representations (compared to the baseline).

Since Tango's sequential component is not designed to work with textual frame representations (unlike Janus), we only compare the performance of Janus$_{seq-t}$ with Janus$_{seq-v}$. Overall, Janus$_{seq-v}$ outperforms Janus$_{seq-t}$ by 15.6%/9.1% mRR/mAP. It substantially outperforms Janus$_{seq-t}$ for five of nine apps by 39.4%/35.8% mRR/mAP on average, while having a lower performance for the remaining four apps (7.4%/14.6% mRR/mAP). The largest improvement is observed in the GPS and ITP apps. ITP is an app used to convert images to PDF, involving mostly image editing, while GPS focuses on editing coordinates and displaying locations on a map. Consequently, video-based bug reports have limited text on each frame, which negatively impacts the performance of Janus$_{seq-t}$. However, since Janus$_{txt}$ leads to high performance for these two apps, we attribute Janus$_{seq-t}$'s relatively low performance to the alignment approach, which processes each video frame text rather than using the text from all frames together.

4.4 RQ4: Component Combination Performance

We linearly combined Janus's components (as described in section 2.6) to determine how much they improve the performance,

Table 3: Performance of different component combinations for Janus and the baseline Tango

App	Visual + Textual				Vis + Seq		Txt + Seq		Vis+Txt+Seq	
	Tango		Janus		Janus		Janus		Janus	
	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP	mRR	mAP
APOD	81.08	75.11	86.72	80.64	86.59	77.30	91.54	86.30	94.95	86.55
DROID	71.74	64.31	83.95	78.98	75.50	64.96	89.26	83.15	88.56	81.06
GNU	89.16	85.92	89.62	81.75	83.48	74.18	84.24	73.80	90.58	81.58
GROW	86.61	80.73	89.84	86.32	91.40	86.88	83.56	80.96	93.32	90.72
TIME	65.06	59.23	67.51	64.31	71.33	63.68	73.69	69.67	74.88	71.92
TOK	71.11	63.95	75.51	62.59	63.35	57.30	55.23	48.57	75.92	67.91
Orig.	**77.46**	**71.54**	**82.19**	**75.76**	**78.61**	**70.72**	**79.59**	**73.74**	**86.37**	**79.95**
FCS	91.11	86.87	88.46	85.95	93.98	89.85	90.38	84.74	94.73	91.90
GPS	97.35	95.53	99.30	98.31	98.09	96.01	89.20	86.91	98.24	97.26
ITP	90.64	86.51	96.84	91.58	96.05	89.99	83.94	77.07	97.41	93.51
Ext.	**93.03**	**89.64**	**94.87**	**91.94**	**96.04**	**91.95**	**87.84**	**82.91**	**96.79**	**94.22**
Overall	**82.65**	**77.57**	**86.42**	**81.16**	**84.42**	**77.79**	**82.34**	**76.80**	**89.84**	**84.71**

compared to the baseline and individual components. We experimented with different weights (from 0 to 1 in 0.1 increments) using all duplicate detection tasks and selected the weights that lead to the highest mRR/mAP performance.

As mentioned earlier, the best Tango configuration is when its visual and textual components are combined (with a weight of 0.8 and 0.2, respectively), as reported in the original paper [26]. Janus's visual and textual components (*i.e.,* Janus$_{vis}$ and Janus$_{txt}$) are combined using 0.9 and 0.1 as weights. This combination is denoted as "Visual + Textual" in Table 3. The table also shows the combination of Janus's visual/textual components and the sequential one: "Vis + Seq" denotes the average of the similarity scores produced by Janus$_{vis}$ and Janus$_{seq-v}$, while "Txt + Seq" denotes the average of the similarity scores produced by Janus$_{txt}$ and Janus$_{seq-t}$. An average combination means a weight of 0.5. Finally, we combine the similarities produced by the last two combinations using a weighted linear combination as follows: Sim(Vis + Seq) × 0.6 + Sim(Txt + Seq) × 0.4. This combination incorporates every information source from the videos and is denoted as "Vis + Txt + Seq".

Table 3 shows that the best performing Janus combinations are "Visual + Textual" and "Vis + Txt + Seq", both outperforming the baseline by 4.6%/4.6% mRR/mAP and 8.7%/9.2% mRR/mAP overall respectively (with statistical significance). The other two Janus combinations lead to mixed results: "Vis + Seq" leads to overall performance gains while "Txt + Seq" does not produce overall gains, due to its lower performance on the extended dataset.

When using "Visual + Textual", Janus significantly outperforms Tango on seven of nine apps and is only worse than Tango on FCS, considering both mRR and mAP. As previously mentioned, Janus's lower performance for FCS, compared to Tango, stems from the nature of the app itself. FCS is a web browser and the bugs used for this app were not dependent on a particular web page. When reproducing the bugs, the users navigated to different web pages, each one having different layouts and appearances. This means that the duplicate video-based bug reports appeared to be substantially different. Since Janus focuses more heavily on global GUI layout information, via its DINO+ViT model, Janus struggles to differentiate duplicates from non-duplicates. The local features learned by Tango seem to be useful for duplicate detection even when the duplicate videos show different layouts. The lower Janus mAP value on GNU is explained by the lower mAP values of Janus$_{vis}$ and Janus$_{txt}$ on that app (by 0.4% and 1.2%—see Table 2).

Janus's configuration "Vis + Txt + Seq" consistently shows mRR/mAP improvement in all nine apps except GNU, when compared to the baseline Tango. Across these apps, we observe improvements ranging from 6.8%/6.2% to 23.4%/26% mRR/MAP in the original dataset, and from 0.9%/1.8% to 7.5%/8.1% mRR/MAP in the extended dataset. This is interesting because the performance of the individual components of this configuration is substantially different across the apps. For instance, for TOK, the sequential aspect of the videos, individually combined with Janus$_{vis}$ or Janus$_{txt}$, is less effective than Tango, but when Janus$_{vis}$ and Janus$_{txt}$ are combined together with Janus$_{seq}$, Janus leads to substantial improvement (by 6.8%/6.2% mRR/mAP). Another example is the FCS app, which seems to benefit from the visual and sequential information, as Janus$_{vis}$ +Janus$_{seq-v}$ seems to contribute most to the overall performance of the " Vis + Txt + Seq" configuration. This suggests that the incorporation of sequential information enhances the Janus's ability to handle dynamic content, resulting in improved performance in comparison to its "Visual + Textual" configuration and the baseline Tango.

Best Janus configuration: The best performing Janus configuration is when combining visual (Janus$_{vis}$), textual (Janus$_{txt}$), and sequential information (Janus$_{seq}$) from video-based bug reports. This configuration consistently outperforms the baseline duplicate detector for 8/9 mobile apps. It achieves an overall performance of 89.8%/84.7 mRR/mAP, outperforming the baseline by 8.7%/9.2% mRR/mAP. This means that Janus can reduce the effort that developers spend determining if a new video-based bug report shows a known bug (by $(1.60 - 1.38)/1.38 = 16\%$, based on avg. rank), since they would need to inspect only 1.38 videos on average (*i.e.,* 1.38 avg. rank across all tasks) for finding the first duplicate video in the candidate duplicates suggested by Janus.

4.5 Qualitative Analysis

We discuss two qualitative examples that illustrate the validity of our hypothesis that the richer representations learned by Janus's transformer-based visual representation and OCR models improve duplicate detection for video-based bug reports.

4.5.1 ***Example 1: Transformer-based Representations Capture Subtle GUI patterns.*** To illustrate why we observed improvement in visual Janus as compared to visual Tango, we use interpretability techniques that generate saliency maps that help visualize the learned visual features. To visualize patterns learned by CNNs, we use a technique called AGF [39]. Although AGF can visualize self-supervised models such as SimCLR (used by the baseline), this requires training a supervised linear classifier after each layer and a dedicated algorithm to extract the segmentation information from their weights. Therefore, to simplify our comparison, instead of visualizing SimCLR directly, we visualize its main component, the ResNet-50 CNN using AGF under supervision. We follow past work and use the pre-trained ResNet-50 (on ImageNet [28]: the training dataset for ResNet) to generate the saliency map based on the class IDs with the highest probabilities for a given target GUI screen [39]. We further visualized the ViT-S/16 model (used by Janus$_{vis}$'s DINO) by directly displaying the self-attention maps.

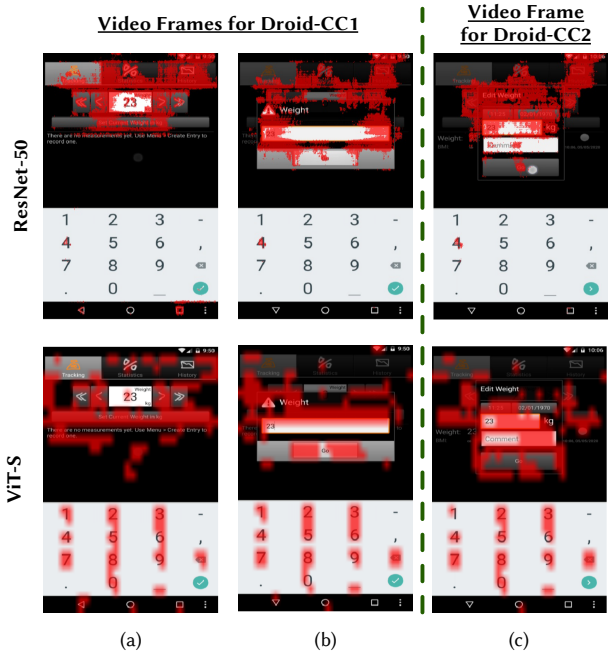

Figure 2: Visualization of ResNet-50 and ViT on keyframes of video-based bug reports

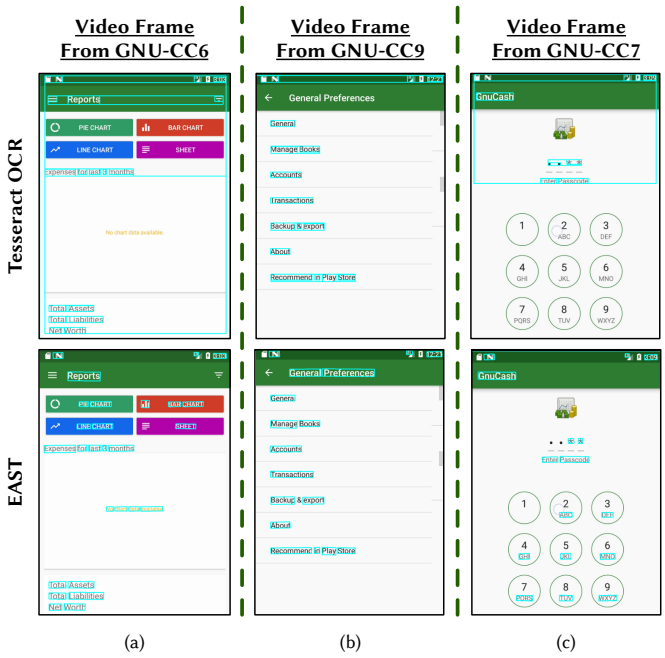

Figure 3: Bounding boxes localized by EAST and the Tesseract OCR library on keyframes of video-based bug reports

Visualization of ViTs does not necessitate sophisticated algorithms, given the inherent attention mechanism within these architectures.

In Fig. 2, we show three keyframes of two non-duplicate video-based bug reports from DroidWeight (DROID) [2]: DROID-CC1 and DROID-CC2. SimCLR fails to distinguish between the videos of these two bugs and mistakenly ranks DROID-CC2 as the first duplicate video of DROID-CC1. The DROID-CC1 video mainly has one trace that generates a new weight record by entering the weight on a pop-up component (Fig. 2 (b)), while the DROID-CC2 video not only includes the previous trace but also a trace that further edits the recorded weight on another different pop-up component (Fig. 2 (c)). Fig. 2 illustrates the saliency maps, overlaid over frames from two Droidweight video-based bug reports. We observe that the ViT-S/16 is able to attend to key parts of GUI components that ResNet-50 does not. Specifically, for the main screen (a) and *entering weight* screen (b) from videos of DROID-CC1 shown in Fig. 2, ResNet-50 and ViT-S/16 are all able to attend well to the objects, but ViT-S/16 pays more attention to the GUI layout information. However, for the *edit weight* screen (c) from videos of DROID-CC2, ResNet-50 has more difficulty in distinguishing between foreground pop-up components and the background. We can see it pays less attention to the lower edges and the bottom part of the foreground component. In contrast, ViT-S/16 effectively attends better to the edges and pays enough attention to the foreground component to help distinguish between (b) and (c), hence improving performance on this specific duplicate detection task.

From this example, there is a clear benefit to the visual nuances learned by ViTs. While here we present one example, after investigating several cases where Janus$_{vis}$ outperforms the baseline, we observed this pattern holds, wherein Janus$_{vis}$ learned visual representation is able to better capture nuanced visual patterns, such as the difference between two similar pop-ups, or the difference between background and foreground element when menus are displayed.

4.5.2 ***Example 2: Scene-based Text Detection Improves Text Localization.*** Textual Tango, which uses Tesseract OCR is unable to distinguish between similar video reports for a number of bugs, including three bugs from the GNUCash (GNU) app [3]. Therefore, we visualize the detection bounding boxes of text for three keyframes of these three videos in Fig. 3 for both Tesseract (first row) and EAST [81] (used by Janus). The first report for the GNU-CC6 bug has a main trace that goes to the `balance sheet` screen and checks the sub-account: we show one keyframe for this report in (Fig. 3-(a)), while the second video report for the GNU-CC9 bug navigates to the `General Preferences` screen, as shown in keyframe in (Fig. 3-(b)), and finally, the report for GNU-CC7 changes the password under the `General Preferences` menu, as shown in (Fig. 3-(c)). While these bugs are different, they include many similar screens where keywords are important for differentiation.

As observed in Fig. 3, EAST is more accurate than TesseractOCR for GUI component and text detection. In Fig. 3-(a), Tesseract OCR fails to localize the text on some buttons (*e.g.,* sheet) and the text in brighter colors (*e.g.,* Asset). Also, for the keyframe of GNU-CC9 (Fig. 3-(b)), Tesseract misses the text `General Preferences`, making it difficult to distinguish between report GNU-CC9 and GNU-CC7, as they both access various parts of the settings menu. In addition, Tesseract fails to detect the text when it is in low brightness and low contrast regions, including the text on the dialing circles (Fig. 3(c)), which also helps with differentiating between GNU-CC9 and GNU-CC7, since GNU-CC7 enters a passcode, but

GNU-CC9 only accesses the passcode settings. Thus, the more accurate text extraction of EAST clearly aids in the accurate extraction of key text that can help to differentiate between similar GUI screens.

5 THREATS TO VALIDITY

5.1 Internal and Construct Validity

Beyond the evaluation dataset, the implementation of Janus's models and experimental settings represent key validity threats. We controlled as many factors as possible for a fair comparison with the baseline. For instance, we implemented the 4-Codebook approach on both Janus and the baseline, used the same duplicate detection tasks, and measured their performance using well-known metrics in duplicate detection studies.

5.2 External Validity

To improve generalization, we created a new dataset to include ≈3k more duplicate detection tasks, for real bugs of different kinds, reported on mobile app issue trackers. These bugs were video recorded by multiple users on various mobile OS versions and did not include touch indicators. We ensured the recorded videos contained different reproduction scenarios for the same bugs. The decisions were made to make our dataset more comprehensive, realistic, and diverse. Our dataset could be improved by considering different app languages or other mobile platforms such as iOS.

6 RELATED WORK

6.1 Duplicate Textual Bug Report Detection

Many approaches have been proposed to detect duplicate textual bug reports to help developers avoid redundant effort during bug management. Most of the approaches leverage text retrieval techniques to obtain a ranked list of candidate duplicates for a query report [13, 64, 67, 69]. Some approaches leverage extra information (fields [63, 68, 76], contexts [12, 41], execution traces [73], *etc.*) and/or more effective similarity techniques (BM25F [68, 70], topic-modeling [58], word-embedding [76], *etc.*) to improve the detection. Wang *et al.* [71] proposed SETU, which combines textual bug descriptions with screenshots to detect duplicates, rather than focusing on video reports (as we do).

6.2 Automated GUI Understanding for SE

Various GUI understanding approaches have been proposed to help software engineering tasks related to mobile apps, such as GUI reverse engineering [15, 21, 57, 79], software testing [16, 56, 77], and GUI search [20, 22]. Most of them detect GUI elements first to understand GUI information. Chen *et al.* [24] show that deep learning-based object detection models (FasterRCNN [62], YOLOv3 [61], and centerNet [32]) and scene text detector EAST [81] outperform old-fashioned detection models [59] and OCR tool Tesseract [66] respectively. However, these models, based on supervised learning, leverage GUI information limited to a few GUI element categories, and the relationships between different elements are not considered, thus lacking an understanding of the entire screen. Fu *et al.* [36] therefore attempt to understand the whole screen by considering these relationships, based on the Transformer architecture to detect GUI elements more accurately.

The most closely related work to our own is Cooper *et al.'s* [26], which proposed the Tango duplicate detector and a dataset to evaluate it. While Janus leverages the same information from video-based bug reports as Tango does, there are key differences that set Janus apart. First, Janus learns visual features from app GUIs (via distillation and Vision Transformers) which capture GUI layouts more effectively for duplicate detection than Tango, which focuses on learning local GUI features (via contrastive learning and CNNs). Second, Janus learns textual representations of videos that are more useful for duplicate detection, by recognizing and extracting frame text more accurately (via fully neural models rather than heuristic+neural based approaches adopted by Tango). Third, Janus's sequential similarity computation, which attempts to align video frames, can be applied to both visual and textual representations, rather than to only visual representations as Tango does. Fourth, the best configuration of Janus combines all three modalities of video information (visual, textual, and sequential), and significantly outperforms the best Tango configuration, on duplicate detection tasks that include both injected and real bugs for a diverse set of mobile apps. Notably, our evaluation dataset is more comprehensive, realistic, and diverse than the one used to evaluate Tango.

7 CONCLUSIONS

To assist developers in identifying video-based bug reports that show identical mobile app bugs, we propose Janus, a new approach for duplicate video-based bug report detection. Janus leverages visual, textual, and sequential information from videos via the combination of representation learning, information retrieval, and frame-alignment approaches.

We evaluated Janus and found that it significantly outperforms an existing duplicate detector. The evaluation considered a new benchmark of 7,290 duplicate detection tasks based on 270 video-based bug reports, drastically extending a prior dataset (with real bugs as opposed to injected bugs from prior work). We conducted ablation experiments and an in-depth qualitative analysis visually showing that Janus learns a more interpretable hierarchical visual representation and localizes text regions more accurately.

ACKNOWLEDGMENTS

This work is supported in part by the following NSF grants: CCF-2311469, CCF-2007246, and CCF-1955853. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] [n. d.]. Tesseract OCR Library https://github.com/tesseract-ocr/tesseract/wiki.
[2] 2020. Droidweight https://test.f-droid.org/de/packages/de.delusions.measure/index.html.
[3] 2020. GnuCash https://github.com/codinguser/gnucash-android.
[4] 2023. Android apps for screen recording: https://www.androidauthority.com/best-screen-recording-apps-600838/.
[5] 2023. Android screenshot and video recording features: https://support.google.com/android/answer/9075928?hl=en.
[6] 2023. FDroid https://f-droid.org/en/.
[7] 2023. Firefox Focus https://github.com/mozilla-mobile/focus-android.
[8] 2023. GitHub video uploads: https://github.blog/2021-05-13-video-uploads-available-github/.
[9] 2023. GPSTest https://github.com/barbeau/gpstest.
[10] 2023. Images To PDF https://github.com/Swati4star/Images-to-PDF.
[11] 2023. Lucene's TFIDF Similarity Javadoc - https://tinyurl.com/ybhqqrqm.

[12] Anahita Alipour, Abram Hindle, and Eleni Stroulia. 2013. A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. https://doi.org/10.1109/msr.2013.6624026

[13] Sean Banerjee, Zahid Syed, Jordan Helmick, and Bojan Cukic. 2013. A fusion approach for classifying duplicate problem reports. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. https://doi.org/10.1109/issre.2013.6698920

[14] Hangbo Bao, Li Dong, Songhao Piao, and Furu Wei. 2021. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254* (2021).

[15] Tony Beltramelli. 2018. pix2code: Generating Code from a Graphical User Interface Screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. https://doi.org/10.1145/3220134.3220135

[16] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3377811.3380328

[17] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. 2020. Unsupervised learning of visual features by contrasting cluster assignments. *Advances in neural information processing systems* 33 (2020), 9912–9924.

[18] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. 2021. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*. 9650–9660.

[19] Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus. 2019. Reformulating queries for duplicate bug report detection. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. 218–229.

[20] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (Oct. 2020), 1–22. https://doi.org/10.1145/3415194

[21] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton. In *Proceedings of the 40th International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3180155.3180240

[22] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI Design Search through Image Autoencoder. *ACM Transactions on Software Engineering and Methodology* 29, 3 (June 2020), 1–31. https://doi.org/10.1145/3391613

[23] Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Extracting replayable interactions from videos of mobile app usage. *arXiv preprint arXiv:2207.04165* (2022).

[24] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination?. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. https://doi.org/10.1145/3368089.3409691

[25] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[26] Nathan Cooper, Carlos Bernal-Cárdenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 957–969.

[27] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*. 845–854.

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 248–255.

[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[30] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[31] Mingzhe Du, Shengcheng Yu, Chunrong Fang, Tongyu Li, Heyuan Zhang, and Zhenyu Chen. 2022. SemCluster: a semi-supervised clustering tool for crowdsourced test reports with deep image understanding. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1756–1759.

[32] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. 2019. CenterNet: Keypoint Triplets for Object Detection. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE. https://doi.org/10.1109/iccv.2019.00667

[33] Camilo Escobar-Velásquez, Michael Osorio-Riaño, and Mario Linares-Vásquez. 2019. Mutapk: Source-codeless mutant generation for android apps. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1090–1093.

[34] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.

[35] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. 2023. Read It, Don't Watch It: Captioning Bug Recordings Automatically. *arXiv preprint arXiv:2302.00886* (2023).

[36] Jingwen Fu, Xiaoyi Zhang, Yuwang Wang, Wenjun Zeng, Sam Yang, and Grayson Hilliard. 2021. Understanding Mobile GUI: from Pixel-Words to Screen-Sentences. *arXiv preprint arXiv:2105.11941* (2021).

[37] Otis Gospodnetic, Erik Hatcher, and Douglas R. Cutting. [n. d.]. Lucene in Action. *Manning Publications*.

[38] Jean-Bastien Grill, Florian Strub, Florent Altch'e, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Ávila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, and Michal Valko. 2020. Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning. *ArXiv* abs/2006.07733 (2020).

[39] Shir Gur, Ameen Ali, and Lior Wolf. 2021. Visualization of Supervised and Self-Supervised Neural Networks via Attribution Guided Factorization. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 13 (May 2021), 11545–11554. https://doi.org/10.1609/aaai.v35i13.17374

[40] Rui Hao, Yang Feng, James A Jones, Yuying Li, and Zhenyu Chen. 2019. CTRAS: Crowdsourced test report aggregation and summarization. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 900–911.

[41] Abram Hindle, Anahita Alipour, and Eleni Stroulia. 2015. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering* 21, 2 (June 2015), 368–410. https://doi.org/10.1007/s10664-015-9387-3

[42] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. *ArXiv* abs/1503.02531 (2015).

[43] MD Zakir Hossain, Ferdous Sohel, Mohd Fairuz Shiratuddin, and Hamid Laga. 2019. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CsUR)* 51, 6 (2019), 1–36.

[44] Zheng Huang, Kai Chen, Jianhua He, Xiang Bai, Dimosthenis Karatzas, Shijian Lu, and CV Jawahar. 2019. Icdar2019 competition on scanned receipt ocr and information extraction. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 1516–1520.

[45] Yu-Gang Jiang, Chong-Wah Ngo, and Jun Yang. 2007. Towards optimal bag-of-features for object categorization and semantic video retrieval. In *Proceedings of the 6th ACM international conference on Image and video retrieval*. 494–501.

[46] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 321–322. https://doi.org/10.1109/SANER53432.2022.00048

[47] Li Kang. 2017. *Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB*. Master's thesis.

[48] Dimosthenis Karatzas, Lluís Gómez i Bigorda, Anguelos Nicolaou, Suman K. Ghosh, Andrew D. Bagdanov, M. Iwamura, Jiri Matas, Lukás Neumann, Vijay Ramaseshan Chandrasekhar, Shijian Lu, Faisal Shafait, Seiichi Uchida, and Ernest Valveny. 2015. ICDAR 2015 competition on Robust Reading. *2015 13th International Conference on Document Analysis and Recognition (ICDAR)* (2015), 1156–1160.

[49] Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Joan Puigcerver, Jessica Yung, Sylvain Gelly, and Neil Houlsby. 2020. Big transfer (bit): General visual representation learning. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*. Springer, 491–507.

[50] Giorgos Kordopatis-Zilos, Symeon Papadopoulos, Ioannis Patras, and Ioannis Kompatsiaris. 2019. FIVR: Fine-grained incident video retrieval. *IEEE Transactions on Multimedia* 21, 10 (2019), 2638–2652.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[52] Hiroki Kuramoto, Masanari Kondo, Yutaro Kashiwa, Yuta Ishimoto, Kaze Shindo, Yasutaka Kamei, and Naoyasu Ubayashi. 2022. Do visual issue reports help developers fix bugs? a preliminary study of using videos and images to report issues on GitHub. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 511–515.

[53] Minghao Li, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. 2021. Trocr: Transformer-based optical character recognition with pre-trained models. *arXiv preprint arXiv:2109.10282* (2021).

[54] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. 2016. Enhancements for Duplication Detection in Bug Reports with Manifold Correlation Features. *Journal of Systems and Software* 121 (2016), 223–233.

[55] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[56] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl eyes. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM. https://doi.org/10.1145/3324884.3416547

[57] Kevin Moran, Carlos Bernal-Cardenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (Feb. 2020), 196–221. https://doi.org/10.1109/tse.2018.2844788

[58] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. https://doi.org/10.1145/2351676.2351687

[59] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ase.2015.32

[60] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.

[61] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[62] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 6 (June 2017), 1137–1149. https://doi.org/10.1109/tpami.2016.2577031

[63] Henrique Rocha, Marco Tulio Valente, Humberto Marques-Neto, and Gail C. Murphy. 2016. An Empirical Study on Recommendations of Similar Bugs. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE. https://doi.org/10.1109/saner.2016.87

[64] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE. https://doi.org/10.1109/icse.2007.32

[65] Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA.

[66] R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*. IEEE. https://doi.org/10.1109/icdar.2007.4376991

[67] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward interactive bug reporting for (android app) end-users. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 344–356.

[68] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. https://doi.org/10.1109/ase.2011.6100061

[69] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press. https://doi.org/10.1145/1806799.1806811

[70] Yuan Tian, Chengnian Sun, and David Lo. 2012. Improved Duplicate Bug Report Identification. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE. https://doi.org/10.1109/csmr.2012.48

[71] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology* 110 (June 2019), 139–155. https://doi.org/10.1016/j.infsof.2019.03.003

[72] Kai Wang, Boris Babenko, and Serge J. Belongie. 2011. End-to-end scene text recognition. *2011 International Conference on Computer Vision* (2011), 1457–1464.

[73] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press. https://doi.org/10.1145/1368088.1368151

[74] Tyler Wendland, Jingyang Sun, Junayed Mahmud, SM Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A dataset of manually-reproduced bug reports for android apps. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 600–604.

[75] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2023. JANUS Replication Package: https://doi.org/10.5281/zenodo.10455811.

[76] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. https://doi.org/10.1109/issre.2016.33

[77] Shengcheng Yu, Chunrong Fang, Yulei Liu, Ziqian Zhang, Yexiao Yun, Xin Li, and Zhenyu Chen. 2022. Universally Adaptive Cross-Platform Reinforcement Learning Testing via GUI Image Understanding. *arXiv preprint arXiv:2208.09116* (2022).

[78] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. 2023. Duplicate bug report detection: How far are we? *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–32.

[79] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. https://doi.org/10.1109/icse43902.2021.00074

[80] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. 2019. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems* 30, 11 (2019), 3212–3232.

[81] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 5551–5560.