applied sciences

MDPI

Article

Unsupervised Detection of Changes in Usage-Phases of a Mobile App

Hoyeol Chae [1,†], Ryangkyung Kang [2,†] and Ho-Sik Seok [1,*]

[1] Department of Computer Science and Engineering, Kangwon National University, Chuncheon-si, Gangwon-do 24341, Korea; chy4748@kangwon.ac.kr
[2] Apptest.ai, Songpa-gu, Seoul 05854, Korea; kimberlyrkkang@gmail.com
* Correspondence: hosik.seok@gmail.com; Tel.: +82-33-250-6381
† These authors contributed equally to this work.

Received: 17 April 2020; Accepted: 22 May 2020; Published: 25 May 2020

Abstract: Under the fierce competition and budget constraints, most mobile apps are launched without sufficient tests. Thus, there exists a great demand for automated app testing. Recent developments in various machine learning techniques have made automated app testing a promising alternative to manual testing. This work proposes novel approaches for one of the core functionalities of automated app testing: the detection of changes in usage-phases of a mobile app. Because of the flexibility of app development languages and the lack of standards, each mobile app is very different from other apps. Furthermore, the graphical user interfaces for similar functionalities are rarely consistent or similar. Thus, we propose methods detecting usage-phase changes through object recognition and metrics utilizing graphs and generative models. Contrary to the existing change detection methods requiring learning models, the proposed methods eliminate the burden of training models. This elimination of training is suitable for mobile app testing whose typical usage-phase is composed of less than 10 screenshots. Our experimental results on commercial mobile apps show promising improvement over the state-of-the-practice method based on SIFT (scale-invariant feature transform).

Keywords: automated mobile app testing; graph entropy; graph kernel; generative model; machine learning; unsupervised learning

1. Introduction

As users prefer mobile devices over conventional personal computers as a platform for news and entertainment, mobile apps now dominate software usage. However, developers have faced new challenges. Under the fierce competition and budget constraints, most developers do not have time for detecting bugs and potential crashes in their apps; thus, most apps are launched without sufficient testing. Testing technologies have yet to catch up, and mobile app testing still depends on manual methods, while reliable automated testing tools are rare [1].

In the domain of automated app testing, Google's Android Monkey tool is regarded as the state-of-the-practice for automated testing for the Android system [2]. Android Monkey takes a practical solution for GUI (graphical user interface) based testing: a random testing approach ("monkey testing") for generating random events [3,4]. Although the "monkey" approach is cost-effective, the unintelligent manner of testing leaves room for improvement. For example, recent developments in object recognition have been utilized for improving random testing [5]. In [6], a reinforcement learning based approach was proposed for identifying how individual UI widgets are interacting. Saumya et al. introduced the idea of the automatic generation of worst case test inputs from a model of program behavior in order to test programs under extreme loads [7]. More detailed descriptions on approaches for automated testing of mobile apps were introduced in [8]. Automated mobile app testing requires a number of functionalities such as feature extraction, event generation, event execution, etc. AI (artificial intelligence)/ML (machine learning) based approaches can augment or improve the performance of some functionalities for example by recognizing UI widgets in GUI and extracting keywords through OCR (optical character recognition).

In this paper, we introduce methods for detecting changes in mobile app usage based on GUI screenshots. The detection of changes in usage is important for generating input events and backtracking past executions. From the viewpoint of detecting changes in data streams, there already exist excellent works on concept drift [9,10]. However, the characteristics of mobile app testing make it difficult to employ existing achievements in the field of concept drift. Typical concept drift methods train models in order to measure the error rate or to estimate densities. In the field of mobile app testing, each usage-phase is typically composed of less than 10 images, and it is practically impossible to construct a training dataset due to time limits or flexible and inconsistent designs and implementations of the same functionalities. Thus, we are unable to utilize training based methods (supervised approaches).

In order to tackle this obstacle, we propose detection methods not requiring training (unsupervised approaches). Our detection methods compare GUI screenshots of a mobile app. In order to measure the difference between consecutive images, we utilize graph entropy [11], graph kernels [12], a probability distribution comparison metric, a generative model (Chapter 1 in [13]), and a sequence of log-likelihood values. Experimental results on 50 commercial apps report that the proposed methods achieve encouraging performance compared to the current state-of-the-practice method based on SIFT (scale-invariant feature transform) [14,15], and it is possible to detect changes in a data stream in an unsupervised manner.

The structure of the rest of this work is as follows. Section 2 summarizes the relevant works. Section 3 describes the details of the dataset and the proposed methods. Section 4 includes the experimental results. Finally, we give concluding remarks in Section 5.

2. Related Work

Many applications require the detection of significant changes in data streams such as video streams or streams of signals. The changes of interest can be detected by methods utilizing concept drift learning [9] or methods based on anomaly detection [16] in general. The goal of concept drift techniques is to mine inherent patterns from data streams by learning the underlying distribution over time [10]. In order to obtain inherent patterns, a typical procedure for drift detection is composed of data retrieval, data modeling, the calculation of test statistics, and hypothesis testing steps [10]. Although data retrieval for data streaming is also an important research topic [17], the key to drift detection is the measurement of dissimilarity. Gama et al. introduced a detection method based on the error rate of a learner in which once the error rate of a model reaches a threshold (the warning level), then the training step for a new learner is started [18]. For fast training, extreme learning machine (ELM) was introduced, whose architecture has a single hidden layer, and only the connections between the hidden layer and output layer are trained [19]. Xu et al. [20] conceived of a dynamic extreme learning machine improving ELM by adjusting the architecture of a model according to the classification performance of the current model.

The distance or dissimilarity between the distribution of the past and new data can be utilized to detect changes. In [21], the relative entropy was used to compare the dissimilarity between two distributions while the bootstrap method enhanced the statistical significance. In least-squares density-difference (LSDD) estimation, a density-difference model is fit to the target density-difference function through the squared loss [22]. For unknown distributions, the difference between two distributions represented by LSDD can be estimated by the Gaussian kernel function, thus enabling change detection [23]. Ross et al. improved traditional sequential monitoring methods through nonparametric charts capable of detecting arbitrary changes to the process distribution [24]. With an estimated distribution, a sequence of the likelihood of the data stream can be computed.

The distributions of log-likelihood were compared so as to detect changes by computing the symmetric Kullback–Leibler divergence [25].

The occurrence of changes in a data stream can be interpreted as anomalies [26]. Recently, graph based approaches have been actively researched [16]. In order to represent graphs as points in a non-Euclidean space and detect changes, adversarial training of autoencoders was employed for graph embeddings on constant-curvature manifolds, and change detection tests were performed considering embedded graphs [27]. Dissimilarity from prototype graphs was also utilized for change detection [28].

The test for detecting changes can be interpreted as a procedure for deciding to either reject a hypothesis that a new instance is generated from the current distribution or not reject it. The Student *t*-test quantifies how significant the difference between two datasets is (chap10 in [29]). For sequential data, the Mann–Kendall test and the CUSUM (cumulative sum) test are suitable [30]. With multiple parameters, the sequence of log-likelihood is a good indicator for change detection. For a log-likelihood stream, *t*-statistics, Kolmogorov–Smirnov, or Lepage can be utilized as the test statistic [25].

3. Materials and Methods

The term "usage-phase" denotes states comprised of a user's experience with a mobile app such as login, viewing goods, writing/reading a post, etc. In most cases, changes of usage-phases accompany changes in GUI compositions. Some changes in usage-phase are easy to detect. However, some changes are so subtle that they make automated testing difficult. For example, if a user reads a long review, dragging down causes changes in screen composition, but the sequence of slightly different images constitutes the same usage-phase semantically. Without semantic knowledge on the current usage, an automated tester is likely to decide that each image corresponds to a new usage-state and devise a complex testing strategy. The proposed methods aim to detect changes in usage-phases while minimizing the false-positive ratio. In order to achieve our goal in an unsupervised manner, we propose methods based on graphs and probability distributions in a screenshot. In Sections 3.1.1 and 3.1.2, we formulate the problem and describe our dataset. Section 3.2 describes the basic change detection algorithm and its proposed variations briefly. Section 3.3 discusses a method based on SIFT. Section 3.4 describes graph kernel and graph entropy based change detection methods. In Section 3.5, methods utilizing probability distributions are explained.

3.1. Experiment Description

3.1.1. Problem Statement

In this research, a user experience is represented as a stream of images, $\mathcal{D} = \{y_1, ...y_n\}$, where each image corresponds to a screenshot of the currently used mobile app. With a given distance metric $d(\cdot,\cdot)$, a change in usage-phase can be detected if the measured difference between two consecutive screenshots (y_i, y_{i+1}) is greater than a threshold (τ),

$$d(y_i, y_{i+1}) > \tau. \tag{1}$$

The choice of $d(\cdot,\cdot)$ and τ is important for implementing error-robust test systems. For the construction of error-robust test systems, we consider not $\mathcal{D}$, but a sequence of graphs converted from $\mathcal{D}$ (i.e., undirected labeled graphs characterized by nodes representing UI widgets and edges denoting relations between UI widgets) and a stream of probability distributions representing each screenshot. For conversion, UI widgets in y_i are recognized through Faster R-CNN [31].

3.1.2. Data

For our study, we collected 13,272 screenshots from 50 Android apps. After installing each app, a user used the installed app and produced screenshots utilizing the ADB (Android Debug Bridge) tool. In general, approximately 4 to 5 screenshots were sampled per second. After sampling, the user grouped a set of consecutive screenshots as a usage-phase manually. Details on the collected datasets are provided in Table 1.

Table 1. A list of tested apps and the number of usage-phase screenshots (numbers in parentheses mean the number of usage-phases manually determined/the averaged number of GUI screenshots per usage-phase).

4shared	**alba heaven**	**albamon**	**amazon**	**baedal**	**cars**	**cbs sports**	**cgv**
231	92	176	201	135	294	252	154
(34/6.8)	(8/11.5)	(20/8.8)	(35/5.7)	(8/16.9)	(42/7.0)	(18/14.0)	(36/4.3)
coupang	**deep booster**	**door dash**	**drink water reminder**	**ebay**	**emart**	**espn**	**facebook**
133	433	580	217	143	158	97	225
(19/7.0)	(55/7.9)	(187/3.1)	(22/9.9)	(24/6.0)	(13/12.2)	(15/6.5)	(27/8.3)
fish brain	**google translator**	**groupon**	**hawhae**	**home& shopping**	**kakao bus**	**korail**	**little caesars**
310	189	335	196	177	279	212	193
(40/7.8)	(32/5.9)	(47/7.1)	(20/9.8)	(46/3.8)	(59/4.7)	(40/5.3)	(48/4.0)
mcdonalds	**melon**	**messenger**	**my fitness**	**naver webtoon**	**news break**	**offerup**	**papago**
373	201	388	387	150	273	365	173
(69/5.4)	(11/18.3)	(53/7.3)	(49/7.9)	(17/8.8)	(25/10.9)	(32/11.4)	(27/6.4)
pininterest	**pluto tv**	**poshmark**	**roku**	**shareit**	**smart news**	**spotify**	**the weather channel**
394	299	348	333	458	263	263	195
(49/8.0)	(32/9.3)	(74/4.7)	(22/15.1)	(63/7.3)	(28/9.4)	(57/4.6)	(18/10.8)
today home	**triple**	**tubi**	**wayfair**	**wish**	**workout for woman**	**yanolja**	**yelp**
227	237	227	549	393	359	258	411
(31/7.3)	(72/3.3)	(14/16.2)	(96/5.7)	(68/5.8)	(56/6.4)	(63/4.1)	(76/5.4)
yeogieotae	**zumo**	**The mean of the manually-labeled usage-phases**					
208	128	265.44					
(44/4.7)	(31/4.1)						

3.2. Basic Change Detection Algorithm

Algorithm 1 provides the basic working of the proposed methods. In essence, a set of usage pages, $\mathcal{U}$, is constructed by computing Equation (1) on $\mathcal{D} = \{y_1, ...y_n\}$. If the computed value is larger than a threshold, τ, it is regarded that a change is detected. However, there exist variations of Algorithm 1 according to the employed methods and the selection of the threshold, τ. The threshold, τ, is determined by three methods: $\frac{Min+Max}{2}$, the mean, and the empirical threshold. In order to compute $\frac{Min+Max}{2}$, we observe differences between two consecutive screenshots in the current usage-phase, $\mathcal{U}_t$. "Min" and "Max" are the minimum and maximum difference in $\mathcal{U}_t$, respectively. Thus, in each usage-phase, $\mathcal{U}_t$, τ is adjusted dynamically. The "mean" is computed by firstly observing differences between consecutive screenshots in a mobile app, then computing the averaged differences. As a result, each app has a distinct "mean" value. In the case of empirical threshold, we fix τ by an empirically-determined number.

There are also variations on the computation of Equation (1). With the SIFT based method (Section 3.3), Equation (2) is utilized for computing Equation (1). For graph based methods (Sections 3.4.1 and 3.4.2), Algorithm 1 becomes more complicated. Firstly, the input, $\mathcal{D} = \{y_1, ...y_n\}$, is converted into a set of graphs, $\mathcal{G} = \{g_1, ...g_n\}$ (for more details, refer to Section 3.4). Then for graph entropy based detection (Section 3.4.1), the conditional graph entropy (Equation (4)) between two consecutive graphs g_i and g_{i+1} is utilized as a measure of difference between two corresponding GUI screenshots y_i and y_{i+1}. For graph kernel-based detection (Section 3.4.2), dissimilarity (measured by Equation (5)) between two graphs g_i and g_{i+1} is utilized as a measure of the difference between two corresponding GUI screenshots.

Algorithm 1: Basic change detection algorithm.

```
   Input  : $\mathcal{D} = \{y_1, ...y_n\}$ and a threshold $\tau$.
   Output: A set of usage-phases, $\mathcal{U} = \{U_1, ..., U_m\}$.
1  Initialize $t \leftarrow 1$.
2  Initialize $U_1$ with $y_1$.
3  for $i \leftarrow 1$ to $n - 1$ do
4      if $d(y_i, y_{i+1}) < \tau$ then
5          Add $y_{i+1}$ to $U_t$.
6      end if
7      else
8          $t \leftarrow t + 1$.
9          Initialize $U_t$.
10         Add $y_{i+1}$ to $U_t$.
11     end if
12 end for
```

For probability distribution based methods (Sections 3.5.1 and 3.5.2), the input, $\mathcal{D} = \{y_1, ...y_n\}$, is converted into a set of probability distributions, $\mathcal{P} = \{P_1, ...P_n\}$ (for more details, refer to Section 3.5). The Kullback–Leibler divergence between consecutive probability distributions P_i and P_{i+1} is computed to measure the difference between corresponding GUI screenshots y_i and y_{i+1} (Section 3.5.1). In Section 3.5.2, a generative model, M_t, for the current usage-phase, U_t, is constructed. Then, the likelihood of a new image y_{new} (exactly the likelihood of a converted probability P_{new}) is computed by Equation (9). By computing Equation (9), a sequence of likelihood values is obtained. We also attempt to detect changes by testing the hypothesis that a probability distribution representing y_{new} is generated by the current usage-phase model (M_t) based on the sequence of likelihood values (Section 3.5.3).

3.3. SIFT Based Method

The scale-invariant feature transform (SIFT) is utilized to detect local features in an image [32]. Because features extracted by SIFT are invariant to orientation, changes in illumination, and uniform scaling, SIFT is widely used for image and video analysis [33,34]. Target image search based on local features (TISLF) was introduced as a method for comparing target images and images in video sources by means of local features [34]. The TISLF is composed of a video segmentation step, a recognition step, and an estimation step. In TISLF, SIFT keypoints are utilized as a similarity measure between two consecutive images y_i and y_{i+1} by:

$$p(y_i, y_{i+1}) = \frac{\text{\# of matching points}}{\text{total \# of keypoints}}. \tag{2}$$

The value computed by Equation (2) is interpreted as the probability of two successive images belonging to the same interval. These values can be concatenated into the vector $W = (P(y_1, y_2), \ldots, P(y_{i-1}, y_i))$ representing the successive similarities over successive images. We utilize Equation (2) for detecting changes and denote it as the "SIFT" method in this paper. The SIFT based method is regarded as the state-of-the-practice method in this work.

3.4. Graph Based Methods

After recognizing UI widgets in a screenshot, a graph can be obtained from a set of recognized UI widgets and their corresponding coordinates. The given user experience, $\mathcal{D} = \{y_1, ...y_n\}$, is converted into a stream of graphs, $\mathcal{G} = \{g_1, ..., g_n\}$, by Prim's algorithm. For graph conversion, each recognized UI widget is considered as a node, and an edge between two nodes is determined with a label denoting the minimum distance between two UI widgets. In detail, a complete graph consisting of every recognized UI widget and node between every possible combinations of nodes is converted into a minimum spanning tree by Prim's algorithm. The resulting minimum spanning trees act as inputs for graph based detection methods. With the converted graphs, we can choose $d(\cdot,\cdot)$ in Equation (1) based on measures such as graph entropy [11] or graph kernels [12].

3.4.1. Graph Entropy Based Detection

Entropy based similarity can be interpreted as a measure of information needed to describe a distribution P utilizing another distribution Q [35]. Therefore, the conditional entropy, $H(X|Y)$, which is the required amount of information for quantifying the outcome of a random variable X given another random variable Y, gives a lead for the similarity between two graphs g_1, g_2. The graph entropy of a graph G and a random variable X is defined as:

$$H_G(X) = \min_{X \in W \in \Gamma(G)} I(W;X). \tag{3}$$

In Equation (3), $\Gamma(G)$ denotes the group of independent sets of G (here, a set of vertices of G is considered as independent if there exists no interconnection), $I(W;X)$ is the relative entropy where $I(W;X) = H(W) - H(W|X)$, and W takes values in $\Gamma(G)$. From the basic definition of graph entropy, the conditional graph entropy [11] is:

$$H_G(X|Y) = \min_{X \in W \in \Gamma(G)} I(W;X|Y). \tag{4}$$

In Equation (4), W, X, Y is a Markov chain, that is $p(w|x,y) = p(w|x)$. From Equation (3) and the definition of the relative entropy, we are able to compute the entropy of a graph g_i and compare two consecutive graphs.

3.4.2. Graph Kernel Based Detection

A kernel $k(x, x')$ is a similarity measure between x and x'. The role of a graph kernel is to evaluate similarity in the graph structure (an extensive study on graph kernels was provided in [36]). A graph can be interpreted as bags of vertices and edges. Then, the level of similarity between two graphs, G_1 and G_2, can be computed by comparing all pairs of labels of the vertex from g_1 and g_2,

$$k_{VL}(g_1, g_2) = \sum_{v_i \in g_1} \sum_{v'_r \in g_2} k(l(v_i), l(v'_r)) \tag{5}$$

where $l(v_i)$ denotes the label of vertex v_i and $k(\cdot,\cdot)$ is the equality indicator function. k_{VL} acts as a linear function of labels (of vertices) in two different graphs. Thus, two consecutive graphs can be compared by graph kernels, especially the vertex label histogram kernel [37].

3.5. Probability Distribution Based Methods

Each screenshot can be represented as a probability distribution of UI widgets and their connections. For a screenshot y_i, the probability distribution of y_i is defined as:

$$P(y_i) = (P(v_1), \cdots, P(v_n), P(e_1), \cdots, P(e_m)) \tag{6}$$

where n denotes the number of UI widget categories and m is the number of possible combinations of UI widgets (note: because the categories of UI widgets are fixed, we are able to generate the possible combination of nodes (UI widgets) and assign a unique identification label to each edge connecting two nodes). Based on Equation (6), a detecting method based on the Kullback–Leibler divergence (KLD) measure, a generative approach utilizing the likelihood measure, and a method through hypothesis testing are conceptualized.

3.5.1. KLD Based Detection

The Kullback–Leibler divergence, Equation (7), is a measure for comparing two probability distributions. For discrete probability distributions P and Q defined on a probability space $\mathcal{X}$, the Kullback–Leibler divergence is:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) log \left(\frac{P(x)}{Q(x)} \right). \tag{7}$$

Because we convert each screenshot into a probability distribution by Equation (6), Equation (7) is a natural choice for computing Equation (1). The procedure is simple: Firstly, two consecutive screenshots y_i and y_{i+1} are represented as probability distributions based on Equation (6). Then, the KLD value between two resulting probabilities, $P(y_i)$ and $P(y_{i+1})$, is computed by Equation (7).

3.5.2. Usage-Phase Model Based Detection

Once we detect usage-phase changes, we are able to collect screenshots ($\mathcal{U}_t = \{y_{t,1}, \cdots, y_{t,2}, y_{t,i}\}$; $y_{t,i}$ denotes the ith screenshot belonging to the tth usage-phase, $\mathcal{U}_t$) deemed to belong to a same usage-phase. With the probability distribution converted screenshots in $\mathcal{U}_t$, we build a model, M_t, for $\mathcal{U}_t$:

$$M_t = \left(P(v^{(1)}|\mathcal{U}_t), \cdots, P(v^{(n)}|\mathcal{U}_t), P(e^{(1)}|\mathcal{U}_t), \cdots, P(e^{(m)}|\mathcal{U}_t) \right) \tag{8}$$

where $P(v^{(k)}|\mathcal{U}_t) = \frac{1}{Z} \times \sum_{i=1}^{|\mathcal{U}_t|} v_{t,i}^{(k)}$ and $P(e^{(k)}|\mathcal{U}_t) = \frac{1}{Z} \times \sum_{i=1}^{|\mathcal{U}_t|} e_{t,i}^{(k)}$ (Z is a normalization constant to make the summation of all elements in M_t equal to one; $v_{t,i}^{(k)}$ and $e_{t,i}^{(s)}$ denote the number of the kth UI widget and the sth edge in the ith screenshot in $\mathcal{U}_t$, respectively). That is, $P(v^{(k)}|\mathcal{U}_t)$ is computed from the number of occurrence of the UI widget, $v^{(k)}$, in screenshots belong to $\mathcal{U}_t$. From Equation (8), we can compute the likelihood $L(y_{new}|M_t)$: the probability of a new screenshot, y_{new}, being sampled from a probability distribution represented by M_t. The log-likelihood $L(y_{new}|M_t)$ is computed by:

$$L(y_{new}|M_t) = logC_{M_t} + \sum_{k=1}^{n} logP(v^{(k)}|\mathcal{U}_t)^{v_{new}^{(k)}} + \sum_{k=1}^{m} logP(e^{(k)}|\mathcal{U}_t)^{e_{new}^{(k)}}, \tag{9}$$

where $C_{M_t} = \frac{(\sum_i v_{new}^{(i)} + \sum_j e_{new}^{(j)})!}{\prod_k v_{new}^{(k)}! \times \prod_l e_{new}^{(l)}!}$. $v_{new}^{(k)}$ and $e_{new}^{(l)}$ denote the number of the kth UI widget and the lth edge in y_{new}, respectively. If the computed likelihood is greater than a threshold, y_{new} is accepted as a member of $\mathcal{U}_t$, else y_{new} is regarded as a member of a new usage-phase, $\mathcal{U}_{t+1}$.

3.5.3. Hypothesis Testing Based Detection

By computing Equation (9) on $\mathcal{U}_t$, a set of likelihood values is obtained. We are able to determine whether to accept the hypothesis that a set of parameters representing an unknown distribution for y_{new} is equal to a set of parameters specifying $\mathcal{U}_t$ by running a likelihood ratio test,

$$\lambda = \frac{L(\hat{\Omega}_0)}{L(\hat{\Omega})} = \frac{\max_{\Theta \in \Omega_0} L(\Theta)}{\max_{\Theta \in \Omega} L(\Theta)}. \tag{10}$$

In order to compute Equation (10) in our setting, we regard the probability values in Equation (8) as a set of parameters specifying the hypothesis. Elements in a probability distribution (Equation (8)) produced by $\mathcal{U}_t$ constitute Ω_0. We calculate the maximum likelihood of the new screenshot y_{i+1}, which is observed after the screenshot in $\mathcal{U}_t$ ($g_{t,i}$) based on the assumed model described by Ω_0 (for more details on the likelihood ratio test, refer to [29]).

4. Results and Discussions

The performance of each method was compared based on datasets from 50 commercial mobile apps. In order to measure performance in terms of the fraction of correctly estimated changes and the fraction of detected changes, TP (true positive), TN (true negative), FP (false positive), and FN (false negative) were deployed. TP means a case in which the proposed method detects the changes successfully. TN is a case where the proposed method does not notice the changes in the same usage-phase. If the proposed method claims a change in a stream of screenshots in fact belongs to the same usage-phase, this is considered as FP. Finally, FN is a case where the proposed method fails to detect changes. The performance of each proposed method was quantified utilizing precision(precision $= \frac{TP}{TP+FP}$), recall(recall $= \frac{TP}{TP+FN}$), and accuracy(accuracy $= \frac{TP+TN}{TP+FP+TN+FN}$). Precision is a measure of the ratio of correctly detected changes, and recall is the fraction of detected changes. Accuracy measures the fraction of correct changes to estimations.

4.1. Overall Performance

Table 2 reports the overall performance of the proposed methods. In terms of precision and accuracy, graph based methods ("graph kernel" and "graph entropy" in Table 2) and probability distribution based methods ("KLD" and "likelihood" in Table 2) achieved better results compared to the SIFT based method and hypothesis testing based method when τ was determined dynamically. In terms of recall, however, the SIFT based method and the hypothesis testing based method reported better result than graph based methods or other probability distribution based methods. The difference in performance tells us that the SIFT based method and hypothesis testing based method were too sensitive to changes in the observed features. Thus, they detected more changes, but incorrectly.

Table 2. Change detection performance in 50 apps.

	Method	TP	FP	TN	FN	Precision	Recall	Accuracy
$(Min + Max)/2$	SIFT	684	1413	9787	1388	0.326	0.330	0.789
	Graph kernel	323	368	10,832	1749	0.467	0.156	0.840
	Graph entropy	369	682	10,518	1703	0.351	0.178	0.820
	KLD	264	428	10,772	1808	0.382	0.127	0.832
	Likelihood	260	432	10,768	1812	0.376	0.125	0.831
	Hypothesis testing	1084	5481	5719	988	0.165	0.523	0.513
Mean	SIFT	1031	4306	6894	1041	0.193	0.498	0.597
	Graph kernel	682	1550	9650	1390	0.306	0.329	0.778
	Graph entropy	953	2413	8787	1119	0.283	0.460	0.734
	KLD	778	2458	8742	1294	0240	0.375	0.717
	Likelihood	781	2533	8667	1,291	0.236	0.377	0.712
	Hypothesis testing	1085	5480	5720	987	0.165	0.524	0.513
Empirical threshold	SIFT	706	1141	10,059	1366	0.382	0.341	0.811
	Graph kernel	569	927	10,273	1503	0.380	0.275	0.817
	Graph entropy	1159	3921	7279	913	0.228	0.559	0.636
	KLD	538	1105	10,095	1534	0.327	0.260	0.801
	Likelihood	751	2214	8986	1321	0.253	0.362	0.734
	Hypothesis testing	1085	5482	5718	987	0.165	0.524	0.513

For detecting changes in usage-phases, reducing the ratio of misdetection(the cases of wrongly detecting changes in a usage-phase) is as important as increasing the ratio of the accuracy, precision, and recall. In order to measure performance in term of misdetection, we defined new measures of NPrecision (negative precision, NPrecision $= \frac{TN}{TN+FN}$) and NRecall (negative recall, NRecall $= \frac{TN}{TN+FP}$). Table 3 reports the performance in terms of NPrecision and NRecall. Each method reported similar performance in terms of NPrecision, but the graph based method ("graph kernel") and the method focusing on the difference between two probability distributions ("KLD") achieved better performance than others in terms of NRecall. From Tables 2 and 3, we concluded that (1) "graph kernel" based methods and the "KLD" based method achieved promising results compared to other methods, and (2) dynamically adjusted thresholds were better than fixed thresholds.

Table 3. Change detection performance in 50 apps: misdetection.

Method	Threshold	NPrecision	NRecall	Threshold	NPrecision	NRecall
SIFT	$(Min + Max)/2$	0.876	0.874	Mean	0.869	0.616
Graph kernel		0.861	0.967		0.874	0.862
Graph entropy		0.861	0.939		0.887	0.785
KLD		0.856	0.962		0.871	0.781
Likelihood		0.856	0.961		0.870	0.774
Hypothesis testing		0.853	0.511		0.853	0.511
SIFT	Empirical threshold	0.880	0.898			
Graph kernel		0.872	0.917			
Graph entropy		0.889	0.650			
KLD		0.868	0.901			
Likelihood		0.872	0.802			
Hypothesis testing		0.853	0.511			

4.2. Case Studies

Dividing a stream of screenshots into distinct usage-phases is not easy. Figure 1 provides an example. In this example, it is possible to group Figure (a), (b), and (c) into one usage-phase (because they represent searching) or make a group of Figure (b) and (c) as they show search results. Because it is ambiguous to make distinct groups from such screenshots in Figure 1, we refer to these cases as ambiguous cases and observe the performance considering ambiguous cases.

Table 4 reports the performance on datasets considering ambiguous cases. From 13,272 screenshots, seven-hundred thirty-six screenshots were designated as ambiguous cases. As expected, the performance on the ambiguous cases (numbers in parentheses in Table 4) was poor. The performance on datasets excluding ambiguous cases was slightly better than the reported performance in Table 2 because screenshots corresponding to ambiguous cases amounted to only 5.5% of the whole screenshots.

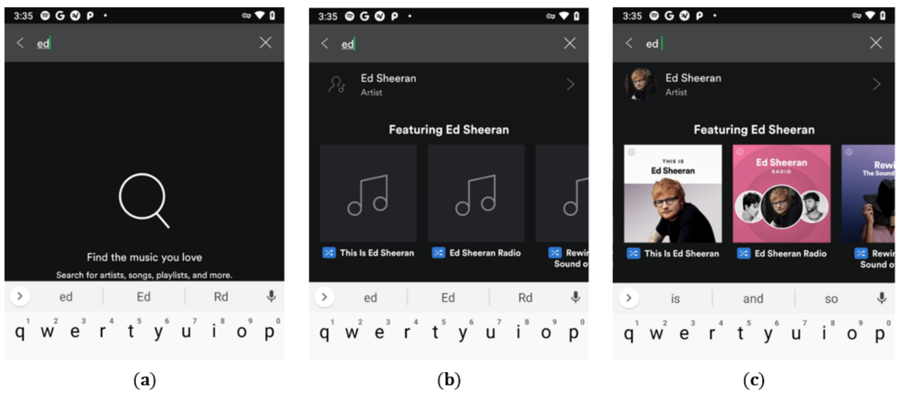

Figure 1. An example of the ambiguous division of usage-phases. (**a**) An initial search window. (**b**) Corresponding search result. (**c**) The same search result with images.

Table 4. Performance on the dataset excluding ambiguous cases (numbers in parentheses mean performance on the dataset of ambiguous cases).

	Method	TP	FP	TN	FN	Precision	Recall	Accuracy
(Min + Max)/2	SIFT	610 (74)	1401 (12)	9643 (144)	882 (506)	0.303 (0.860)	0.409 (0.128)	0.818 (0.296)
	Graph kernel	208 (115)	355 (13)	10,689 (143)	1284 (465)	0.369 (0.898)	0.139 (0.198)	0.869 (0.351)
	Graph entropy	294 (75)	672 (10)	10,372 (146)	1198 (505)	0.304 (0.882)	0.197 (0.129)	0.851 (0.300)
	KLD	190 (74)	442 (6)	10,622 (150)	1302 (506)	0.310 (0.925)	0.127 (0.128)	0.862 (0.304)
	Likelihood	187 (73)	426 (6)	10,618 (150)	1305 (507)	0.305 (0.924)	0.125 (0.126)	0.862 (0.303)
	Hypothesis testing	819 (265)	5401 (80)	5643 (76)	673 (315)	0.132 (0.768)	0.549 (0.457)	0.515 (0.463)
Mean	SIFT	905 (126)	4251 (55)	6793 (101)	587 (454)	0.176 (0.696)	0.607 (0.217)	0.614 (0.308)
	Graph kernel	480 (202)	1505 (45)	9539 (111)	1012 (378)	0.242 (0.818)	0.322 (0.348)	0.799 (0.425)
	Graph entropy	774 (179)	2379 (34)	8665 (122)	718 (401)	0.245 (0.840)	0.519 (0.309)	0.753 (0.409)
	KLD	577 (202)	2409 (43)	8635 (113)	915 (378)	0.193 (0.824)	0.387 (0.348)	0.735 (0.428)
	Likelihood	582 (200)	2485 (44)	8559 (112)	910 (380)	0.190 (0.820)	0.390 (0.345)	0.729 (0.424)
	Hypothesis testing	821 (265)	5403 (80)	5641 (76)	671 (315)	0.132 (0.768)	0.550 (0.457)	0.515 (0.463)
Empirical threshold	SIFT	629 (77)	1127 (14)	9917 (142)	863 (503)	0.358 (0.846)	0.422 (0.133)	0.841 (0.298)
	Graph kernel	392 (177)	891 (36)	10,153 (120)	1100 (403)	0.306 (0.831)	0.263 (0.305)	0.841 (0.404)
	Graph entropy	956 (203)	3869 (52)	7175 (104)	536 (377)	0.198 (0.796)	0.641 (0.350)	0.649 (0.417)
	KLD	406 (131)	1087 (18)	9957 (138)	1086 (449)	0.272 (0.879)	0.272 (0.226)	0.827 (0.365)
	Likelihood	549 (201)	2175 (34)	8869 (122)	943 (379)	0.202 (0.855)	0.368 (0.347)	0.751 (0.439)
	Hypothesis testing	819 (265)	5399 (80)	5645 (76)	673 (315)	0.132 (0.768)	0.549 (0.457)	0.516 (0.463)

Graph based methods require multiple nodes and edges to construct a graph. If a screenshot contains too few UI widgets (Figure 2) or the number of successfully recognized UI widgets is too small, graph based methods are likely to produce poor performance. Table 5 reports the performance on datasets excluding cases generating too small graphs (a graph with less than three nodes) and the performance on datasets corresponding to too small graphs (numbers in parentheses in Table 5). From Tables 2 and 5, we can see that the performance on datasets excluding small graphs was better than performance on datasets considering whole screenshots. However, the improvement in performance was not so impressive due to the small number of screenshots corresponding to small graphs (4.5% of all screenshots).

Besides precision, recall, and accuracy, the number of estimated usage-phases gave insight into the performance of the proposed methods. Table 6 provides the average number of estimated usage-phases (the number of manually labeled usage-phases is provided in Table 1). The performance in Table 6 informs us that methods based on SIFT (when utilizing the "mean" threshold), graph entropy (when utilizing the "empirical threshold"), and hypothesis testing were too sensitive.

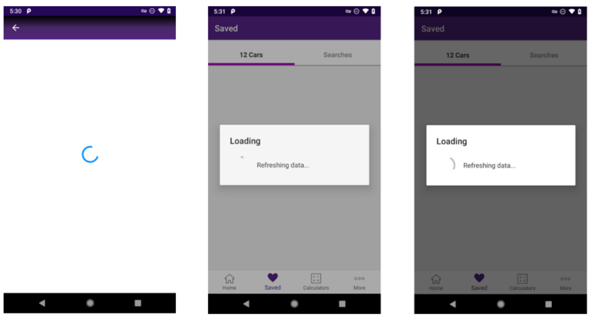

Figure 2. An example of screenshots generating too small graph.

Table 5. Performance on datasets capable of forming large enough graphs (numbers in parentheses mean performance on datasets forming relatively small graphs).

	Method	TP	FP	TN	FN	Precision	Recall	Accuracy
(Min + Max)/2	SIFT	621	1404	9515	1136	0.307	0.353	0.800
		(63)	(9)	(272)	(252)	(0.875)	(0.200)	(0.562)
	Graph kernel	254	351	10,568	1503	0.420	0.145	0.854
		(69)	(17)	(264)	(246)	(0.802)	(0.219)	(0.559)
	Graph entropy	342	661	10,258	1415	0.341	0.195	0.836
		(27)	(21)	(260)	(288)	(0.563)	(0.086)	(0.482)
	KLD	207	415	10,504	1550	0.333	0.118	0.845
		(57)	(13)	(268)	(258)	(0.814)	(0.181)	(0.545)
	Likelihood	203	420	10,499	1554	0.326	0.116	0.844
		(57)	(12)	(269)	(258)	(0.826)	(0.181)	(0.547)
	Hypothesis testing	983	5436	5483	774	0.153	0.559	0.510
		(101)	(45)	(236)	(214)	(0.692)	(0.321)	(0.565)
Mean	SIFT	907	4,220	6699	850	0.177	0.516	0.600
		(124)	(86)	(195)	(191)	(0.590)	(0.394)	(0.535)
	Graph kernel	585	1510	9409	1172	0.279	0.333	0.788
		(97)	(40)	(241)	(218)	(0.708)	(0.308)	(0.567)
	Graph entropy	862	2385	8534	895	0.265	0.491	0.741
		(91)	(28)	(253)	(224)	(0.765)	(0.289)	(0.577)
	KLD	680	2,407	8512	1077	0.220	0.387	0.730
		(99)	(45)	(236)	(216)	(0.688)	(0.314)	(0.562)
	Likelihood	684	2,484	8435	1073	0.216	0.389	0.720
		(98)	(45)	(236)	(217)	(0.685)	(0.311)	(0.560)
	Hypothesis testing	985	5438	5481	772	0.153	0.561	0.510
		(101)	(45)	(236)	(214)	(0.692)	(0.321)	(0.565)
Empirical threshold	SIFT	640	1129	9790	1117	0.362	0.364	0.823
		(66)	(12)	(269)	(249)	(0.846)	(0.210)	(0.562)
	Graph kernel	473	887	10,032	1284	0.348	0.269	0.829
		(96)	(40)	(241)	(219)	(0.706)	(0.305)	(0.565)
	Graph entropy	1064	3892	7027	693	0.215	0.606	0.638
		(95)	(29)	(252)	(220)	(0.766)	(0.302)	(0.582)
	KLD	458	1079	9840	1299	0.298	0.261	0.812
		(79)	(26)	(255)	(236)	(0.752)	(0.251)	(0.560)
	Likelihood	649	2164	8755	1108	0.231	0.369	0.742
		(101)	(45)	(236)	(214)	(0.692)	(0.321)	(0.565)
	Hypothesis testing	983	5434	5485	774	0.153	0.559	0.510
		(101)	(45)	(236)	(214)	(0.692)	(0.321)	(0.565)

Table 6. Average number of the estimated usage-phases.

Real	Method	Threshold	Estimated	Threshold	Estimated	Threshold	Estimated
41.44	SIFT	$\frac{Min+Max}{2}$	41.94	Mean	106.74	Empirical threshold	36.94
	Graph kernel		13.82		44.64		26.92
	Graph entropy		21.02		67.32		101.6
	KLD		13.84		64.62		32.84
	Likelihood		13.84		66.22		59.18
	Hypothesis testing		131.3		131.38		131.26

Compared to the average value of the number of the manually-labeled usage-phases (265.44), it seemed that the SIFT based method and hypothesis testing achieved better results. However, we should be cautious when comparing the number of detected usage-phases. SIFT and hypothesis testing based methods detected more usage-phases than other methods, but the usage-phases detected by these two methods contained more false positive cases.

In this research, we proposed various candidates for detecting changes in a mobile app's usage in an unsupervised manner. The empirical results in Tables 2 and 3 state that each method was relatively good at avoiding misdetection, but poor at detecting changes. Graph based methods utilizing the graph kernel and graph entropy and probability distribution based methods based on Kullback–Leibler divergence (KLD) and the usage-phase model (the "likelihood" method) achieved promising results in terms of accuracy. However, the performance from the perspective of precision and recall showed that we could not confirm the superiority of any method compared to other methods.

However, our work provided valuable insights for other researchers. The first insight was the need for a dynamically adjusted threshold. Our research confirmed that in order to detect changes, a threshold should be adjusted dynamically (Tables 2, 4, and 5). The empirical results showed that conventional change detecting methods based on hypothesis testing ([28,30]) were not suitable for detecting changes in usage-phases of a mobile app. The inferior performance of the hypothesis testing based method resulted from the lack of sufficient screenshots in each usage-phase. In our datasets, each usage-phase consisted of 6.0 screenshots on average. Thus, conventional hypothesis testing based methods may be unsuitable for app testing.

The second insight was the need for utilizing probability distributions resulting from screenshots. Although a graph kernel based method achieved better results overall, graph based methods had some deficiencies due to external causes. Firstly, current object recognition techniques are not perfect, so there existed some UI widgets that were unrecognized or wrongly recognized. Secondly, each screenshot from a mobile app contained a relatively small number of UI widgets. Thus, the failure of recognition could hinder the formation of graphs.

5. Conclusions

This paper presented change detection methods based on graph entropy, graph kernel, the Kullback–Leibler divergence, a generative model, and a hypothesis testing method. By utilizing recent advancements in object recognition, we were able to convert the input sequence of GUI screenshots into a sequence of graphs or probability distributions, thus constructing more robust change detection methods compared to the current state-of-the-practice method. The proposed methods detected changes in an unsupervised manner, not requiring training. This elimination of training requirements was a very significant advantage compared to the current change detection methods. Contrary to the datasets analyzed by existing change detection methods, our intended application, mobile app testing, typically had less than 10 GUI screenshots per usage-phase; thus, we could not obtain enough instances to train a model. In addition, the requirements of mobile app testing did not allow time for training. Our experimental results demonstrated that the proposed methods achieved promising results compared to the current state-of-the-practice method, but the result also clarified that the proposed methods should be improved before being employed for mobile app testing.

Our experience from this research and daily usage of mobile apps told us that we could not deny the existence of usage-phases in one app, but it was very difficult to define usage-phases in an objective manner and that the concept of the usage-phase was very ambiguous. In our future works, we will develop methods based on these findings. Rather than attempting to fix change-occurring points, we will search for micro usage-phases (composed of 2~3 screenshots), then combine these micro usage-phases into bigger clusters dynamically. Although this method is likely to be inferior to the proposed methods in this research in terms of memory cost, the hierarchical combination of micro usage-phases may overcome the inherent ambiguity of usage-phases. We also have plans to enhance the proposed methods by utilizing additional features such as event generation.

Author Contributions: Conceptualization, H.-S.S.; methodology, H.-S.S.; software, H.C. and R.K.; validation, H.C., R.K., and H.-S.S.; formal analysis, H.-S.S.; investigation, H.C., R.K., and H.-S.S.; data curation, H.C. and R.K.; writing, original draft preparation, H.-S.S.; writing, review and editing, H.-S.S.; visualization, H.C. and R.K.; supervision, H.-S.S.; project administration, H.-S.S.; funding acquisition, H.-S.S. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A1B07047156).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial intelligence
CUSUM	Cumulative sum
ELM	Extreme learning model
FN	False negative
FP	False positive
GUI	Graphical user interface
KLD	Kullback–Leibler divergence
LSDD	Least-squares density-difference
ML	Machine learning
NPrecision	Negative precision
NRecall	Negative recall
OCR	Optical character recognition
R-CNN	Regions with convolutional neural network
SIFT	Scale-invariant feature transform
TISLF	Target image search based on local features
TN	True negative
TP	True positive
UI	User interface

References

1. Mao, K.; Harman, M.; Jia, Y. Sapienz: Multi-objec tive automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 94–105.
2. Google. Android Monkey. 2017. Available online: https://developer.android.com/studio/test/monkey (accessed on 25 May 2020).
3. Wetzlmaier, T.; Ramler, R.; Putschögl, W. A framework for monkey GUI testing. In Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation, Chicago, IL, USA, 11–15 April 2016; pp. 416–423.
4. Nyman, N. Using monkey test tools. *STQE* **2000**, *29*, 18–23.
5. White, T.D.; Fraser, G.; Brown, G.J. Improving random GUI testing with image-based widget detection. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 307–317.

6. Degott, C.; Borges, N.P., Jr.; Zeller, A. Learning user interface element interactions. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 296–306.
7. Saumya, C.; Koo, J.; Kulkarni, M.; Bagchi, S. XSTRESSOR: Automatic generation of large-scale worst-case test inputs by inferring path conditions. In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation, Xi'an, China, 22–27 April 2019; pp. 1–12.
8. Moran, K.; Linares-Vásquez, M.; Bernal-Cárdenas, C.; Vendome, C.; Poshyvanyk, D. Automatically discovering, reporting and reproducing android application crashes. In Proceedings of the IEEE International Conference on Software Testing, Verification and Validation, Chicago, IL, USA, 10–15 April 2016; pp. 33–44.
9. Iwashita, A.S.; Papa, J.P. An overview on concept drift learning. *IEEE Access* **2019**, *7*, 1532–1547. [CrossRef]
10. Lu, J.; Liu, A.; Dong, F.; Gu, F.; Gama, J.; Zhang, G. Learning under concept drift: A review. *IEEE Trans. Knowl. Data Eng.* **2019**, *31*, 2346–2363. [CrossRef]
11. Orlitsky, A.; Roche, J.R. Coding for computing. *IEEE Trans. Inf. Theory* **2001**, *47*, 903–917. [CrossRef]
12. Vishwanathan, S.V.N.; Schraudolph, N.N.; Kondor, R.; Borgwardt, K.M. Graph kernels. *J. Mach. Learn. Res.* **2010**, *11*, 1201–1242.
13. Bishop, C.M. *Pattern Recognition and Machine Learning*; Springer: New York, NY, USA, 2006; pp. 42–43.
14. Li, X.; Hu, W.; Shen, C.; Zhang, Z.; Dick, A. A survey of appearance models in visual object tracking. *ACM Trans. Intell. Syst. Technol.* **2013**, *58*, 1–48. [CrossRef]
15. Wang, Y.; Du, L.; Dai, H. Unsupervised SAR image change detection based on SIFT keypoints and region information. *IEEE Geosci. Remote Sens. Lett.* **2016**, *13*, 931–935. [CrossRef]
16. Akoglu, L; Tong, H.; Koutra, D. Graph-based anomaly detection and description: A survey. *Data Min. Knowl. Discov.* **2015**, *29* 626–688. [CrossRef]
17. Ramírez-Gallego, S.; Krawczyk, B.; García, S.; Woźniak, M.; Herrera, F. A survey on data processing for data stream mining: Current status and future directions. *Neurocomputing* **2017**, *239*, 39–57. [CrossRef]
18. Gama, J.; Medas, P.; Castillo, G.; Rodrigues, P. Learning with drift detection. In *Advances in Artificial Intelligence—SBIA 2004. SBIA 2004. Lecture Notes in Computer Science*; Bazzan, A.L.C., Labidi, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3171, pp. 286–295.
19. Huang, G.; Huang, G.-B.; Song, S.; You, K. Trends in extreme learning machines: A review. *Neural Netw.* **2015**, *61*, 32–48. [CrossRef]
20. Xu, S.; Wang, J. Dynamic extreme learning machine for data stream classification. *Neurocomputing* **2017**, *238*, 433–449. [CrossRef]
21. Dasu, T.; Krishnan, S.; Venkatasubramanian, S.; Yi, K. An information-theoretic approach to detecting changes in multi-dimensional data streams. In Proceedings of the 38th Symposium on the Interface of Statistics, Computing Science, and Applications, Pasadena, CA, USA, 24–27 May 2006; pp. 1–24.
22. Nguyen, T.D.; Du Plessis, M.C.; Kanamori, T.; Sugiyama, M. Constrained least-squares density-difference estimation. *IEICE Trans. Inf. Syst.* **2014**, *97*, 1822–1829. [CrossRef]
23. Bu, L.; Alippi, C.; Zhao, D. A PDF-free change detection test based on density difference estimation. *IEEE Trans. Neural Netw. Learn. Syst.* **2018**, *29*, 324–334. [CrossRef] [PubMed]
24. Ross, G.J.; Adams, N.M. Two nonparametric control charts for detecting arbitrary distribution changes. *J. Qual. Technol.* **2012**, *44*, 102–116. [CrossRef]
25. Alippi, C.; Boracchi, G.; Carrera, D.; Roveri, M. Change detection in multivariate datastreams: Likelihood and detectability loss. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, New York, NY, USA, 9–15 July 2016; pp. 1368–1374.
26. Ranshous, S.; Shen, S.; Koutra, D.; Harenberg, S.; Faloutsos, C.; Samatova, N.F. Anomaly detection in dynamic networks: A survey. *WIREs Comput. Stat.* **2015**, *7*, 223–247. [CrossRef]
27. Grattarola, D.; Zambon, D.; Alippi, C.; Livi, L. Change detection in graph streams by learning graph embeddings on constant-curvature manifolds. *arXiv* **2019**, arXiv:1805.06299v3.
28. Zambon, D.; Alippi, C.; Livi, L. Concept drift and anomaly detection in graph streams. *IEEE Trans. Neur. Netw. Learn. Syst.* **2018**, *29*, 5592–5605. [CrossRef]
29. Wackerly, D.D.; Mendenhall, W., III; Scheaffer, R.L. Likelihood ratio tests. In *Mathematical Statistics with Applications*; Thomson Brooks/Cole: Belmont, NV, USA, 2008; pp. 549–550.
30. Alippi, C.; Roveri, M. An adaptive CUSUM-based test for signal change detection. In Proceedings of the International Symposium on Circuits and Systems, Island of Kos, Greece, 21–24 May 2006; pp. 5752–5755.

31. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.* **2016**, *39*, 1137–1149. [CrossRef]
32. Lowe, D.G. Object recognition from local scale-invariant features. In Proceedings of the International Conference on Computer Vision, Kerkyra, Greece, 20–27 September 1999; pp. 1150–1157.
33. Park, M.-H.; Park, R.-H.; Lee, S.W. Shot boundary detection using scale invariant feature matching. In *Proceedings of SPIE Visual Communications and Image Processing*; SPIE: Bellingham, WA, USA, 2006; pp. 569–577.
34. Guan, B.; Ye, H. Target image video search based on local features. *arXiv* **2019**, arXiv:1808.03735v2.
35. Korhonen, A.; Krymolowski, Y. On the robustness of entropy-based similarity measures in evaluation of subcategorization acquisition systems. In Proceedings of the 6th Conference on Natural Language Learning (CoNLL-2002), Taipei, Taiwan, 31 August–1 September 2002; pp. 1–7.
36. Kriege, N.M.; Johansson, F.D.; Morris, C. A survey on graph kernels. *Appl. Netw. Sci.* **2020**, *5*, 6. [CrossRef]
37. Sugiyama, M.; Borgwardt, K.M. Halting in Random Walk Kernels. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 1630–1638.

Sample Availability: Samples of the GUI screenshots are available from the authors.