GUI Element Identification with Semantic Mapping

Evanfiya Logacheva 2001779
Master's thesis in Information Technology
Supervisor: Ivan Porres Paltor
The Faculty of Science and Engineering
Åbo Akademi University
2023

Abstract

User Interface test automation faces significant obstacles due to test failures connected to application changes. Additionally, current User Interface testing methods are not context aware and usage-based, which makes exploring web application functionality challenging. Robots used for crawling web application interfaces are slow and do not reflect human interaction with them. Semantic mapping (semantic matching) has been proposed as a method for reusing existing tests between web applications in the same domain to mitigate issues with testing speed and context awareness. This thesis explores semantic mapping for robust User Interface element identification that could alleviate the issue with test failures upon application changes.

Semantic mapping uses textual cues of User Interface elements neighboring testable features to identify similar features in other applications of the same domain. This work argues that the same technique can be applied to various versions of the same web application. Existing tools leverage text attributes of features' neighbors based on the hierarchy and position of an element, while this study applies semi-supervised learning methods to extract relevant text from elements surrounding features. It uses state-of-the-art pretrained language models for embedding textual cues. To find similar features, it uses cosine similarity between sentences as a measure of semantic similarity.

This implementation of semantic matching has demonstrated promising results for User Interface element identification between two versions of the same web application.

Keywords: semantic mapping, semantic matching, user interface testing, test automation, machine learning.

Preface

This thesis was written as part of the AIDOaRt project under the supervision of Professor Ivan Porres Paltor. I would like to thank Professor Ivan Porres Paltor for providing guidance and support during the writing process.

Evanfiya Logacheva

May 2, 2023

Contents

1 Introduction

1.1 Motivation

This thesis investigates the topic of Graphical User Interface (GUI) element identification for GUI testing. The primary goal of the study is to explore methods that could reduce time and effort associated with GUI testing. Studies [1]–[3] indicate that manual GUI testing is prevalent despite its high cost, which is explained by intensive effort required for automated test maintenance. In addition, many existing techniques for automated testing are time-consuming [4], [5] and not usage-based [4], [6], which leads to some functionality of applications being left unexplored. For example, users asked to make an online purchase would follow a simple route of adding products to their cart and paying for their order. Robots used for testing web applications are not capable of constructing such a course of actions on their own; instead, they exhaustively crawl applications by interacting with available GUI elements until they succeed in finding a path that leads them to completing the purchase [4], [5]. Such time-consuming crawling does not reflect how users interact with applications. Rau et al. propose a method of semantic mapping (semantic matching) for reusing existing tests between web applications in the same domain to mitigate issues with testing speed and context awareness [4], [5]. They suggest that once there is a test suite that includes a set of interactions for testing a certain functionality, it can be adapted for another application that has the same functionality, e.g., tests for one online store can be reused for another.

Nass et al. [2] argue that there are many challenges that prevent full automation of GUI tests, one of which is test execution failure after a system under test (SUT) has been altered. Tests fail due to changes in GUI element locations, and researchers suggest that robust identification of GUI elements could remedy existing obstacles [2]. Semantic

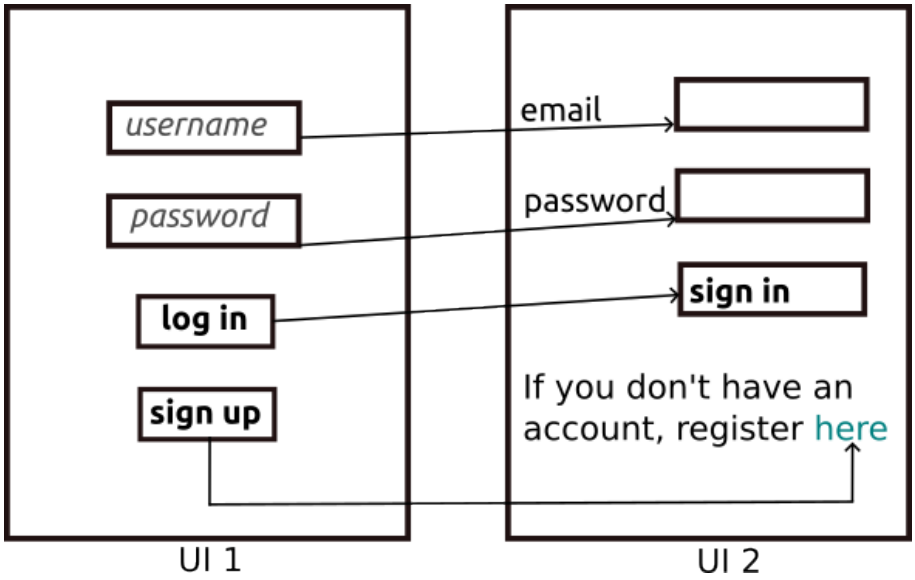


Figure 1: An example of semantically identical web elements

mapping, which is based on semantic similarity between GUI elements, could alleviate the issue of text execution failure upon SUT changes. It allows context-aware test transfer between different applications in the same domain [4], [5], [7]–[10], and it is especially successful when used with applications developed by the same company [6]. Figure 1 provides an example of two user interfaces that allow users to authorize into their account or create a new one. Since textual features in both of them are semantically similar, for example, *log in* has the same meaning here as *sign in*, it is possible to identify these elements as sharing the same functionality. Natural language processing (NLP) provides means to represent semantic meaning of texts and measure the strength of semantic similarity between them. Using NLP methods, we can leverage textual content of web elements for semantic mapping by finding the most semantically similar elements. It is then assumed that such elements also share similar functionality and can be validated using the same GUI tests.

In this work, semantic mapping is proposed for matching GUI element locators between different versions of the same web application within the framework of Continuous Integration, Delivery, and Deployment (CI/CD). The CI/CD approach emphasizes reduction of human effort in software testing through automation [11]. Semantic mapping has the potential for solving certain issues of GUI test automation maintenance and decreasing associated manual effort through test reuse.

Semantic mapping can be described as a two-step procedure. The initial step involves identifying what GUI features are to be tested and grouping remaining GUI elements around them for the purpose of extracting text attributes. Existing techniques for obtain-

ing attributes rely on visual web page segmentation [4], [5] and leveraging neighboring and hierarchical structure of applications [7]–[9], [12]. The procedure may be improved with an introduction of machine learning methods for GUI element grouping and text attribute extraction. The second step measures semantic similarity between text attributes to identify the most semantically similar units. Previous studies use older language models to represent text attributes, while this work relies on state-of-the-art contextual embeddings.

This thesis aims to answer the following questions:

1. Are machine learning methods suitable for extracting textual cues from GUI elements?
2. Do sentence embeddings produced by pre-trained language models provide adequate representation of web element attributes?
3. Is it possible to identify web elements by finding semantically similar groups within different versions of a web application?

1.2 Structure of the Thesis

Chapter 2 discusses current research in the field of UI testing and obstacles to its automation. Chapter 2 introduces research on semantic mapping for GUI testing related to this work. It outlines architecture and methods used in existing tools as well as their evaluation metrics. It provides a brief introduction of machine learning methods used in this study: semi-supervised learning algorithms, word and sentence embeddings, and the notion of semantic similarity. Thus, Chapter 2 includes background information related semantic mapping.

Chapter 3 describes the design of a semantic mapping implementation step by step and explains technologies involved in it. It discusses the results of its application on the example of a small web application and explains the difficulties involved in the execution of the experiment. Chapter 4 summarizes the findings of the study and introduces suggestions for future research on semantic mapping.

2 Background

2.1 Automated GUI Testing

Banerjee et al. [13] define GUI testing as examining the functionality of a front-end side of an application by following a sequential order of events through interaction with GUI widgets. Although testing can be performed manually or automatically, all testing techniques represent sampling of input space, since it contains a large number of executable event sequences [13]. The authors also state that GUI test oracles are essentially based on sampling of output space, because its evaluation as a whole is impractical. Automated testing seeks to emulate human interaction with SUT through software or robotics (or their combination) and should reduce high cost and effort associated with manual testing [2]. However, automated tests often break and require alterations anytime applications undergo changes [1]. As a consequence, testers might choose manual testing over automated scripts when they are faced with a tight deadline [1].

In spite of a growing body of research on GUI test automation, manual GUI testing is still practiced, which indicates numerous obstacles to test automation [2], [3], [13]. Nass et al. [2] have reviewed 49 publications on automated GUI testing and identified common issues encountered by researchers. The challenge mentioned in the article that is central to this work is preventing test execution from breaking after application changes. The authors mention seventeen publications released in the 2010s that report this issue. They consider this challenge to be essential due to the fact that large-scale SUT alterations should break test execution. Nass et al. [2] divide SUT changes into intentional, in case of which tests should be adjusted to them, and unintentional, which are caused by deliberate modifications but remain unaccounted for in tests. Since small changes can break automated tests while remaining obscure to testers, the researchers suggest that

future work should concentrate on solving issues with unintentional changes.

Nass et al. [2] suggest that robust identification of GUI widgets could improve the existing situation with test breaks appearing after SUT changes. They consider some failures to be anticipated, for example, if a widget is missing, while others, e.g., if a widget is located in another place, are not. Certain proposed solutions for improved accuracy of GUI element identification focus on harnessing information of surrounding GUI widgets. Nguyen et al. [14] propose a method to locate a target GUI widget using its own attributes (unique IDs and anchor texts, textual attributes of images and links) or its neighbors' locators (a candidate locator without numbers or a closest locator containing unique text). Their experiments show that applications whose structure is stable across versions benefit from their approach; however, when GUI widgets do not contain the same attributes and are relocated from their initial component, the method fails. They also mention that the approach is not successful when neighbors' text changes along with GUI widget attributes. Another solution that leverages surrounding GUI widgets is proposed by Yandrapally et al. [15]. Their method relies on contextual cues, which are essentially neighboring labels, e.g., text values of buttons and links. It is, similarly to the technique proposed by Nguyen et al. [14], vulnerable to label changes and DOM reorganization.

In addition to the challenges described above, Lin et al. [9] and Rau et al. [4], [5] argue that current testing approaches are not context-aware but use random inputs, which can prevent exploration of SUT's functionality. Rau et al. [4], [5] note that exhaustive crawling of web applications is time-consuming and not representative of how a human would use an application. They state there is a slim chance that crawling results in discovering a sequence of actions that completes a meaningful task. They provide an example of how easy it is for users to make an online order by following well-known steps: adding products to a cart, pressing a checkout button, and paying for it. Crawlers are not aware of the steps. Instead, they interact with GUI elements until they find a path that leads them to a desired outcome, which is completing the order in this particular case.

Semantic mapping is a technique proposed to mitigate certain challenges associated with GUI testing. It relies on semantic similarity between textual attributes of GUI widgets. Since it employs existing GUI tests to generate new ones, it makes test inputs more contextualized when compared to exhaustive GUI crawling [4], [5]. As semantic mapping is not based on unique locators of GUI widgets, it can also help mitigate issues with automated GUI testing that appear as a result of SUT changes.

2.2 Semantic Mapping for GUI Testing

Applications in the same domain often share functionality, which is why their GUIs are similar. This makes it possible to transfer GUI tests from one application to another or use them to generate new tests [4]–[9], [12]. For example, many applications offer users to log into their account or create one. In Figure 1, there are two login forms. Based on the assumption of similarity between their GUI elements, proposed semantic mapping tools [4], [5], [7]–[9], [12] can leverage textual data to match the elements in UI 1 with those elements in UI 2 they have the highest similarity score with, e.g., the *Sign up* button with the *register here* link.

Tool	**Application**	**Features**	**Similarity Measure**	**Language Model**
Poster [4], [5]	Web	Labels grouped with VIPS [16]	Normalized pair-wise cosine similarity between word vectors	Word2vec
GUITestMigrator (GTM), AppTestMigrator (ATM) [7], [8]	Android	Attributes derived from neighbors determined by proximity	Normalized pair-wise cosine similarity or edit distance scores between word vectors	WordNet, Word2vec
CRAFTDROID [9]	Android	Attributes derived from parent and sibling elements	Normalized pair-wise cosine similarity between word vectors	Word2vec
SemFinder [12]	Android	Attributes derived from neighboring, parent, and sibling elements	Similarity between sentence embeddings of concatenated attributes	Various word and sentence embedding techniques

Table 1: Comparison of GUI tools that use semantic mapping for test reuse and transfer

Table 1 describes existing semantic similarity tools discussed in this section. Research by Rau et al. [4], [5] is dedicated to GUI test transfer for web applications. The semantic mapping technique introduced by Rau et al. starts with an existing web application test. A source application, from which tests are transferred, contains states that are created

by individual steps of the test. Semantic features, which are interactive GUI widgets and surrounding them text labels, are compared with those in a target application, which receives tests from the source application. To extract semantic features, Rau et al. use a visual page segmentation algorithm (VIPS) [16] to group widgets and related labels. The authors mention that a threat to validity of their work lies in the feature identification, since there are features without text attributes that use image or iconic content instead. What they do not discuss is the performance of VIPS – a widget can potentially be placed into an incorrect feature group, which would then hinder the accuracy of their semantic mapping algorithm.

GTM, ATM, CraftDroid, and SemFinder are meant for Android application test reuse (Table 1). ATM is a more recent technique based on GMT. The tools are built on the same premise of applications in the same domain sharing comparable GUIs, which makes it possible to migrate test cases from one application to another. Similarly to Poster, ATM computes a static graph where nodes are windows and edges are transitions between them. CraftDroid also statically analyzes an application to create a transition graph describing how the application's Activities[1] interact with each other. It traverses the meta data in Resource files[2] to identify statically created GUI widgets and the source code of Activities/Fragments[3] to identify dynamically created GUI widgets. GTM [7] analyzes the hierarchical structure of each screen, flags interactable elements, and stores each screen as a set of features. SemFinder [12] uses both neighboring and hierarchical approaches to extract information.

The next step of semantic mapping is matching events in the source application and the target application. In the articles by Rau et al. [4], [5], this step is called the *feature mapping phase*, where every feature in the source application is evaluated against features in the target application using cosine similarity (9). Text labels are preprocesssed, stemmed, and lemmatized, stop words are removed from consideration. Similarity is calculated for each individual word only once, word order is disregarded. Resulting similarity scores are then normalized to be within the interval $[-1, 1]$. GMT [7] determines similarity based on the type of an event's action and its target. If there is no corresponding action in the target application, GMT assigns 0 to its similarity score. To compare the target and source applications further, GMT extracts neighboring labels for non-image elements and meta data

[1] "An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI" [17].

[2] "Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more" [18].

[3] "A Fragment is a piece of an application's user interface or behavior that can be placed in an Activity" [19].

for image elements. ATM [8] also extracts surrounding labels and other attributes for all elements and, additionally, filenames for image elements. ATM uses an ontology based on Word2Vec, which is pre-trained on user manuals for mobile applications, to produce embeddings for similarity evaluation. The similarity score is based on cosine (9) and edit distance [20], which is used for measuring string similarity. CraftDroid [9] also leverages additional information about GUI widgets from its attributes and neighboring elements to compute similarity. Likewise, it uses Word2Vec and cosine (9) similarity; however, it normalizes obtained cosine scores between widgets so that GUI widgets that are closer to the currently considered state are given a higher overall score. Lin et al. explain their decision to normalize textual similarity scores with the idea that the number of steps that are required to test a specific functionality should be roughly the same in similar applications. Generally, all the techniques use GUI widgets' neighbors to extract additional data, which is proposed by Yandrapally et al. [15]. To summarize, while Poster, ATM, and CraftDroid use Word2Vec for word embeddings, GMT uses WordNet[4]. According to Church [21], although not producing the best results in terms of accuracy, Word2Vec has become popular due to its accessibility and simplicity. Thus, the proposed tools could be enhanced by choosing a more advanced method for embedding, for example, contextual embedding [22]–[25]. The similarity metric for semantic mapping is mostly calculated as cosine (9), but other techniques [7], [8], [20] are additionally used. Mariani et al. [12] compare the existing tools by Lin et al.[9] and Behrang et al.[7], [8] to their own implementation of semantic matching using both word and sentence level embeddings. Their results indicate the following: sentence level word embeddings outperform word embeddings, since many attribute descriptions include numerous words; the algorithm computing the semantic similarity score has a bigger impact on performance than other factors, e.g., the type of features extracted or language models.

2.2.1 Evaluating Similarity Based Tools

Works using semantic mapping for test transfer and reuse do not have standardized metrics to measure their performance. Zhao et al. have created a framework for UI test reuse evaluation that accounts for both fidelity and utility of proposed solutions [6]. Although this study does not introduce any test transfer per se, certain metrics suggested by the authors can be applied to measure the quality of improved element identification. Most of existing research on test reuse with semantic mapping focuses on fidelity, i.e., how well tests from the original application match the new one. Zhao et al. use seven fidelity

[4]WordNet is a lexical database available at https://wordnet.princeton.edu/

metrics [6], [26]:

1. True Positive (TP), when a match is correct
2. False Positive (FP), when a match is incorrect
3. True Negative (TN), when a match does not exist
4. False Negative (FN), when a match is missed
5. Accuracy = $\frac{TP+TN}{P+N}$
6. Precision = $\frac{TP}{P*}$
7. Recall = $\frac{TP}{P}$

Additionally, they introduce two utility metrics that evaluate how useful a transferred test is: the *effort* metric, which is defined as Levenshtein distance between tests [27] measuring a required number of steps to convert one test to another by performing *insertion, deletion, substitution*. The other metric is *reduction*, which shows manual effort needed for the test transfer compared to creating a new test. The latter can be negative if the test transfer takes more steps. The suggested utility metrics are useful for evaluating an actual test transfer tool, as the primary goal of such a system is to reduce manual efforts involved in GUI testing.

2.3 Machine Learning

According to Murphy [28], machine learning comprises automated methods that uncover underlying patterns in data, which then assist us in decision-making under uncertainty and future data prediction. This study applies machine learning to achieve the primary goal of improved GUI element identification. This section introduces machine learning methods required for further discussion of the similarity-based tool introduced in this work.

Typically, machine learning methods are grouped into supervised and unsupervised [26], [28], [29], where the first requires a labeled data set with each data point x_i having a label y_i, and the latter allows us to find patterns in an unlabeled data set. Common supervised tasks belong to either classification, when a target variable y_i is categorical, or regression, when y_i is real-valued. Unsupervised learning presumes that there are no

known values of y_i. As James et al. [26] note, unsupervised learning is more challenging than supervised due to the absence of widely accepted validation mechanisms. Another class of machine learning methods is semi-supervised, which takes both labeled and unlabeled samples as input [30]. Labels for unlabeled data points are then determined during learning. Semi-supervised learning is useful when there is a small amount of labeled data. In this study, manually annotated data are not available as input for training, which is why three semi-supervised methods are applied for the task of web page element classification.

2.3.1 Self-Organizing Maps

Self-organizing maps (SOMs) are built on a shallow artificial neural network (ANN) architecture (Figure 2) [31]. It consists of an input layer fully connected to a two-dimensional output layer [32]. Neurons in the output layer are interconnected (Figure 2), and changing one neuron has an effect on neighboring neurons as a result their relationship. The advantages of SOMs include a lower chance of overfitting due to this neighborhood relationship of neurons and easy visual comprehension of the output two-dimensional grid [32].

This work uses the semi-supervised implementation SOM by Riese and Keller [32]. The unsupervised SOM algorithm can be broken down into the following steps, where $\{\mathbf{x}(\mathrm{t})\}$ is a sequence of real n-dimensional Euclidean vectors $\mathbf{x}$, the integer t is a step in the sequence [33]:

1. Initialization of a SOM, choosing weights randomly, and setting t to 1.

2. Randomly selecting an observation $\mathbf{x}(\mathrm{t})$.

3. Finding the Best Matching Unit (BMU) $\mathrm{c}(\mathbf{x})$. The selected observation is compared to all weight vectors in the grid, and the closest one is chosen with the Euclidean distance (1) as a metric.

$$d(x,x') = \sqrt{\sum_{i=1}^{n}(x_i - x_i')^2} \tag{1}$$

4. Calculation of the learning rate $\alpha(\mathrm{t})$ and the neighborhood function $\sigma(\mathrm{t})$. The learning rate α is "a positive scalar determining the size of the step" [29], i.e. controlling

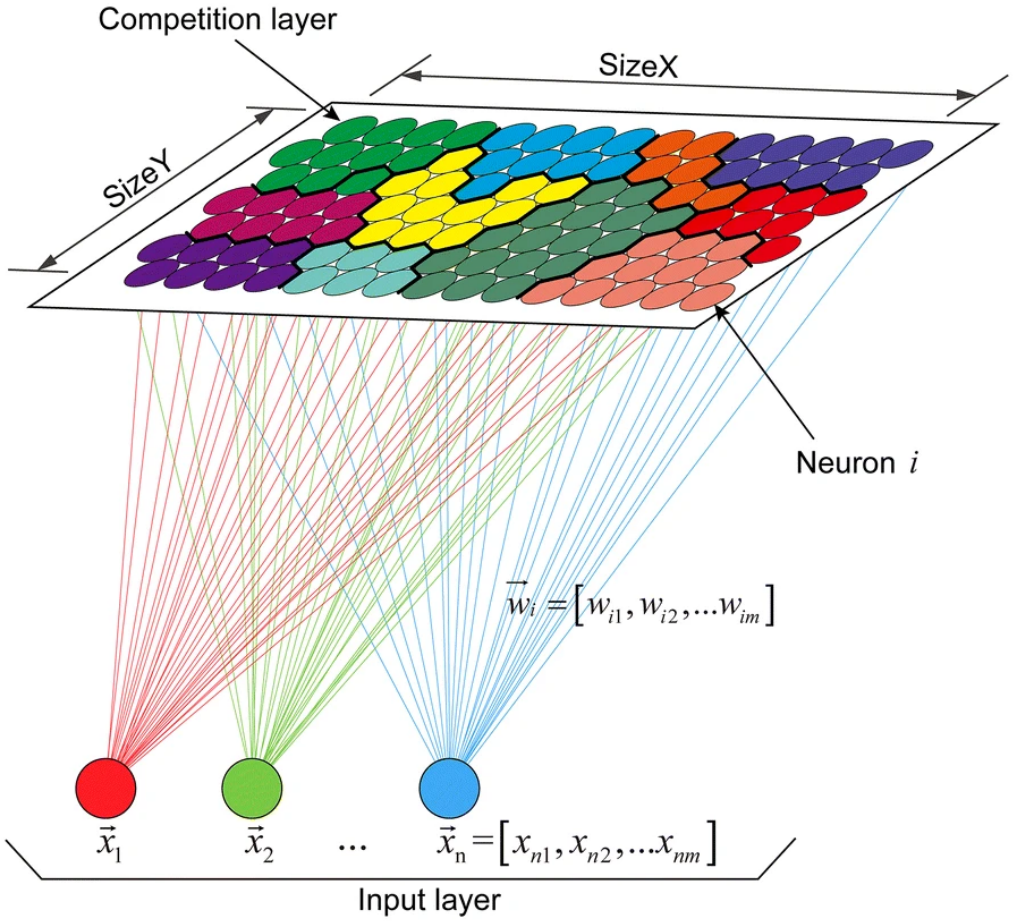


Figure 2: The basic architecture of SOM consisting of two fully connected layers [31].

the rate of weights changing. It decreases from the start value α_0 as t grows. The implementation by Riese et al. [32] also includes the end value α_{end}:

$$\alpha(\mathrm{t}) = \alpha_0 \cdot \left(\frac{\alpha_{end}}{\alpha_0}\right)^{t/t_{max}} \tag{2}$$

The neighborhood function is also decreasing with the number of iterations and has the initial value of σ_0:

$$\sigma(\mathrm{t}) = \sigma_0 \cdot \left(1 - \frac{t}{t_{max}}\right) \tag{3}$$

5. Calculation of the neighborhood distance weight $\mathrm{h_{c,i}(t)}$ between the BMU c and and the neuron i:

$$h_{c,i}(t) = \exp\left(-\frac{d^2}{2 \cdot \sigma(t)^2}\right), \tag{4}$$

where the distance $d(c,i)$ is the Euclidean distance (1) between the BMU c and the neuron i in the two-dimensional grid (Figure 2).

6. The SuSi framework [32] uses the online implementation of SOM by default, but the batch option is also available. Kohonen [33] recommends it for practical applications of SOMs. In this mode, all input data points N are fed into the algorithm and used in each iteration. Each weight $\mathbf{w}_i$ is adapted as follows:

$$\mathbf{w}_i(t+1) = \frac{\sum_{j=1}^{N} h_{c,i}(t) \cdot \mathbf{x}_j}{\sum_{j=1}^{N} h_{c,i}(t)}, \tag{5}$$

where $h_{c,i}(t)$ is defined in Equation 4.

7. Steps 2-6 are repeated until t_{max} is reached. At this point, the model is trained and the BMU for every data point is determined.

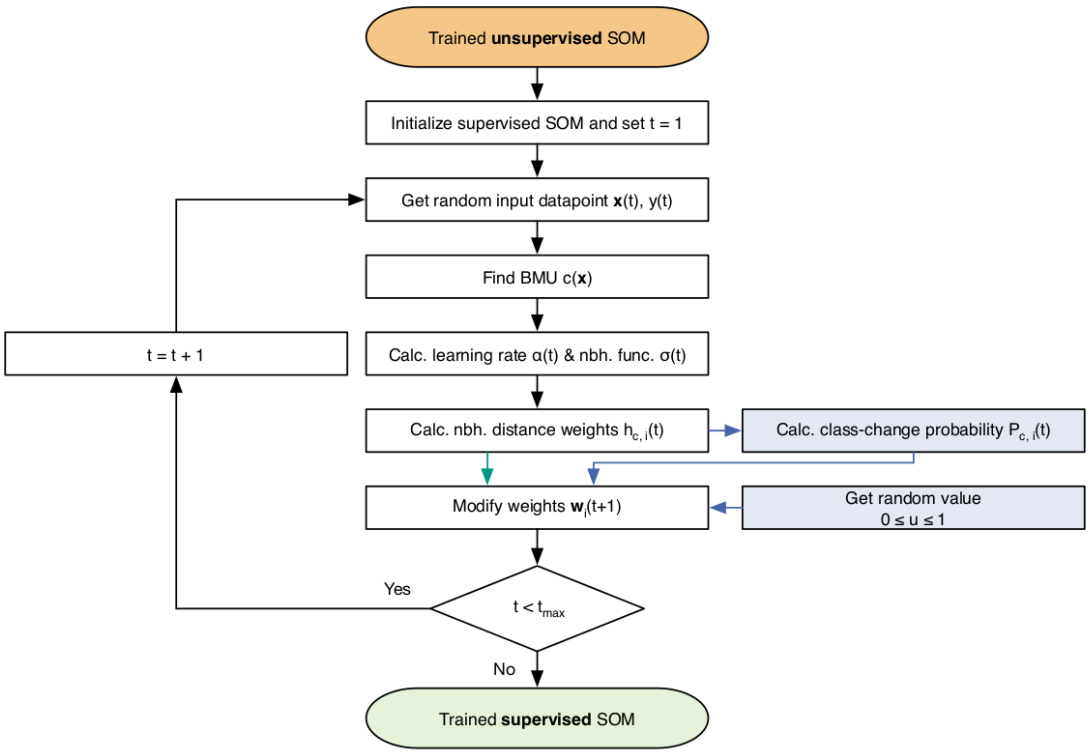


Figure 3: The flowchart of supervised SOM, the blue boxes and arrows are for the Classification algorithm

[32]

This study uses the semi-supervised SOM algorithm by Riese and Keller [32], which combines the unsupervised version introduced above and the supervised Classification SOM algorithm (Figure 3). The unsupervised algorithm is applied to all data points, then

the supervised part is applied to labeled data. The Classification SOM implementation by Riese and Keller [32] introduces the class-change probability (Figure 3). Data sets are often imbalanced in terms of the distribution of classes, i.e., some classes have more observations than others. To mitigate the issue, Riese and Keller propose the option to re-weight the data set. The optional class weight $w_{class(j)}$ for the data set with N data points and the number of data points N_j in class j is defined as follows:

$$w_{class(j)} = \begin{cases} N/(n_{classes} \cdot N_j), & \text{if class weighting,} \\ 1, & \text{otherwise.} \end{cases} \tag{6}$$

The probability $P_{c,i}(t)$ of class change for a node i is as follows:

$$P_{c,i}(t) = w_y(t) \cdot \alpha(t) \cdot h_{c,i}(t), \tag{7}$$

where $\alpha(t)$ is defined in Equation 2, $h_{c,i}(t)$ is defined in Equation 4, and $w_y(t)$ is as in 6. The decision whether to change a class for a specific node depends on a binary decision rule. A random uniformly distributed value $u_i(t)$ is generated per node in each iteration, and the modification of the weights is defined as follows:

$$w_i(t+1) = \begin{cases} y(t), & \text{if } u_i(t) < P_{c,i}(t), \\ w_i(t), & \text{otherwise.} \end{cases} \tag{8}$$

In the case of the semi-supervised SOM, which is a combination of both algorithms discussed above, more weight is given to labeled data points compared to unlabeled ones.

2.3.2 Label Propagation and Label Spreading

Label propagation is based on the idea of graph representation of data points. Such an empirical graph $g = (V, E)$ consists of nodes V, which are data points, and edges E, which are similarities between them. The weight of the edge between the nodes i, j depends on the local Euclidean distance $d_{i,j}$ given in 1 between the nodes i, j so that the closer they are, the larger the weight $W_{i,j}$ [34] is.

Nodes $1, 2, ..., l$ are labeled, and nodes $l+1, ..., n$ are not. Predicted labels are $\hat{Y} = (\hat{Y}_l, \hat{Y}_u)$ for labeled and unlabeled data points. $\hat{Y}_l$ is equal to $Y_l = (y_1, ..., y_l)$. Known labels are binary $(-1, 1)$, unknown are equal to 0. Data sets with more than two classes can be one-hot encoded.

The algorithm is defined as follows [34], [35]:

1. Computing a weight matrix W defined as $W_{i,j} = \exp\left(-\frac{d_{i,j}^2}{\sigma^2}\right)$
2. Computing a diagonal degree matrix D by $D_{ii} \leftarrow \sum_j W_{ij}$
3. Initializing $\hat{Y}^{(0)} \leftarrow (y_1, ..., y_l, 0, 0, ..., 0)$
4. Iterating until converging to $\hat{Y}^{(\infty)}$
   (a) $\hat{Y}^{(t+1)} \leftarrow D^{-1}W\hat{Y}^t$
   (b) $\hat{Y}_l^{(t+1)} \leftarrow Y_l$
5. A data point x_i is assigned a label $\hat{y}_i^{(\infty)}$

Label spreading was introduced by Zhou et al. [36]. Given a data set containing data points $X = \{x_1, ..., x_l, x_{l+1}, ..., x_n\} \subset R^m$ and labels $L = \{1, ..., c\}$, the points $x_i (i \leq l)$ are labeled and the rest $x_u (l+1 \leq u \leq n)$ are unlabeled.

The algorithm is defined as follows [35], [36]:

1. Computing an affinity matrix W defined as $W_{i,j} = \exp\left(-\frac{d_{i,j}^2}{2\sigma^2}\right)$ if $i \neq j$ and $W_{ii} \leftarrow 0$
2. Computing a diagonal degree matrix D by $D_{ii} \leftarrow \sum_j W_{ij}$
3. Computing a normalized graph Laplacian matrix $L \leftarrow D^{-1/2}WD^{-1/2}$
4. Initializing $\hat{Y}^{(0)} \leftarrow (y_1, ..., y_l, 0, 0, ..., 0)$
5. Selecting a parameter $\alpha \in [0, 1)$
6. Iterating $\hat{Y}^{(t+1)} \leftarrow \alpha L\hat{Y}^{(t)} + (1-\alpha)\hat{Y}^{(0)}$ until converging to $\hat{Y}^{(\infty)}$
7. A data point x_i is assigned a label $\hat{y}_i^{(\infty)}$

2.4 Text Representation and Semantic Similarity

Since the basis of semantic similarity originates from linguistics and information retrieval, this section provides a brief introduction to vector semantics and similarity measures. The idea that semantically similar words appear in similar contexts, and the notion that there

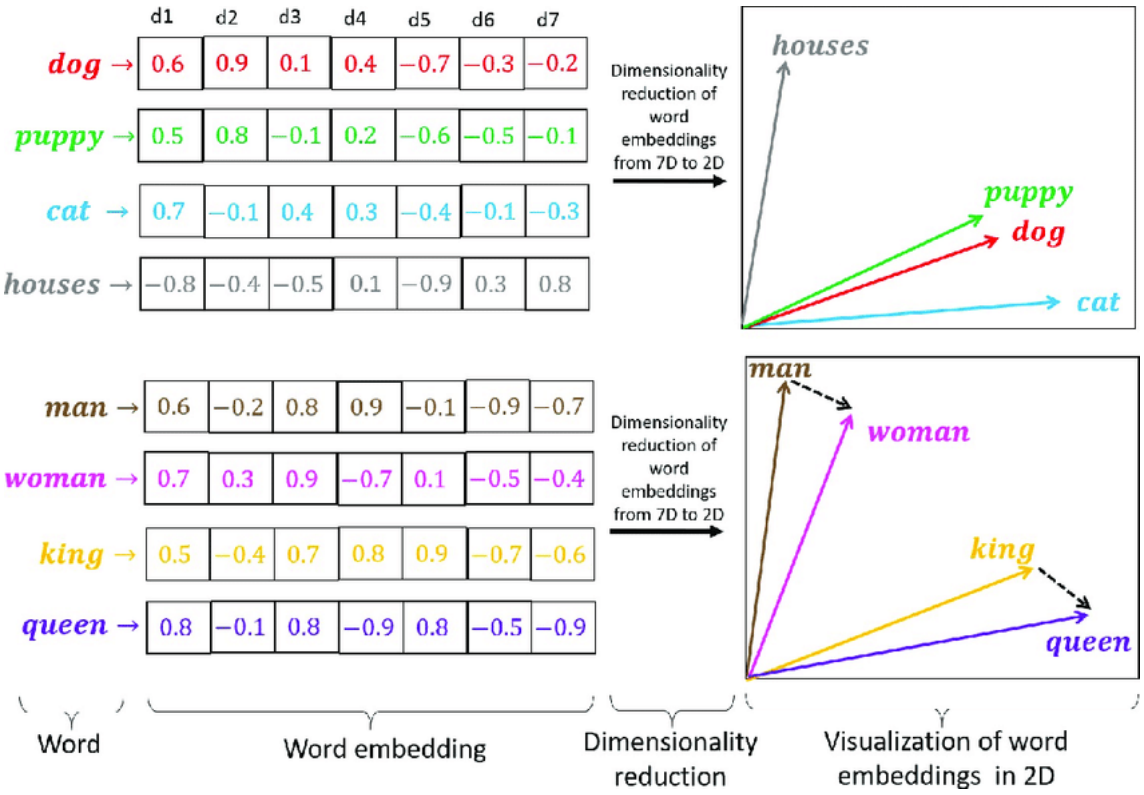


Figure 4: Word embeddings in the vector space
[39]

is a connection between the meaning of a word and the frequency of its occurrence in certain environments is called distributional hypothesis [37], [38].

Word similarity is often conveyed through word embeddings, where a word is represented as a vector in a continuous space. Since it is possible to calculate similarity between vectors using cosine similarity (9), a pair of word vectors v, w can have a numerical value [25], [40]. Such vectors are frequently called word embeddings [40] . To create such embeddings, vectors are represented in the vocabulary R^d, where d is the dimensionality of the vector space [40].

$$sim(v,w) = \frac{\mathbf{v} \cdot \mathbf{w}}{|| \mathbf{v} || || \mathbf{w} ||} \tag{9}$$

Distributional hypothesis has given a basis for the computational framework of distributional semantics, which is also called distributional similarity [40]. According to distributional semantics, words can be represented within a data set D containing word context pairs (w,c). Here, by contexts, Levy [40] means documents, sentences, or words.

The data set D produces V_w, a vocabulary of words that are to be represented, and V_c, a vocabulary of contexts. The data is stored in a matrix, where each cell illustrates an occurrence of a word w in a context c. Figure 4 provides an illustration of word embeddings, where words are placed into the rows of the matrix, and the columns represent contexts (in this case, these are dimensions in a corpus). In Figure 4, context *d1* represents living beings, which is why the word *houses* has a negative value in *d1*, whereas *dog, puppy, cat, man, woman, king, queen*, which are living beings, have positive values [39]. Using dimensionality reduction methods, words can be visualized in the vector space. As Figure 4 shows, the word vectors *dog, puppy, cat* are positioned closer to each other than to the term *houses*.

The similarity between the words in Figure 4 can be measured with cosine similarity between respective word vectors. Smaller similarity values correspond to larger degrees between vectors, thus, representing words that are less semantically similar.

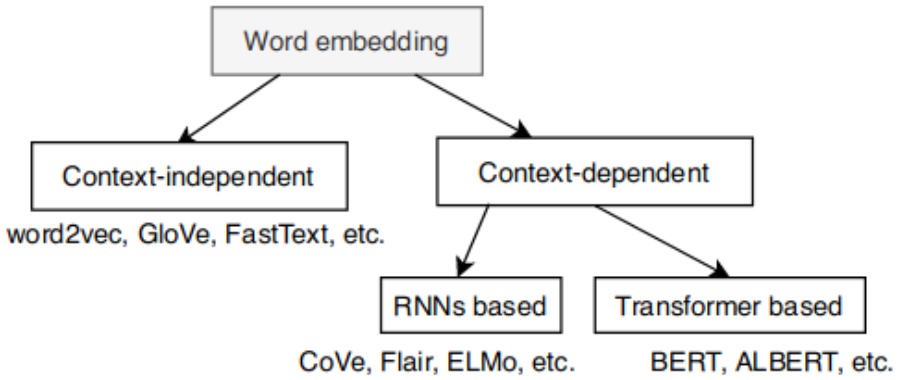


Figure 5: Word embedding types
[24]

Word embeddings as a means of text representation can be viewed as context-independent, such as Word2Vec, and context-dependent, for example, those based on the Transformer architecture (Figure 5) [24], [25]. The first category produces word representations that are not sensitive to context, since they are obtained through shallow neural network models or co-occurrence matrix factorization, whereas the second type is context aware, which allows for the same word to be encoded differently based on the context it is encountered in. Wang et al. provide an example of the polysemy *bank*, which can mean a financial institution or a piece of land along a river, and thus requiring a different representation based on its situational definition [24]. The following sub-chapters focus on Word2Vec, a context-independent word representation method, which has previously been applied in semantic mapping works, and context-dependent language models based on the Trans-

former architecture.

2.4.1 Word2Vec

Word2Vec is a distributional method that has been used for semantic mapping in GUI test transfer research. Church [21] states that the popularity of Word2Vec [41]–[43] in a supporting role in numerous research works is due to its accessibility and simplicity. The idea of Word2Vec lies in representing linguistic patterns as linear relations. There is an example provided by Mikolov [42], where the result of *vec("Madrid") - vec("Spain") + vec("France")* is closest to *vec("Paris")*. The analogy here is that *Madrid* to *Spain* is the same as *Paris* to *France*, where *Paris* is the best candidate word. In order to solve the task in the example, the Word2Vec model finds the embedded vectors x_a, x_b, x_c for the words in the analogy *a:b c:d*, where *d* is unknown, and calculates $y = x_b - x_a + x_c$, which is the continuous space representation of the best candidate [43]. Word2Vec determines the best candidate word for *d* by iterating through all words x' in the vocabulary V, and finding the one that satisfies (10). The similarity measure in (10), *sim*, is cosine, defined in (9) [21].

$$ARGMAX_{x' \in V} sim(x', c + b - a) \tag{10}$$

2.4.2 Context-dependent Embeddings

Similarly to word embeddings, text can be represented with sentence embeddings [25]. Sentence embeddings used in this work are obtained with a pre-trained language model based on the Transformer network architecture (Figure 6)[22], [44]. Liu et al. note that the Transformer excels at modeling sequential data and, as a result, has become popular for various natural language processing tasks [25].

The Transformer has an encoder-decoder structure. The encoder represents an input $(x_1, ..., x_n)$ as a sequence $z = (z_1, ..., z_n)$, and the decoder accepts z and outputs $(y_1, ..., y_n)$ symbols. The encoder consists of 6 layers, each having a multi-head self-attention sublayer and a position-wise fully connected feed-forward sublayer. The sublayers have a residual connection followed by layer normalization. The multi-head attention sublayer has multiple attention heads. A head is a representation of a scale dot-product attention (Figure 7):

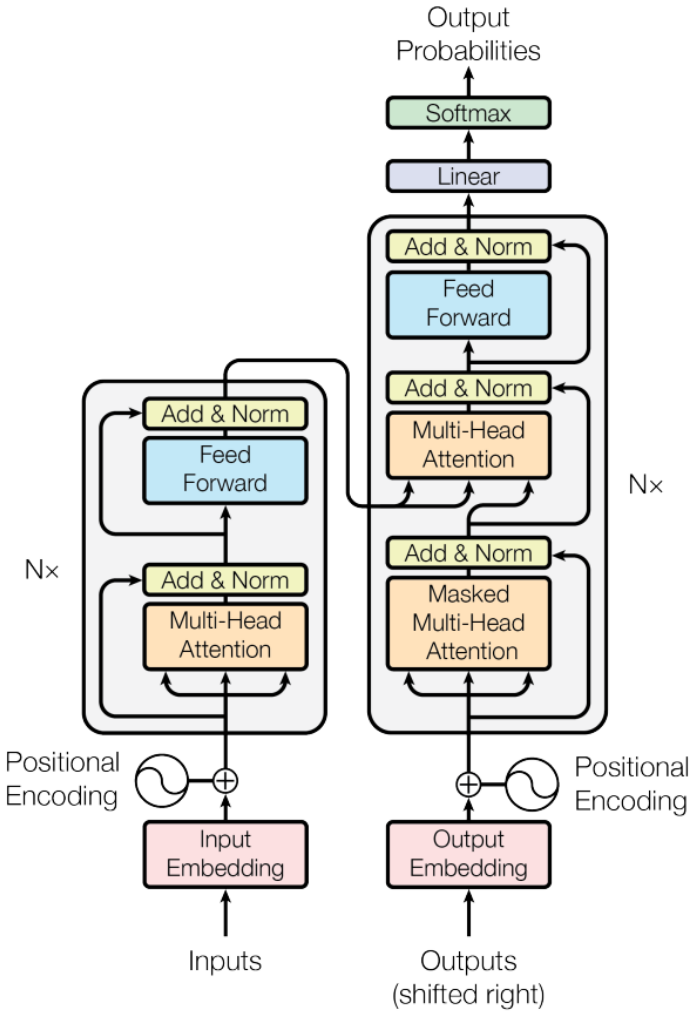


Figure 6: Transformers architecture
[44]

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{11}$$

where Q is a matrix of queries q, K is a matrix of keys k, V is a matrix of values v, and d_k is a dimension of vectors of q and k.

The input, which contains matrices of Q, K, and V, is used to calculate the dot product (11). The scaled dot-product attention mechanism assigns weights to scale each V by multiplying each Q by K and dividing the result by $\sqrt{d_k}$. The multi-head attention then uses linear projections of V, K, and Q h times to calculate their scale dot-product attention and concatenate results of h outputs (Figure 7). The final output is then once again

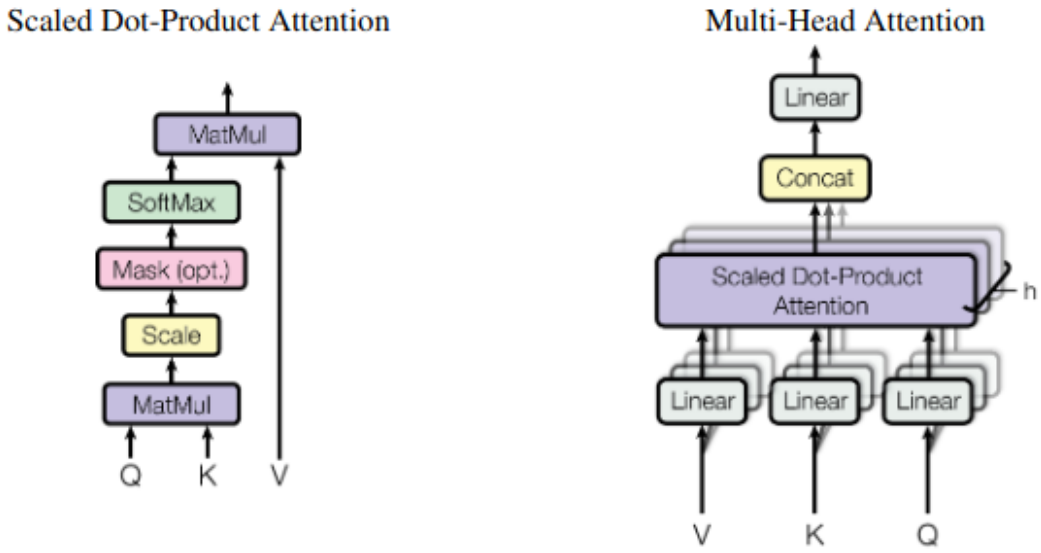


Figure 7: Attention mechanism
[44]

projected. The multi-head attention architecture allows to learn information from various representations of sub-spaces. It has three applications in the Transformer [44].:

1. Queries originate from the previous decoder layer, and memory keys and values are from the encoder output in "encoder-decoder attention" layers.

2. The encoder has self-attention layers, which means that all keys, values, and queries are taken from the previous layer of the encoder.

3. The decoder similarly has self-attention layers.

Another feature of the Transformer is positional encoding, which represents the order of the tokens in a sequence (Figure 6). The encoding has the same dimension as the embedding and is determined by the sinusoidal functions:

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{model}}}) \tag{12}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{model}}}) \tag{13}$$

Vaswani et al. conclude that self-attention layers are faster than recurrent layers for

such tasks as machine translation and word-piece representations. They are also less computationally expensive than convolutional layers, which are even more expensive than recurrent layers. What is more, attention heads allow models to learn syntactic and semantic structures of sequences [44].

2.4.2.1 Sentence BERT Models

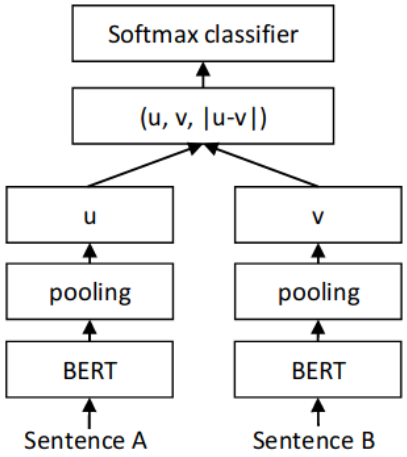


Figure 8: Attention mechanism
[22]

State-of-the-art language models like BERT use the Transformer architecture for solving such tasks as semantic textual similarity (STS) [22], [23], [44]. Reimers and Gurevych argue that using BERT for STS is computationally expensive and sub-optimal for problems requiring semantic similarity search [22]. They propose Sentence-BERT (SBERT) as a more efficient solution for embedding sentences that need to be evaluated in terms of cosine similarity. SBERT uses siamese and triplet networks [45] to create embeddings that can be used for cosine similarity (9) comparisons (Figure 8). The default strategy is to add the *MEAN* pooling operation to the BERT output, which means that the model computes the mean of all output vectors [22]. The authors conclude that SBERT shows improved performance in terms of the quality of sentence embeddings and computational efficiency for STS tasks when compared to other embedding strategies.

Liu et al. note that transferred models are capable of extracting large amounts of information from language, which is why they can be used for various tasks once they are trained on data [25]. Such models are Pre-trained Language Models (PLM), and SBERT is one of them. PLMs can be applied to various tasks by fine-tuning a model to any specific input and receiving token representation as its output.

3 Implementation of Semantic Mapping

The semantic mapping implementation is in Python and consists of three stages. The first stage involves parsing a web application and extracting attributes and properties of HTML elements. Interactive elements, which are determined by their HTML tags, are assigned individual class labels in ascending order starting from *0*, while remaining elements are labeled as *-1*. The goal of the second stage is to group the unassigned elements around the labeled interactive elements using a machine learning classifier. The final stage includes finding semantically similar web elements between different versions of the web application by measuring cosine similarity between sentences that consist of concatenated text features of each class.

3.1 Parsing the Web Document

Retrieving data from the web application[1] is the first stage of semantic mapping. The initial step is parsing an HTML document obtained from a given web link. This task is completed using the Selenium WebDriver API, which provides access to the web page and its elements by controlling a browser window, and the lxml XML toolkit, which allows one to obtain an HTML tree from the web page [46], [47]. The following code snippet contains a Python function for initiating a Selenium webdriver and parsing the web page. It returns an *ElementTree* object containing the entire structure of the web page.

```
1 import lxml.html
2 from selenium import webdriver
3
```

[1] In this study, the web application has only one page, which is why *web application* and *web page* are used interchangeably.

```
def get_html_tree(link):
    driver = webdriver.Chrome(service=
        ChromeService(ChromeDriverManager().install()))
    driver.get(link)
    tree = lxml.html.fromstring(driver.page_source).getroottree()
    return tree
```

To access individual elements, the implementation iterates through the elements of the object and retains their XPath[2] values. Each element's Xpath value is evaluated against a list of interactive elements (forms, links, buttons, drop-down menus, input fields, options, selections etc.). If the element's path ends with an interactive tag, it is assigned an individual class label, otherwise it is given *-1* as its class label. In practice, any element can be assigned its own class label if necessary. The web application used for testing the implementation contains large text paragraphs, which is why their parent headers were also given individual labels, so that the paragraphs could be grouped around them. If the element is hidden, its CSS[3] property is changed to *"display: block"*; and if the driver cannot access the element by its XPath, it is omitted. Once the element is located, it is possible to obtain its attributes and position on the web page. The code snippet below retrieves any available properties and attributes of the HTML elements present in the *ElementTree* object obtained earlier.

```
def get_features(tree, interactive_list):
    labeled_class = 0
    errors = []
    data = []
    display_style = "arguments[0].style.display = 'block';"
    #Iterating through the ElementTree tree
    for e in tree.iter():
        path = tree.getpath(e)
        #Checking if the element ends with an interactive tag
        if [i for i in interactive_list if re.search(i, path)]:
            element_class = labeled_class
            labeled_class += 1
        else:
            element_class = -1
        #Trying to find the element by its XPath
        try:
            element = driver.find_element(By.XPATH, path)
            if not element.is_displayed():
                driver.execute_script(display_style, element)
```

[2]Xpath is an HTML locator used to access HTML elements with the Selenium API.

[3]"Cascading Style Sheets (CSS) is a stylesheet language used to describe the presentation of a document written in HTML or XML" [48].

```
            location = element.location
        except (InvalidSelectorException, NoSuchElementException,
                ValueError, TypeError):
            errors.append(path)
        #Extracting the element's attributes
        values = [path, location['x'], location['y'],
                      element.get_attribute('alt'),
                      element.get_attribute('title'),
                      element.get_attribute('name'),
                      element.get_property('value'),
                      element.get_attribute('id'),
                      element.get_attribute('type'),
                      e.text, element_class]
        data.append(values)
    return data, errors
```

As the result of the first stage, the web page has been parsed and stored in the *ElementTree* object, any available attributes and properties of each HTML element located in the object have been retrieved. Those unassigned elements that do not have any attributes that could be used for semantic mapping, i.e., their *'alt', 'title', 'name', 'value', 'id', 'type', 'text'* attributes are empty, are removed from the data used in the consecutive stages. This step is performed for two versions of the application: the source version and the target version. Thus, two data sets containing information extracted the web pages are obtained.

Compared to the existing tools described in Table 1, this implementation lacks a test suite, which is why the interactive features are identified by the type of their HTML tag. For the same reason, this solution does not construct a graph to retain information about states and transitions between the elements in the web application.

3.2 Grouping the Web Elements

The goal of the second stage is to assign the unlabeled elements to an interactive element class. Three semi-supervised machine learning classifiers have been proposed to achieve the goal: Self-Organizing Maps, Label Spreading, and Label Propagation. Each algorithm has been introduced in Chapter 2. The SOM algorithm implementation used in the experiment is from the SuSi package, the implementations of Label Spreading and Label Propagation are from the scikit-learn package [32], [49]. The performance of the semi-supervised models is also compared a supervised classifier.

Although the semi-supervised methods should be able to assign class labels without a large amount of labeled data, they still need to be optimized in terms of both feature and model parameters. The basic assumption of this study is that once a model has been optimized for the web application, it can be successfully applied to another version without fine-tuning unless the application undergoes significant changes. In order to choose appropriate parameters, the scikit-learn package provides methods that run all possible combinations of any given parameters and output their performance metrics. In this particular case, accuracy introduced in Chapter 2 is the metric that has been used to evaluate every model's performance. The data obtained in the first step include the HTML elements with their Xpath values and attributes. In order to evaluate the classification models, the data have been manually annotated to assign each element to its true class based on its position on the web page and its relation to the neighboring elements.

Although there is a number of features extracted in the first step, only the XPath values and location coordinates are used for the classification task. The motivation behind using only these features is that certain words are often repeated in unrelated HTML elements. This results in them being erroneously placed in the same class, which lowers the overall accuracy of semantic mapping. An Xpath value example is provided below:

```
1    "/html/body/div/div/main/div[1]/section/div[1]/div[1]/div[2]/
    label"
```

Since it is a string, it can be represented as a text input. Language models introduced in Chapter 2 are not necessarily pre-trained on HTML tags, which on their own do not have the same semantic and syntactic meaning as natural languages. Nevertheless, they can be encoded as a term-document matrix. Each HTML tag becomes a term and each HTML element is a document. It is possible to use n-grams of different lengths as terms. N-grams are sequences of length *N* comprised of tokens [50]. Here, since tokens are individual tags, a trigram from the XPath example above is *"div[1], section, div[1]"* and a bigram is *"div[1], section"* or *"section, div[1]"*. N-grams can be used as raw or binary counts; however, it is possible to give more weight to more representative terms that could potentially improve classification accuracy. The scikit-learn package provides *CountVectorizer* and *TfidfVectorizer* feature extractors [51]. The first returns an ordinary term-document matrix, and the latter produces a matrix of TF-IDF features (TF-IDF is the product of term-frequency and inverse document-frequency). It is calculated as follows [50], [51]:

$$tf_{(t,d)} = count(t,d) \tag{14}$$

$$idf_{(t,d)} = \log \frac{1+n}{1+df(t)} + 1 \tag{15}$$

The location attributes are x and y positions on the web page, their values can be within hundreds. They need to be scaled to be within the interval $[0,1]$, since the semi-supervised models in this study use the Euclidean distance measure (1). If they are not scaled to be within the same interval as the text features, they will have a disproportionately big impact on learning. Both feature extraction and scaling constitute preprocessing. The Python code snippet below shows a scikit-learn column transformer preprocessor, which produces a dense matrix containing both TF-IDF features and re-scaled location attributes.

```
preprocessor = make_column_transformer(
    (TfidfVectorizer(token_pattern = r'(?u)\b\w+(?:\[.?\])?'),
            'xpath'),
    (MinMaxScaler(), ['x', 'y']),
    sparse_threshold=0
)
```

The feature extractors have certain parameters that can be optimized:

1. *max_df, a float or an integer*: if a term appears in the number of documents above the given threshold, it is ignored.
2. *ngram_range, a tuple* represents the range of n-grams; for example, (1,2) would result in unigrams and bigrams.
3. *binary, a boolean* allows *CountVectorizer* to return binary values for term frequencies.
4. *sublinear_tf, a boolean* transforms *tf* to $1 + log(tf)$ in *TfidfVectorizer*; it can be useful in cases where a term appears multiple times in a document.
5. *norm, 'l1' or 'l2'* specifies output row unit norm: *'l1'* for the sum of absolute vector values equal to 1 and *'l2'* for the sum of squares of vector values equal to 1.

The implementations of Label Propagation and Label Spreading used in this study allow parameter optimization for both algorithms. The algorithms represent relationships between data points in a graph. The main difference between the two that is important

here is that Label Spreading performs soft clamping, which means that by changing the parameter α, the algorithm controls how much data points learn from their neighbors compared to their original label. Label Propagation uses hard clamping, which means that original data points do not change their labels [52]. Both algorithms implemented in the scikit-learn package use a Radial Basis Function (RBF) kernel to determine edge weights between points. The parameter σ is replaced by γ in the scikit-learn library[34], [35], [52]:

$$exp(-\gamma|x_i - x_j|^2), \gamma > 0 \tag{16}$$

By changing γ, we can regulate the width of the RBF kernel.

There are numerous parameters that could be optimized for SOMs. Since it is a neural net, starting and ending learning rates are of great importance. The authors of the SuSi package suggest a range of values that can be tried. The size of a learning grid, which is the number of rows and columns, is challenging to choose. Riese et al. suggest starting with a grid of 5 by 5. The larger the grid is, the longer it takes to fit a model, hence without having an educated guess about this parameter, it is hard to start with an appropriate value. Opting for a large size grid can lead to overfitting, and the authors consider grids of size $100 * 100$ to be large. There is also a possibility to select various distance metrics, the default being Euclidean; however, the model takes a considerable amount of time to converge with a non-default choice.

The Python code below shows how the parameters for feature extraction are tested. The scikit-learn package provides a Pipeline module, which allows us to include both pre-processors and classifiers as steps in learning. *GridSearchCV* fits the Pipeline model to the data and chooses the best estimator using cross validation; it calculates accuracy with the true labels obtained from the manual annotation and returns the metric for each combination of the parameters discussed earlier. *GridSearchCV* cannot be applied to the data used in this study in a straightforward fashion because each class contains only one data point, which prevents splitting the data into portions for cross validation. The semi-supervised models take both unlabeled and labeled data for fitting, which is why *GridSearchCV* cannot be used to obtain the best estimator for Label Spreading, Label Propagation, or SOM. Nevertheless, it can be used with supervised methods to explore what feature parameters perform the best and determine how they compare to the semi-supervised algorithms. This implementation uses its own Python solution to determine how the parameters affect

the performance of SOM, Label Spreading, and Label Propagation (see Appendix B).

```
import pandas as pd
from sklearn.model_selection import  GridSearchCV
from sklearn.pipeline import Pipeline

def opt_classifier(parameters, preprocessor, classifier, data,
    pathCV):
    #Determing the indices of training and test data sets
    condition = data['class'] != -1
    train_idx = data.index[condition].values
    test_idx = data.index[~condition].values
    #Defining the steps of the Pipeline
    steps = [("preprocessor", preprocessor),
        ("classifier", classifier)]
    model = Pipeline(steps=steps)
    #Defining the GridSearchCV
    #Instead of cross validation, the GridSearchCV fits
    #the model to the training data and evaluates it
    #on the test data
    search = GridSearchCV(model, param_grid=parameters,
            cv=[(train_idx, test_idx)])
    search.fit(data, data['true_label'])
    #Storing the GridSearchCV results
    params = pd.DataFrame(search.cv_results_["params"])
    accuracy = pd.DataFrame(search.cv_results_["mean_test_score"],
            columns = ["Accuracy"])
    tuning_data = pd.concat([params, accuracy], axis=1)
    tuning_data.to_csv(pathCV)
    return tuning_data
```

The main difference of this study's approach to extracting relevant textual cues is the use of machine learning classifiers. The tools described in Chapter 2 mostly use an arbitrary distance in the hierarchy of UI elements or their position to determine whether another element is related a widget. The technique by Rau et al. [4], [5] uses a proprietary algorithm; the authors do not discuss its accuracy. This work tests three classifiers (Label Spreading, Label Propagation, and SOM) that take the tokenized XPath values and location of the elements as input and determine what classes the elements belong to. Thus, the selection of a method for extracting cues is based on the performance of the chosen parameters and models.

3.3 Semantic Mapping

The final stage of the semantic mapping implementation is finding equivalent class labels between the data sets containing the HTML elements extracted from the source and target versions of the application during the previous stages. Once all the data points in both data sets have been assigned labels in the second stage, their text data are concatenated based on their class into sentences, i.e., each class' *'text', 'value', 'id', 'input', 'title', 'name', 'type'* contents are merged. Before being concatenated, long texts are limited to 50 characters. The reason behind this is SBERT truncating long sequences. Next, each sentence from the source and target data sets is embedded using a pre-trained SBERT model [22]. The embeddings and their labels are then stored as two lists and compared using cosine similarity. It is calculated for each pair of the sentences and the highest scored pairs are considered to be semantically similar, for example, a group with label *0* in the source application is closest to the one with label *15* in the target application. Knowing which groups are the most similar, we can find corresponding interactive tags within them for GUI element identification.

The code below shows how semantic similarity between the two versions of the web application is calculated. It is adapted from the official SBERT documentation on semantic similarity [53]. It takes two Pandas DataFrames[4] containing concatenated texts and labels and a pre-trained model as input and returns two Pandas DataFrames: one containing the pairs of the most similar classes and a cosine similarity matrix for each sentence pair.

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer('all-distilroberta-v1')

def measure_similarity(model, target_app_data, source_app_data):
    # Adapted from https://www.sbert.net/docs/usage/semantic_textual_similarity.html
    #Extracting sentences and labels
    sentences1 = target_app_data.iloc[:, 1].to_list()
    sentences2 = source_app_data.iloc[:, 1].to_list()
    labels1 = target_app_data.iloc[:, 0].to_list()
    labels2 = source_app_data.iloc[:, 0].to_list()
    #Embedding the sentences
    embeddings1 = model.encode(sentences1, convert_to_tensor=True)
```

[4]Pandas is a Python library for data analysis [54]. A Pandas DataFrame is a two-dimensional data structure.

```
    embeddings2 = model.encode(sentences2, convert_to_tensor=True)
    #Obtaining cosine similarity scores
    cosine_scores = util.pytorch_cos_sim(embeddings1, embeddings2)
    data = []
    #Finding the pairs with the highest scores
    for i in range(len(sentences1)):
        for j in range(len(sentences2)):
            values = [labels1[i], labels2[j], cosine_scores[i]
        [j].item(), sentences1[i], sentences2[j]]
            data.append(values)
    df_cos = pd.DataFrame(data, columns=['label2', 'label1',
        'cosine', 'text2', 'text1'])
    idx = df_cos.groupby(['label2'])['cosine'].transform(max) ==
        df_cos['cosine']
    df_cos = df_cos[idx]
    cos_matrix = pd.DataFrame(data=cosine_scores.numpy(),
        index=labels1, columns=labels2)
    return df_cos, cos_matrix
```

This approach to determining similarity is similar to the one used in the implementation by Mariani et al. [12]. Instead of using word embeddings, both solutions use sentence embeddings. This work, however, does not average values, adapting a more straightforward way of using cosine similarity as is. Unlike the other tools described in Chapter 2, the type of an event associated with an element is not considered in evaluating similarity, because different events might perform the same function (however, the attribute *type* is used as a text feature).

In conclusion, this semantic mapping implementation differs from the existing tools discussed in Chapter 2 in the following ways:

1. There is no available test suite for the chosen web application, which is why interactive elements are selected manually. For the same reason, the tool does not extract any information about the elements' interactions.

2. The implementation uses extracted XPath values and location attributes as input for machine learning classifiers to assign class labels.

3. Semantic similarity is calculated as cosine similarity between sentence embeddings, which consist of grouped labels. State-of-the-art pre-trained language models are used for embedding text.

3.4 Results

This work uses a web application for generating privacy policies, which is available online[5] [55]. The source code for several versions can be found on GitHub[6] [56]. The chosen application has gone through several revisions, which is the reason it is suitable for the experiment. Its source code is also freely available, which permits for further analysis of how the proposed solution works with data obtained from the web application. There has been a number of changes in the UIs of the application, certain parts have been re-positioned, a small number of new elements have been added. Nevertheless, the core functionality of the application has remained the same, and the text content has remained similar.

Using the processing methods described earlier in this Chapter, both versions of the web application have been parsed. Only those HTML elements that have original individual labels and those that are unlabeled but contain textual attributes have been retained. The resulting data set for the earlier (source) version contains 190 HTML elements, and the data set for the newer (target) version has 153 HTML elements. Both are stored as Pandas DataFrames for convenient data manipulation.

3.4.1 Parameter Tuning

The feature extraction methods used in this study have numerous parameters that can be optimized. *GridSearchCV* is designed for tuning parameters with cross validation; however, since there is only one data point per class in both data sets, all labeled points are used for training and remaining points are used for testing. To explore how accuracy is affected by various combinations of parameters and establish the baseline performance of a supervised classifier, *GridSearchCV* has been used with a Logistic Regression model from the scikit-learn package. Appendix B contains the Python code for tuning the semi-supervised models. The results of the Logistic Regression classifier are displayed in Appendix A showing that longer range *ngram* and higher *max_df* values produce better outcomes. However, the supervised model has produced modest outcomes, especially when tested on the target data set. The *CountVectorizer* has led to higher accuracy for both data sets (79% for the source data set and 67% for the target data set).

Figure 9 shows how the *max_df* and *ngram* parameters affect the performance of

[5]https://app-privacy-policy-generator.nisrulz.com/
[6]https://github.com/nisrulz/app-privacy-policy-generator

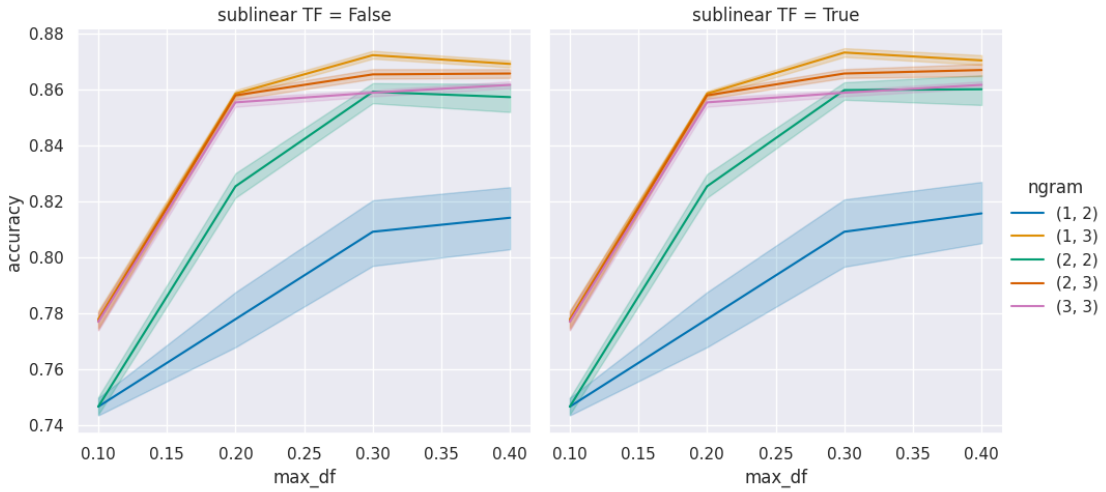


Figure 9: TF-IDF vectorizer parameters for the source application with Label Spreading

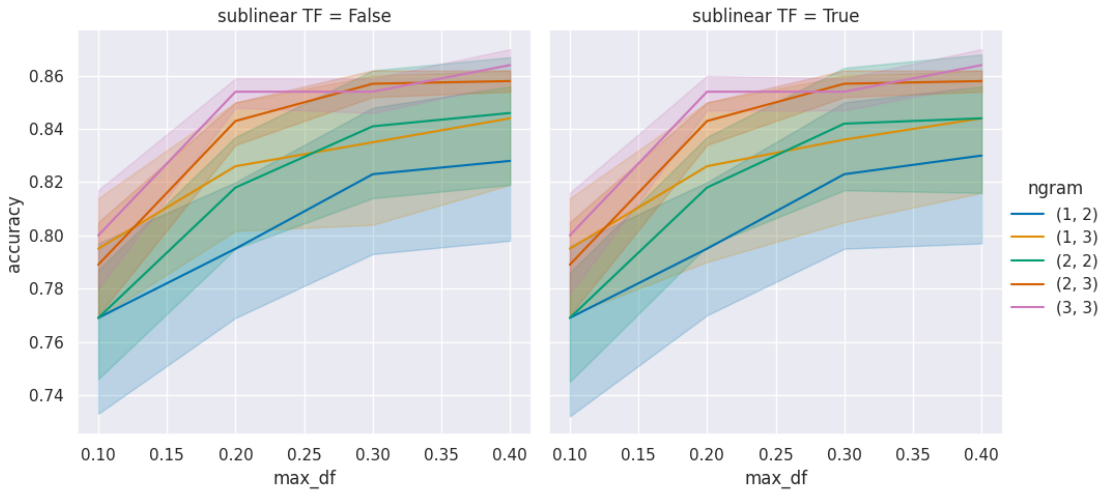


Figure 10: TF-IDF vectorizer parameters for the source application with Label Propagation

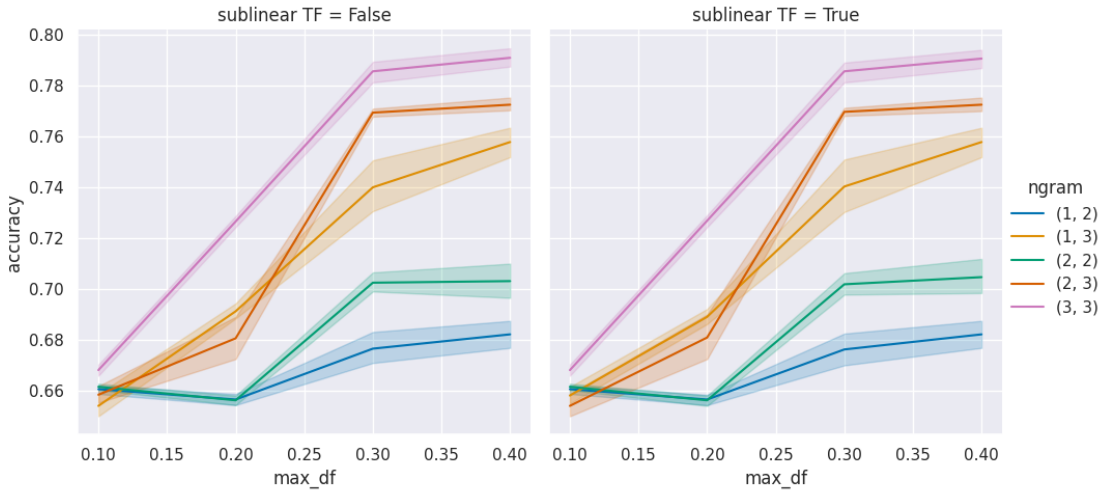


Figure 11: TF-IDF vectorizer parameters for the target application with Label Spreading

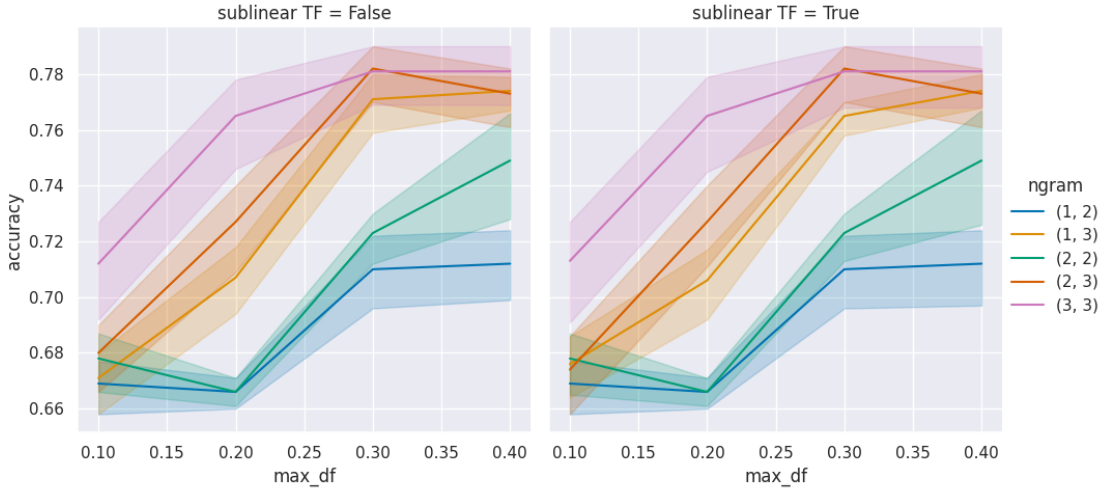


Figure 12: TF-IDF vectorizer parameters for the target application with Label Propagation

the Label Spreading classifier used with the *TfidfVectorizer* to assign labels in the source application data set. The accuracy varies a lot depending on the values of parameters. Generally, the longer *ngram* sequences and the higher values of *max_df* improve the performance. Figure 9 suggests that the bigrams and trigrams are more suitable for the classification. It appears that the application of *sublinear tf* does not make a significant difference. Figure 10 describes the same parameters but used with the Label Propagation model, which appears to produce similar results; however, the spread of the accuracy range is wider with the Label Propagation classifier.

The results for the target application label prediction with the Label Spreading classifier, depicted in Figure 11, show a lower overall accuracy of prediction. Here, similarly to the outcomes of the previous classification, the *max_df* and *ngram* parameters seem

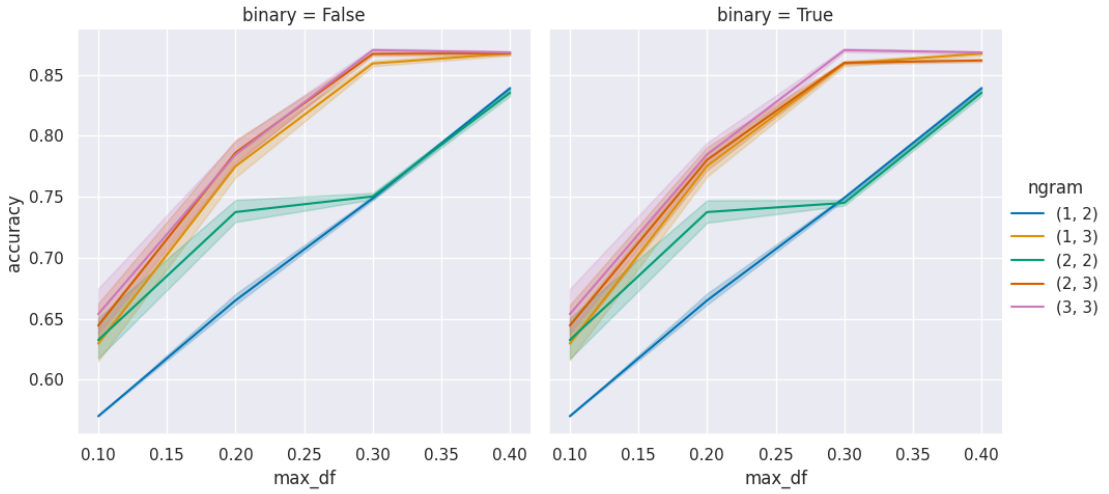


Figure 13: Count vectorizer parameters for the source application with Label Spreading

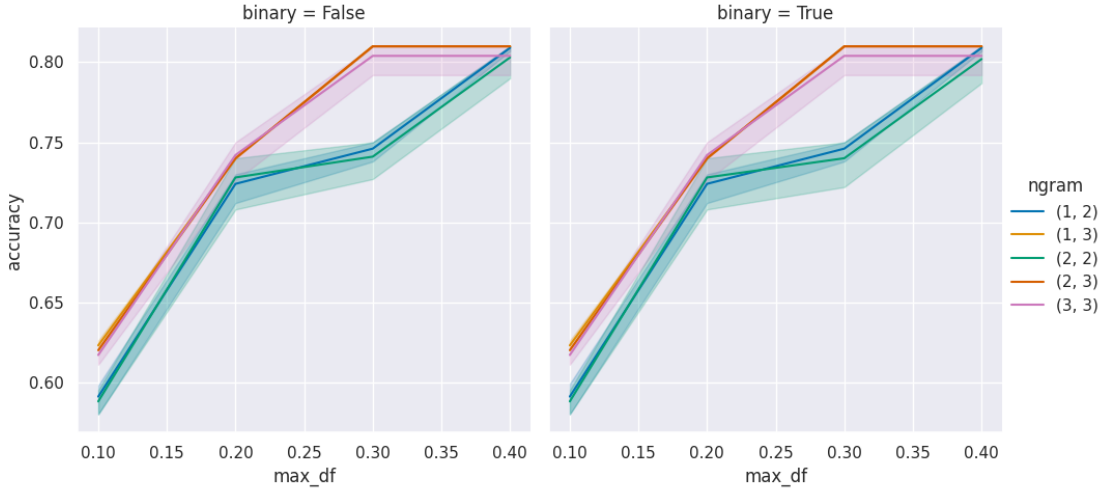


Figure 14: Count vectorizer parameters for the source application with Label Propagation

to affect the classification performance, whereas the effect of *sublinear tf* appears to be limited. Generally, *sublinear tf* is useful in cases when a term reoccurs in a document numerous times, while in this particular setting, terms are unlikely to reappear in the same document. Figure 12 shows that the same input features produce slightly lower accuracy scores when used with the Label Propagation classifier.

Figures 13 and 14 display the effect of *max_df* and *ngram* parameters on the performance of the Label Spreading and Label Propagation classifiers used with the *CountVectorizer* to assign labels in the source application data set. The general results do not differ from the outcomes with the *TfidfVectorizer*: the trigrams and bigrams perform the best, and limiting the most frequent terms appearing across numerous documents improves the performance of the model.

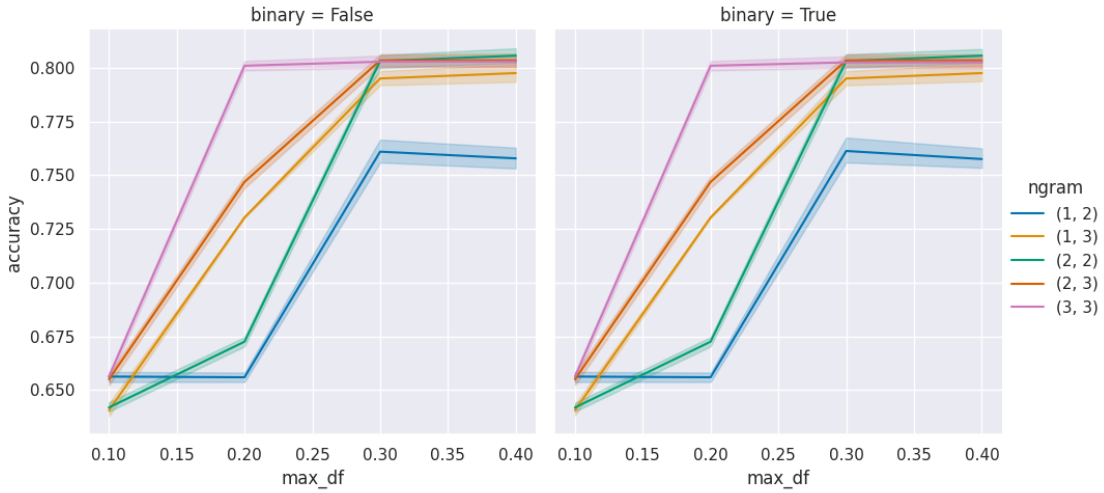


Figure 15: Count vectorizer parameters for the target application with Label Spreading

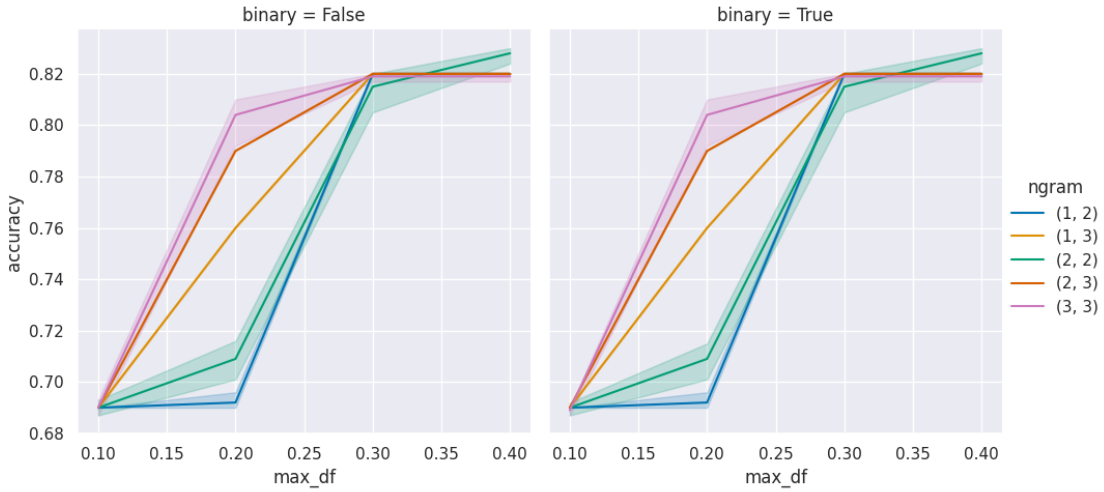
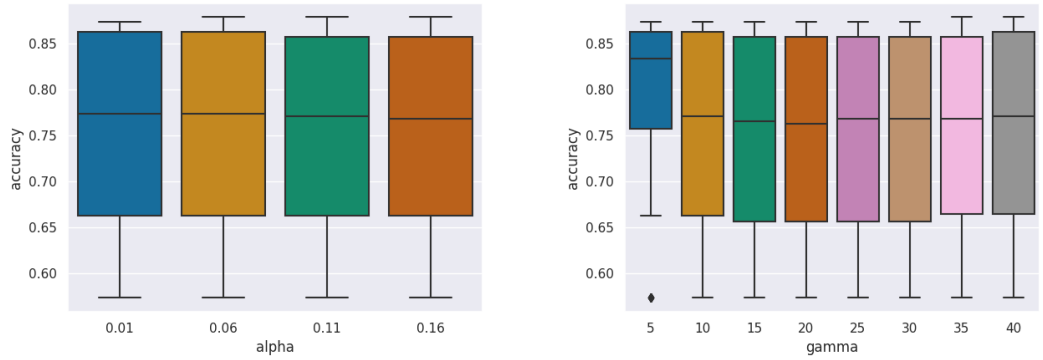


Figure 16: Count vectorizer parameters for the target application with Label Propagation



(a) α (b) γ

Figure 17: Label Spreading accuracy scores (the source data with the *CountVectorizer* input)

accuracy	**count**	**mean**	**std**	**min**	**25%**	**50%**	**75%**	**max**
α								
0.01	320	0.8303	0.04276721958	0.74	0.79	0.855	0.86	0.88
0.06	320	0.8279	0.04545964313	0.74	0.79	0.86	0.86	0.88
0.11	320	0.8254	0.0466067954	0.74	0.7775	0.85	0.86	0.88
0.16	320	0.8234	0.04713811994	0.74	0.77	0.85	0.86	0.88

Table 2: Label Spreading accuracy scores for different α values (the source data with the *TfidfVectorizer* input)

Figures 15 and 16 shows how the text feature parameters impact the performance of the Label Spreading and Label Propagation classifiers applied to the target application data set. The trends observed in previous figures remain, the trigrams and bigrams provide better accuracy scores. Overall, the features produced by the *CountVectorizer* have lead to better results in the target data set.

Figure 17, Table 2, and Table 3 show the distribution of the accuracy scores for the Label Spreading classifier with respect to the parameters α and γ (for the source data set). The smallest values of γ result in the highest median accuracy, whereas the best performance is achieved with the highest values (for the *CountVectorizer* input). The default value of 20 results in good prediction outcomes for both types of input features. Changing α does not lead to better predictions. The default α is equal to 0.2, which corresponds to the algorithm retaining 80% of the original label distribution, and the tested values are below 0.2.

Figure 18 shows the distribution of the accuracy scores for the Label Propagation classifier trained on the *TfidfVectorizer* output in respect of the parameter γ. Interestingly,

accuracy	count	mean	std	min	25%	50%	75%	max
γ								
5	160	0.8311	0.04472118376	0.74	0.8	0.86	0.86	0.87
10	160	0.8303	0.04458645584	0.74	0.8	0.85	0.86	0.87
15	160	0.8298	0.04616060969	0.74	0.7775	0.86	0.86	0.88
20	160	0.829	0.04563730234	0.74	0.79	0.855	0.87	0.88
25	160	0.8276	0.04513044511	0.74	0.79	0.85	0.8625	0.88
30	160	0.8255	0.04541557756	0.74	0.79	0.85	0.86	0.88
35	160	0.8222	0.04548757519	0.74	0.77	0.85	0.86	0.88
40	160	0.8185	0.04666486969	0.74	0.77	0.84	0.86	0.88

Table 3: Label Spreading accuracy scores for different γ values (the source data with the *TfidfVectorizer* input)

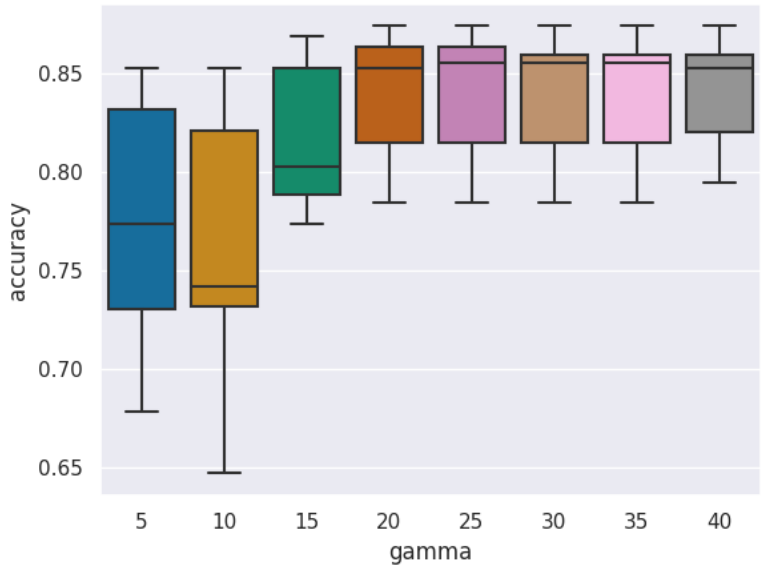
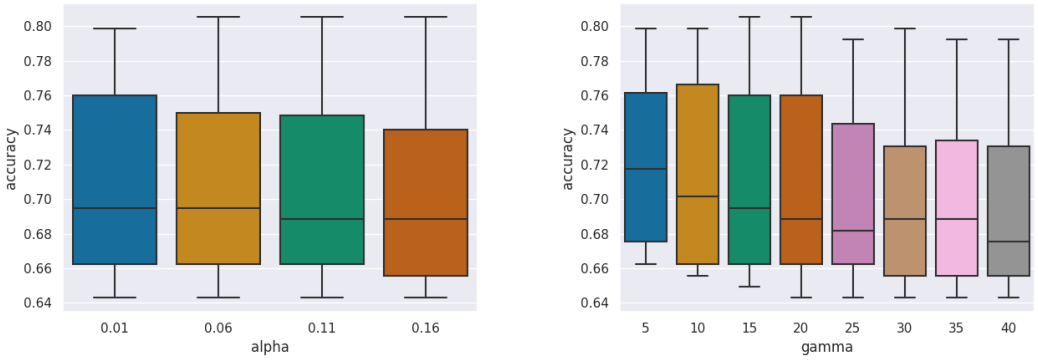


Figure 18: Label Propagation accuracy scores for different γ values (the source data with the *TfidfVectorizer* input)

accuracy	count	mean	std	min	25%	50%	75%	max
γ								
5	40	0.7045	0.0845	0.55	0.65	0.725	0.7625	0.81
10	40	0.7325	0.0809	0.57	0.68	0.745	0.81	0.81
15	40	0.737	0.079	0.6	0.7025	0.75	0.81	0.81
20	40	0.7375	0.0783	0.6	0.705	0.75	0.81	0.81
25	40	0.7365	0.0801	0.59	0.705	0.75	0.81	0.81
30	40	0.736	0.0808	0.59	0.7025	0.75	0.81	0.81
35	40	0.736	0.0808	0.59	0.7025	0.75	0.81	0.81
40	40	0.736	0.0808	0.59	0.7025	0.75	0.81	0.81

Table 4: Label Propagation accuracy scores for different γ values (the source data with the *CountVectorizer* input)



(a) α (b) γ

Figure 19: Label Spreading accuracy scores (the target data with the *TfidfVectorizer*)

unlike in the case of the Label Spreading model trained on the *CountVectorizer* output described in the previous paragraph, the highest median accuracy scores are not achieved with the smallest values of γ. The median accuracy improves with γ values of 20 and higher. Table 4 shows the equivalent data for the *CountVectorizer* input. The overall accuracy is lower, but the smallest value of γ also results in the worst median score. Small values of γ correspond to wider RBF kernels, which means that two data points are estimated to be similar even if they are positioned far from each other, while the opposite is true for large values of γ. Regardless of the type of input features, the default value of γ (20) delivers optimal performance.

Figure 19, Table 5, and Table 6 shows the distribution of the accuracy scores in the target application in respect to the parameters α and γ for the Label Spreading model. It appears that using the default value for γ is optimal for both data sets. However, high values of γ appear to hinder performance of the Label Spreading classifier on the target data set. Similarly to the results obtained from the source data set, changing α does not

accuracy	count	mean	std	min	25%	50%	75%	max
α								
0.01	320	0.7394375	0.07137473957	0.63	0.66	0.755	0.81	0.82
0.06	320	0.74015625	0.06625901932	0.63	0.66	0.75	0.8	0.82
0.11	320	0.7398125	0.0649617382	0.64	0.66	0.75	0.8	0.82
0.16	320	0.73834375	0.06450248926	0.64	0.66	0.75	0.8	0.82

Table 5: Label Spreading accuracy scores for different α values (the target data with the *CountVectorizer* input)

accuracy	count	mean	std	min	25%	50%	75%	max
γ								
5	160	0.750375	0.07090508759	0.64	0.66	0.79	0.82	0.82
10	160	0.741625	0.06881647729	0.63	0.66	0.77	0.81	0.82
15	160	0.74	0.06822087694	0.63	0.66	0.755	0.8	0.82
20	160	0.73825	0.06636614656	0.63	0.66	0.75	0.8	0.82
25	160	0.738	0.06569224991	0.63	0.66	0.75	0.8	0.82
30	160	0.7375	0.06628507141	0.63	0.66	0.75	0.8	0.81
35	160	0.7355	0.06395556634	0.63	0.66	0.75	0.8	0.8
40	160	0.73425	0.06363763374	0.63	0.66	0.75	0.79	0.8

Table 6: Label Spreading accuracy scores for different γ values (the target data with the *CountVectorizer* input)

improve accuracy. Figure 20 and Table 7 show that opting for the default γ value leads to optimal results with the Label Propagation algorithm. Similarly to the results of the Label Propagation classifier applied to the source application, small values reduce accuracy.

accuracy	count	mean	std	min	25%	50%	75%	max
γ								
5	40	0.68175	0.04689800717	0.62	0.6475	0.66	0.74	0.76
10	40	0.69825	0.04012081115	0.64	0.65	0.695	0.74	0.75
15	40	0.72725	0.04574104929	0.66	0.68	0.725	0.77	0.79
20	40	0.729	0.04556201747	0.67	0.68	0.725	0.78	0.79
25	40	0.731	0.04589843861	0.67	0.68	0.725	0.78	0.79
30	40	0.73425	0.0430198225	0.68	0.69	0.73	0.78	0.79
35	40	0.73525	0.04332273046	0.67	0.69	0.73	0.78	0.79
40	40	0.7365	0.04329682881	0.67	0.69	0.73	0.78	0.79

Table 7: Label Propagation accuracy scores for different γ values (the target data with the *TfidfVectorizer* input)

Tables 8 and 9 show the maximum achieved accuracy scores for both data sets. The results of the SOM algorithm are not promising. As explained earlier, its training is extremely time-consuming, and as a result, parameter optimization for SOM is practically

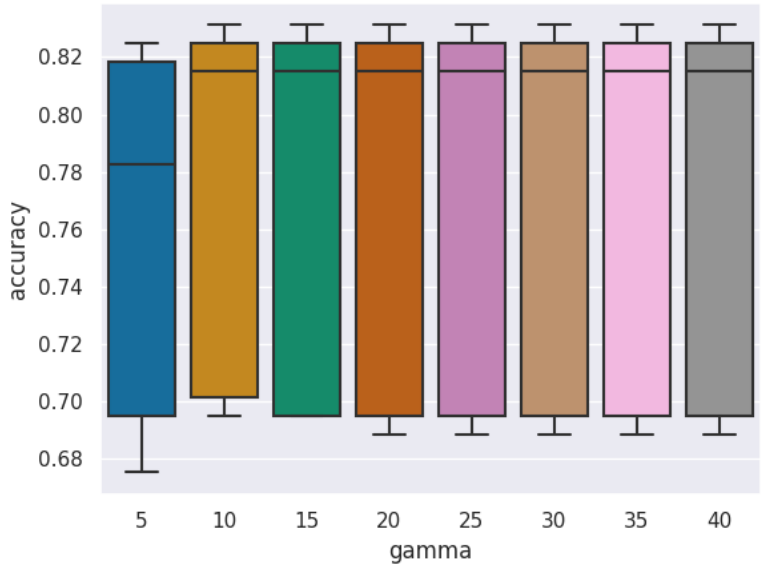


Figure 20: Label Propagation accuracy scores for different γ values (the target data with the *CountVectorizer* input)

Vectorizer	**Label Spreading**	**Label Propagation**	**SOM**	**Logistic Regression**
CountVectorizer	87%	81%	53%	79%
TfidfVectorizer	88%	87%	66%	78%

Table 8: Classification accuracy results for the source application

impossible without additional computational resources. The highest obtained accuracy with SOM has not exceeded 66% with a large grid of 100 by 100 rows.

The results for the source application are presented in Table 8. The best result for the Label Propagation algorithm is 87% with the TF-IDF features and 81% with the term frequencies; the highest accuracy for the Label Spreading algorithm is 88% with the TF-IDF features and 87% with the term frequencies. Compared to best scores produced by the Logistic Regression classifier (79%) and SOM (66%), the Label Spreading and Propagation algorithms perform better for both types of input. It appears that the Label Spreading algorithm makes better predictions with the term frequency features than the Label Propagation model.

The outcomes for the target application are presented in Tables 9. The best result for the Label Propagation algorithm is 79% with the TF-IDF features and 83% with the term frequencies; the highest accuracy for the Label Spreading algorithm is 80% with the TF-IDF features and 82% with the term frequencies. The overall performance of the models is weaker for the target application. While Logistic Regression produces good

Vectorizer	Label Spreading	Label Propagation	SOM	Logistic Regression
CountVectorizer	82%	83%	60%	67%
TfidfVectorizer	80%	79%	54%	62%

Table 9: Classification accuracy results for the target application

results with the source application data, it barely outperforms SOM when used with the target data set. The Label Spreading and Propagation algorithms display higher accuracy overall. It appears that features extracted with *CountVectorizer* produce higher accuracy, while the opposite is true for the source data set.

The parameter tuning performed in this study shows that the method for feature extraction and its parameters affect accuracy. The general trend is that longer *ngram* sequences and *max_df* values of 0.2 and 0.3 improve performance; however, binarizing term frequencies and using *sublinear_tf* do not seem to enhance accuracy scores. The default values of γ and α parameters used by the scikit-learn implementation offer optimal performance for both Label Spreading and Propagation.

The motivation for parameter tuning is that an optimized model can be used for various versions of the same application. For both data sets, the Label Spreading model with term frequency features has produced second best results, which is why its predictions are used for the next step in semantic mapping. The chosen parameters for the *CountVectorizer* are the following: *ngram_range =(2 ,3) , binary = True , max_df =0.3*. The parameters for the Label Spreading model: *alpha=0.16, gamma=20*.

Figure 21 shows the *CountVectorizer* input consisting of the term frequency features and location attributes mapped into a 2D space with t-SNE[7]; each data point is colored according its label assigned by the Label Spreading model. The visualization shows that certain classes overlap, while others appear to be well separated. Generally, the input features represent the data in an adequate fashion.

Figure 22 shows the *CountVectorizer* input consisting of the term frequency features and location attributes mapped into a 2D space with t-SNE; each data point is colored according its label assigned by the Label Spreading model. The plot suggests that the target application data are not as well separated as the source application data, as there are many areas where classes overlap. Nevertheless, the accuracy is over 80%, which means that the features extracted from the data suffice as the model's input.

[7]The T-distributed Stochastic Neighbor Embedding is a dimensionality reduction method for visualizing high dimensional data in a 2D space [57]

Figure 21: The source application embeddings

3.4.2 Semantic Similarity Estimation

After assigning labels to all the data points in both source and target application data sets (there are no more elements whose label is equal to -1), their text attributes are concatenated based on their class label, e.g., all the elements that belong to the class 5 are merged. The outcome is a list of sentences for each data set that can be further embedded using a pre-trained language model. The XPath values and location attributes used for the classification task are not used for determining semantic similarity.

There are two Sentence-Transformer models ('all-mpnet-base-v2' and 'all-distilroberta-v1') used for embedding the sentences, both of them are available through SBERT [22], [58]–[60]. The models are chosen based on their reported average performance and the base models they were trained on. Both models are pre-trained on large amounts of data and fine-tuned in on a 1B sentence pairs data set. The resulting embeddings are N number of 768 dimensional vectors, where N is the number of classes in each data set.

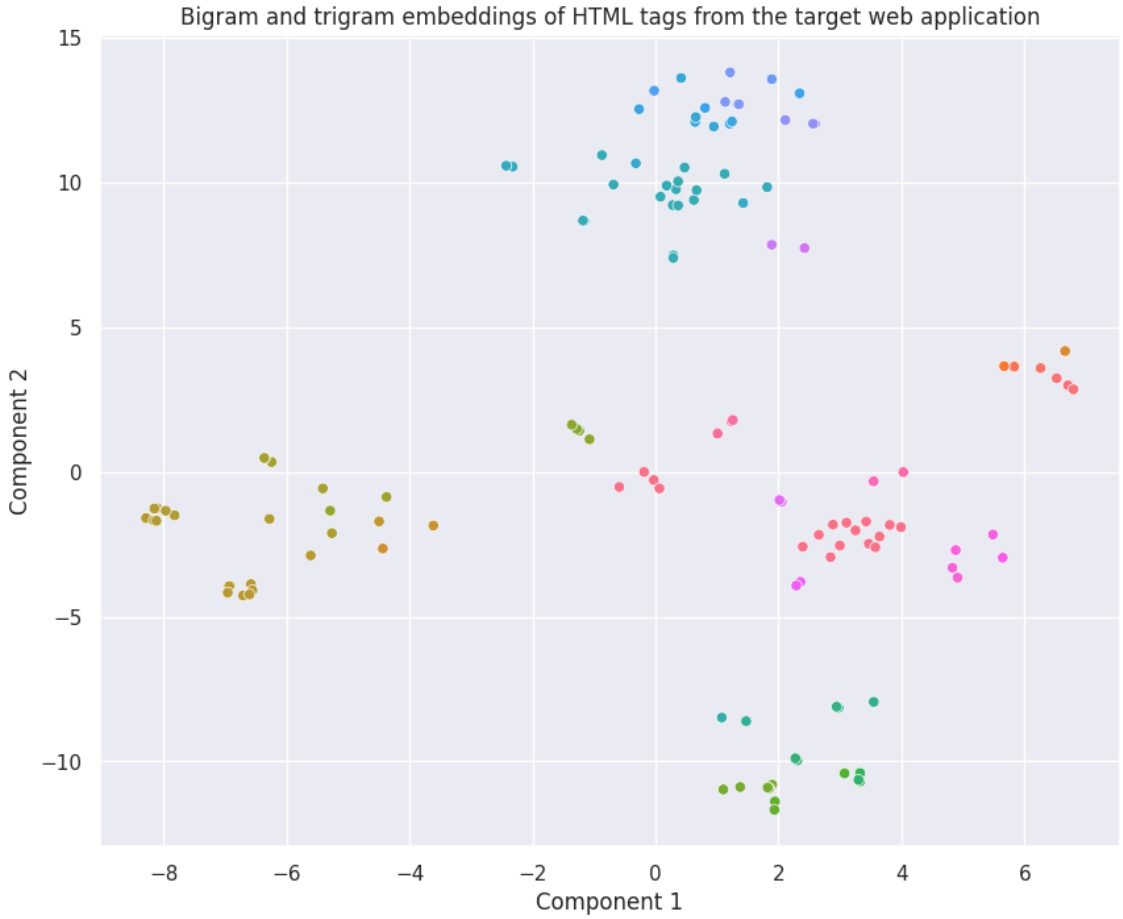


Figure 22: The target application embeddings

Figures 23 and 24 display cosine similarity matrices. The X axis represents labeled sentences from the source application, and the Y axis contains labeled sentences from the target application. The algorithm behind the construction of the matrices is introduced earlier in the chapter. Essentially, each pair of sentences is used to calculate a cosine similarity score. Since we are interested in finding a match for every target application element, we choose pairs with the highest scores. In the visualizations, the lightest cells represent semantically similar elements, whereas the dark ones are the most dissimilar. Both figures are produced by different models, but the plots show that the light and dark areas are not significantly different. The important matter in this case is whether the labels containing equivalent elements are matched correctly. To assess the performance of the semantic matching procedure, one can use the fidelity metrics defined by Zhao et al. [6], which are introduced in Chapter 2. Not all of them are applicable to this work, because their metrics are used for text reuse, not GUI element identification. Nevertheless, the ratio of correctly matched elements is an important metric in this study. There is also a need for a separate metric for those elements that exist only in one version of the

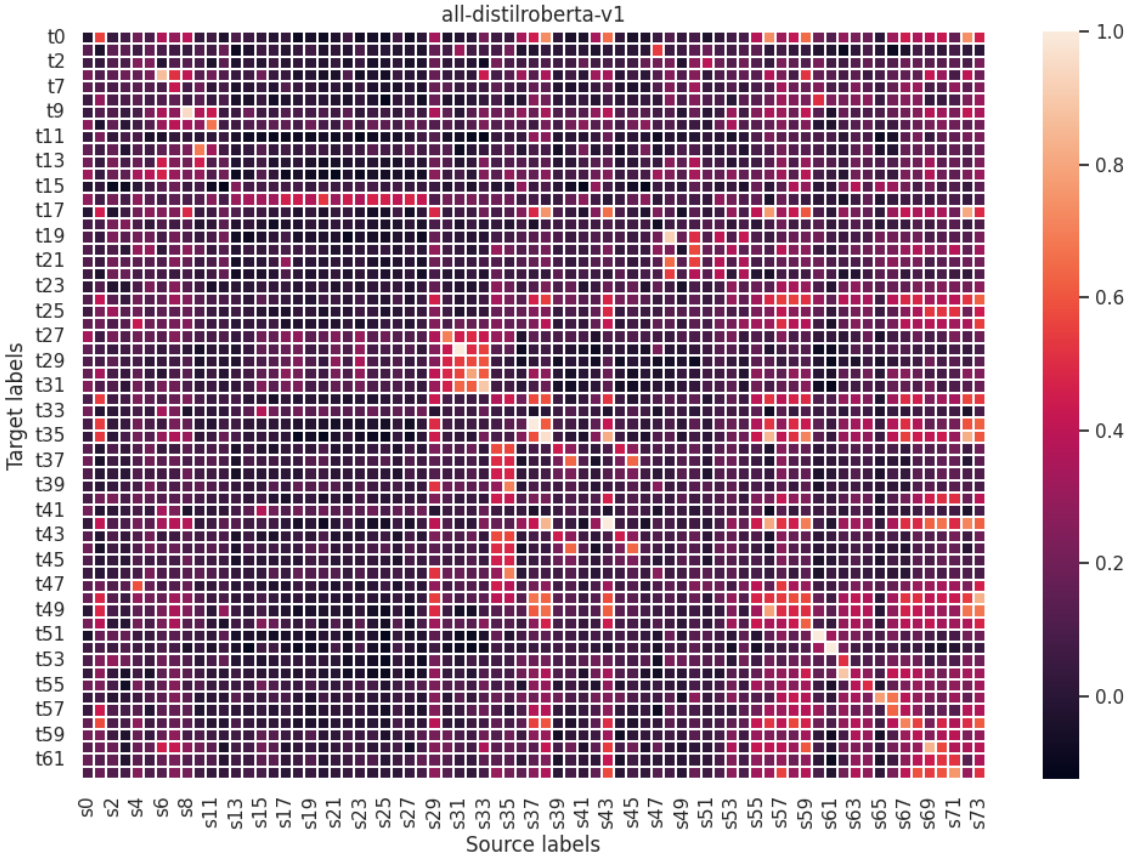


Figure 23: The cosine matrix produced with 'all-distilroberta-v1'

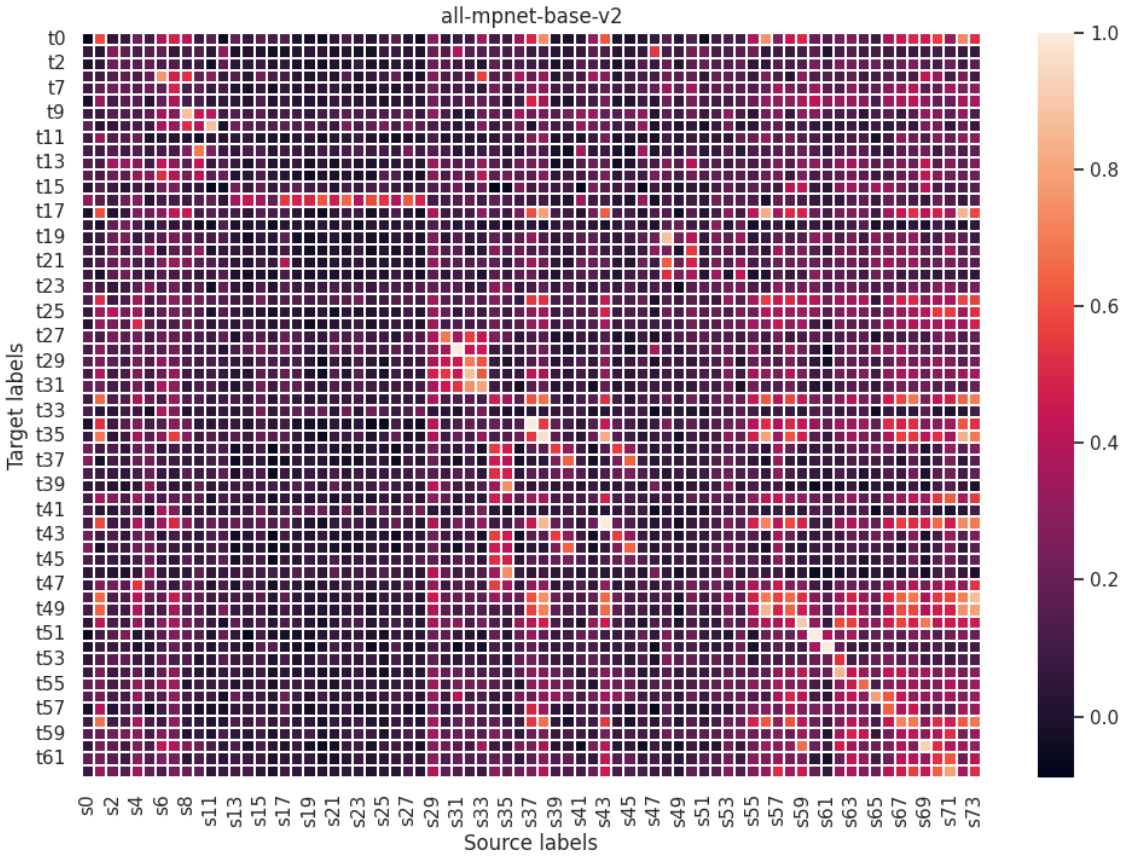


Figure 24: The cosine matrix produced with 'all-mpnet-base-v2'

application and cannot be matched correctly or are matched only partially (e.g., when not all elements belonging to a certain class are grouped together). In Table 10, they are marked as other.

Outcome	**'all-distilroberta-v1'**	**'all-mpnet-base-v2'**
Number of correct matches	0.5	0.52
Number of incorrect matches	0.27	0.25
Other	0.23	0.23

Table 10: Semantic matching results

The results showed in Table 10 produced by both models are comparable and promising. Most of the elements have been identified correctly, and the correctly matched rate is 50%. Figure 25 shows an example of correctly mapped features. However, there is a great number of incorrectly matched labels (the rate is 25%). There are several reasons for the high rate of incorrect matches:

1. Incorrectly labeled elements lead to irrelevant text appearing within the same group.

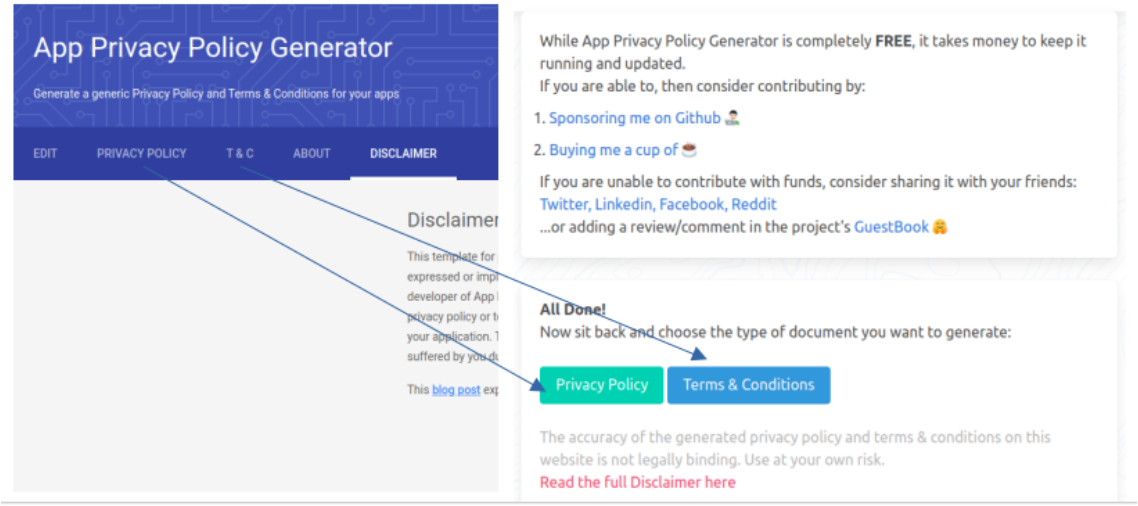


Figure 25: Correctly mapped forms between the two versions of the application [55], [56]

The classification outcomes for the target application have not been higher than 82%, which results in a lot of mixed text attributes.

2. Extracted text attributes might not be useful for semantic similarity.

3. Certain elements lack text attributes, and there is not enough text to determine their similarity with other elements.

4. Information in iconic or image content has not been properly extracted, and thus corresponding elements are not represented in the data sets.

The main challenge of this work has been the lack of available web applications with ready test suites. Commercial applications are protected by copyright and license laws that prevent their free use for research purposes. As a result, this work has had to rely on developers sharing their work online under a license that allows further modification and redistribution of their work. Unfortunately, such applications do not necessarily reflect continuous development procedures used by businesses, which is why it is hard to estimate how well the proposed tool would actually work if it could have been tested on a larger number of applications. This also constitutes a possible threat to the external validity, as it is not possible to determine whether the results obtained in this study generalize to other web applications.

Another issue is connected with manual annotation of data. Although it is possible to establish rules around such annotations, for example, only first child or parent is allowed into the same class with an element, it does not mean that other elements are devoid of cues. There could be a relevant image positioned in a close proximity that is neither

a parent or a child, or a set of children elements with relevant text down the line. There could possibly be internal errors in label and semantic similarity annotations, which would constitute a threat to the internal validity of the study.

Since the tools discussed in Chapter 2 are designed for test migration and reuse, it is hard to compare the performance of this implementation to their results. Nevertheless, with the 50% of matches being correct, the proposed solution that incorporates machine learning for extracting relevant text cues and sentence embedding of text attributes appears promising.

4 Conclusion

This work has served the purpose of implementing a semantic mapping method and exploring new application of machine learning in continuous development, specifically in UI testing. Existing research shows that test automation faces significant challenges that are related to test failure due to SUT changes. The latter appears when GUI elements that are to be tested are relocated, or their locators are changed. What is more, it has been noted that crawlers used to explore GUI of applications do not interact with a system the way a human would. They systematically go through features to find a path that would lead to a test suite execution. This makes testing slow and devoid of an actual representation of its use by application users. To mitigate such issues, semantic mapping for test reuse and transfer has been introduced. The idea behind it is that applications of the same domain share a lot of functionality, and once there is a test suite for one application, it can be transferred to another one. Following this suggestion, this study argues that the same can be done to mitigate issues of element identification in web application testing. Applications are constantly updated and changed, and robust identification of elements could resolve some issues faced by developers and testers.

Test reuse and transfer depends on efficient and accurate GUI element identification, and there has been a lot novel research on semantic mapping. It leverages contextual cues around web elements to aid in better identification of said elements. This study explores the idea that contextual information could be gathered by grouping web elements and extracting their textual information. Once elements are grouped, similarity between web element groups in different releases of the same application can be established. By finding the most similar groups, we can locate GUI elements that perform the same function in different application versions. Unlike previous research on semantic matching, this study emphasizes the use of machine learning methods for extracting textual cues.

This thesis answers several questions based on the results of the implementation of semantic mapping. First, machine learning methods offer a significant potential for extracting, representing, and gathering context around web application elements. There are three semi-supervised classifiers that have been used in this study: SOMs, Label Spreading, and Label Propagation. They have demonstrated various results based on their application in web element classification. Label Spreading and Label Propagation have demonstrated promising results with up to 88% and 83% accuracy respectively.

As earlier research has showed, using semantic meaning of textual cues can be used for semantic mapping. This study has found that if a web application provides necessary textual information within its elements' attributes, semantic matching can be a viable strategy for GUI element identification. Sentence embeddings produced with pre-trained language models have demonstrated to be suitable for determining semantic similarity for the purpose of semantic matching. According to the results of the semantic mapping technique in this study, 50% of web elements were correctly matched between different versions of the same application.

To conclude, the implementation of semantic mapping for GUI element identification has demonstrated promising results in terms of accuracy; however, its performance could be improved. This leads to the list of suggestions for further research in the area of semantic mapping for element identification:

1. As discussed in Chapter 2, the best outcomes in terms of test reuse and robust element identification have been observed in applications that share the same domain and structure. This suggests that using a supervised model could improved textual cues' extraction. It would require a data set with annotated HTML elements extracted from applications that share the same domain or a similar structure.

2. Feature selection for such a classifier could to be revisited. Although the Selenium Web Driver can provide location and text attributes of HTML elements, it sometimes fails. Additional features could be also considered.

5 Summary in Swedish – Svensk sammanfattning

Identifikation av användargränssnittselement med semantisk matchning

I denna magisteravhandling behandlas problemen med identifikation av grafiska användargränssnittselement i testning av webbapplikationer. Det primära målet är att undersöka nya metoder för att minska mänsklig ansträngning och spara tid som behövs när man testar webbapplikationer. Det har bevisats att manuella tester som är uppdaterade när applikationer ändras är fortfarande populära, fastän de kräver stora kostnader [1]–[3]. Detta beror på att automatiserad testning kräver intensiv ansträngning och mycket tid för att underhållas. Dessutom är den inte användningsbaserad, vilket är en orsak till att det kan förekomma problem då olika funktioner på webbsidor testas [4], [6]. Dessa områden utgör de största svårigheterna med automatiserad testning.

Nass et al. skriver att det finns många hinder för fullständig automatiserad testning [2]. Till exempel misslyckas tester när nya versioner av webbapplikationer lanseras. Detta beror på att grafiska användargränssnittselement byter sin plats och sitt namn. Forskare hävdar att förbättring av dess indentifikation kan lösa problemen [2]. I denna magisteravhandling framförs semantisk matchning som en metod att identifiera grafiska användargränssnittselement mellan nya och gamla versioner av befintliga webbapplikationer. Den använder semantiska betydelser av element på webbsidor för att finna motsvarigheter till dem [4], [5]. Denna metod har visat sig vara framgångsrik när den används med webbapplikationer för samma domän [4]–[8]. Existerande tekniker kan bli bättre med maskininlärning och kontextualiserad textinbäddning.

Maskininlärningen indelas traditionellt i tre typer: övervakat lärande, oövervakat lärande

och semi-övervakat lärande [28], [30]. Det behövs märkta data som har analyserats i förväg vid övervakat lärande. Till exempel kan övervakade modeller tillämpas för att förutsäga om ett mejl är skräppost eller inte. I oövervakat lärande är uppgiften att hitta en slags struktur i data. Det finns ingen information om rätta klasser. Semi-övervakat lärande kan användas när man har lite märkta data men man behöver en modell för att göra förutsägelser ändå. Tre semi-övervakade modeller utvärderas för klassificering av element i denna avhandling: Label Spreading, Label Propagation och Self-Organized Maps (SOM)[32], [34], [35]. Label Spreading och Label Propagation bygger en affinitetsmatris som beräknar likheter mellan element och upptäcker okända data [34], [35]. SOM är ett artificiellt neuralt nät vars storlek man kan välja [32]. Algoritmen lär sig av data och tilldelar dataprover olika klasser som ett resultat av den närliggande funktionen. Märkta data har större vikt än omärkta i denna modell. Algoritmerna används med textinbäddning som producerades av Sentence Bidirectional Encoder Representations from Transformers (SBERT) [22], [44]. SBERT är den senaste tekniken för att representera text i verktorformen. SBERT kan bevara kontextuell information av text på grund av Transformers arkitekturen och har visat enastående resultat i semantisk jämförelseuppgifter.

Semantisk matchning har använts för att testa webb och mobilapplikationer [4], [5], [7]–[9], [12]. Den har visat sig minska den tid och den mänskliga ansträngning som behövs för att utföra testning. Tidigare forskning har använt gamla tekniker, till exempel Word2Vec och WordNet, för att generera vektorrepresentation av text och olika metoder för att gruppera grafiska användargränssnittselement som inte baserar sig på maskininlärning. Detta arbete försöker använda maskininlärning för att klassificera och dra lärdom av användargränssnittselement på webbsidor. Den lösningen som behandlas i avhandlingen söker textinformation på webbsidor. Den anlägger ett Extensible Markup Language-träd med lxml XML toolkit och sparar ett Xpath-innehåll för varje element som kan nås med det [46]. Systemet laddar textinformation ('text' , 'alt' , 'title' , 'name' ,'value' , 'id', 'location') av elementen med Selenium Driver API [47]. Sedan grupperar en semi-övervakad modell elementen. Label Propagation, Label Spreading och SOM accepterar interaktiva element som märkta data. När data grupperas, sammanfogas text i varje grupp. Vid denna tidpunkt är det möjligt att identifiera grafiska användargränssnittselementen mellan nya och gamla versioner med den cosinuslikheten mellan ordvectorer av grupperna. Paren med den största likheten är kandidater för semantisk matchning. När man vet deras Xpath-innehåll kan tester överföras från den gamla versionen av webbapplikationen.

Det har funnits många hinder i arbetet. Det största problemet har varit att hitta en webbapplikation som kan användas i forskning. Webbapplikationer är upphovsrättsskyddade och kan inte användas för forskningsändamål. Därför har denna lösning applicerats

på en liten webbapplikation som kan generera en integritetspolicy för en annan webbapplikation. Systemet har fungerat med två versioner av webbapplikationen. Analysen av arbetsresultatet påvisar att SBERT-textinbäddning representerar semantiska betydelser av elementen. I allmänhet spelar datakvalitet en viktig roll vid maskininlärning och utfallet beror på det. Label Propagation och Label Spreading har producerat goda resultat men det är möjligt att de blir bättre med mer data av högre kvalitet.

Framtidsforskning inom semantisk matchning av grafiska användargränssnittselement kan koncentreras på multimodal maskininlärning som innehåller olika typer av information. Övervakat lärande lämpar sig förmodligen bättre för elements klassificeringsproblem, men det kräver mycket märkta data. Dataanteckning av webbsidor är inte alls ett trivialt problem och därför kräver den mänskliga ansträngningar och stora kostnader.

Bibliography

[1] J. Mahmud, A. Cypher, E. Haber, and T. Lau, "Design and industrial evaluation of a tool supporting semi-automated website testing: A TOOL SUPPORTING SEMI-AUTOMATED WEBSITE TESTING," en, *Software Testing, Verification and Reliability*, vol. 24, no. 1, pp. 61–82, Jan. 2014, ISSN: 09600833. DOI: 10.1002/stvr.1484. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.1484 (visited on 11/14/2021).

[2] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with GUI test automation (will) remain," en, *Information and Software Technology*, vol. 138, p. 106 625, Oct. 2021, ISSN: 09505849. DOI: 10.1016/j.infsof.2021.106625. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000963 (visited on 11/14/2021).

[3] M. Linares-Vasquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do Developers Test Android Applications?" en, in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai: IEEE, Sep. 2017, pp. 613–622, ISBN: 978-1-5386-0992-7. DOI: 10.1109/ICSME.2017.47. [Online]. Available: http://ieeexplore.ieee.org/document/8094467/ (visited on 11/17/2021).

[4] A. Rau, J. Hotzkow, and A. Zeller, "Efficient GUI test generation by learning from tests of other apps," en, in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, Gothenburg Sweden: ACM, May 2018, pp. 370–371, ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3195014. [Online]. Available: https://dl.acm.org/doi/10.1145/3183440.3195014 (visited on 11/09/2021).

[5] A. Rau, J. Hotzkow, and A. Zeller, "Transferring Tests Across Web Applications," en, in *Web Engineering*, T. Mikkonen, R. Klamma, and J. Hernández, Eds., vol. 10845, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 50–64, ISBN: 978-3-319-91661-3 978-3-319-91662-0. DOI: 10.1007/978-3-319-91662-0_4. [Online]. Available: http://link.springer.com/10.1007/978-3-319-91662-0_4 (visited on 11/09/2021).

[6] Y. Zhao, J. Chen, A. Sejfia, *et al.*, "FrUITeR: A framework for evaluating UI test reuse," en, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA: ACM, Nov. 2020, pp. 1190–1201, ISBN: 978-1-4503-7043-1. DOI: 10.1145/3368089.3409708. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3409708 (visited on 11/05/2021).

[7] F. Behrang and A. Orso, "Test migration for efficient large-scale assessment of mobile app coding assignments," en, in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Amsterdam Netherlands: ACM, Jul. 2018, pp. 164–175, ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213854. [Online]. Available: https://dl.acm.org/doi/10.1145/3213846.3213854 (visited on 11/13/2021).

[8] F. Behrang and A. Orso, "Test Migration Between Mobile Apps with Similar Functionality," en, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA: IEEE, Nov. 2019, pp. 54–65, ISBN: 978-1-72812-508-4. DOI: 10.1109/ASE.2019.00016. [Online]. Available: https://ieeexplore.ieee.org/document/8952387/ (visited on 11/13/2021).

[9] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test Transfer Across Mobile Apps Through Semantic Mapping," en, in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA: IEEE, Nov. 2019, pp. 42–53, ISBN: 978-1-72812-508-4. DOI: 10.1109/ASE.2019.00015. [Online]. Available: https://ieeexplore.ieee.org/document/8952228/ (visited on 11/05/2021).

[10] G. Hu, L. Zhu, and J. Yang, "AppFlow: Using machine learning to synthesize robust, reusable UI tests," en, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 269–282, ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236055. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3236055 (visited on 11/24/2021).

[11] S. Rossel, *Continuous Integration, Delivery, and Deployment*, eng. Birmingham: Packt Publishing, 2017, OCLC: 1011246395, ISBN: 978-1-78728-418-0.

[12] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of GUI events for test reuse: Are we there yet?" en, in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Denmark: ACM, Jul. 2021, pp. 177–190, ISBN: 978-1-4503-8459-9. DOI: 10.1145/3460319.3464827. [Online]. Available: https://dl.acm.org/doi/10.1145/3460319.3464827 (visited on 11/14/2021).

[13] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," en, *Information and Software Technology*, vol. 55, no. 10, pp. 1679–1694, Oct. 2013, ISSN: 09505849. DOI: 10.1016/j.infsof.2013.03.004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584913000669 (visited on 11/09/2021).

[14] V. Nguyen, T. To, and G.-H. Diep, "Generating and selecting resilient and maintainable locators for Web automated testing," en, *Software Testing, Verification and Reliability*, vol. 31, no. 3, May 2021, ISSN: 0960-0833, 1099-1689. DOI: 10.1002/stvr.1760. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.1760 (visited on 11/09/2021).

[15] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," en, in *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, San Jose, CA, USA: ACM Press, 2014, pp. 304–314, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610390. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2610384.2610390 (visited on 11/14/2021).

[16] M. E. Akpınar and Y. Yesilada, "Vision Based Page Segmentation Algorithm: Extended and Perceived Success," en, in *Current Trends in Web Engineering*, Q. Z. Sheng and J. Kjeldskov, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2013, pp. 238–252, ISBN: 978-3-319-04244-2. DOI: 10.1007/978-3-319-04244-2_22.

[17] *Activity*, en. [Online]. Available: https://developer.android.com/reference/android/app/Activity (visited on 04/24/2023).

[18] *App resources overview*, en. [Online]. Available: https://developer.android.com/guide/topics/resources/providing-resources (visited on 04/24/2023).

[19] *Fragment*, en. [Online]. Available: https://developer.android.com/reference/android/app/Fragment (visited on 04/24/2023).

[20] E. Ristad and P. Yianilos, "Learning string-edit distance," en, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, May 1998, ISSN: 01628828. DOI: 10.1109/34.682181. [Online]. Available: http://ieeexplore.ieee.org/document/682181/ (visited on 11/29/2021).

[21] K. W. Church, "Word2Vec," en, *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, Jan. 2017, ISSN: 1351-3249, 1469-8110. DOI: 10.1017/S1351324916000334. [Online]. Available: https://www.cambridge.org/core/product/identifier/S1351324916000334/type/journal_article (visited on 11/17/2021).

[22] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," *arXiv:1908.10084 [cs]*, Aug. 2019, arXiv: 1908.10084. [Online]. Available: http://arxiv.org/abs/1908.10084 (visited on 10/19/2021).

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805 [cs]*, May 2019, arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805 (visited on 10/19/2021).

[24] C. Wang, P. Nulty, and D. Lillis, "A Comparative Study on Word Embeddings in Deep Learning for Text Classification," en, in *Proceedings of the 4th International Conference on Natural Language Processing and Information Retrieval*, Seoul Republic of Korea: ACM, Dec. 2020, pp. 37–46, ISBN: 978-1-4503-7760-7. DOI: 10.1145/3443279.3443304. [Online]. Available: https://dl.acm.org/doi/10.1145/3443279.3443304 (visited on 11/26/2021).

[25] Z. Liu, Y. Lin, and M. Sun, *Representation Learning for Natural Language Processing*, en. Singapore: Springer Singapore, 2020, ISBN: 9789811555725 9789811555732. DOI: 10.1007/978-981-15-5573-2. [Online]. Available: http://link.springer.com/10.1007/978-981-15-5573-2 (visited on 11/19/2021).

[26] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, en, 2nd ed., ser. Springer Texts in Statistics. New York, NY: Springer New York, 2021, ISBN: 978-1-07-161418-1. [Online]. Available: https://www.statlearning.com/ (visited on 11/30/2021).

[27] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966, ADS Bibcode: 1966SPhD...10..707L. [Online]. Available: https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L (visited on 11/26/2022).

[28] K. P. Murphy, *Machine learning: a probabilistic perspective*, en, ser. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012, ISBN: 978-0-262-01802-9.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[30] T. Jo, *Machine Learning Foundations: Supervised, Unsupervised, and Advanced Learning*, en. Cham: Springer International Publishing, 2021, ISBN: 978-3-030-65899-1 978-3-030-65900-4. DOI: 10.1007/978-3-030-65900-4. [Online]. Available: http://link.springer.com/10.1007/978-3-030-65900-4 (visited on 12/02/2021).

[31] L. Han, G. Yang, H. Dai, *et al.*, "Combining self-organizing maps and biplot analysis to preselect maize phenotypic components based on UAV high-throughput phenotyping platform," *Plant Methods*, vol. 15, no. 1, p. 57, May 2019, ISSN: 1746-4811. DOI: 10.1186/s13007-019-0444-6. [Online]. Available: https://doi.org/10.1186/s13007-019-0444-6 (visited on 12/15/2021).

[32] F. M. Riese and S. Keller, *SuSi: Supervised Self-Organizing Maps for Regression and Classification in Python*, arXiv: 1903.11114, Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.2609130 (visited on 12/07/2021).

[33] T. Kohonen, "Essentials of the self-organizing map," en, *Neural Networks*, vol. 37, pp. 52–65, Jan. 2013, ISSN: 08936080. DOI: 10.1016/j.neunet.2012.09.018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0893608012002596 (visited on 12/15/2021).

[34] X. Zhu and Z. Ghahramani, "Learning from Labeled and Unlabeled Data with Label Propagation," en, p. 8, 2002.

[35] B. Yoshua, D. Olivier, and R. Nicolas Le, "Label Propagation and Quadratic Criterion," en, in *Semi-Supervised Learning*, O. Chapelle, B. Scholkopf, and A. Zien, Eds., The MIT Press, Sep. 2006, pp. 192–216, ISBN: 978-0-262-03358-9. DOI: 10.7551/mitpress/9780262033589.003.0011. [Online]. Available: http://mitpress.universitypressscholarship.com/view/10.7551/mitpress/9780262033589.001.0001/upso-9780262033589-chapter-11 (visited on 01/25/2022).

[36] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf, "Learning with Local and Global Consistency," en, p. 8, 2004.

[37] Z. S. Harris, "Distributional Structure," en, *WORD*, vol. 10, no. 2-3, pp. 146–162, Aug. 1954, ISSN: 0043-7956, 2373-5112. DOI: 10.1080/00437956.1954.11659520. [Online]. Available: http://www.tandfonline.com/doi/full/10.1080/00437956.1954.11659520 (visited on 11/25/2021).

[38] J. Firth, "A Synopsis of Linguistic Theory 1930-1955," in *Studies in Linguistic Analysis*, Philological Society, Oxford, 1957.

[39] D. Rozado, "Wide range screening of algorithmic bias in word embedding models using large sentiment lexicons reveals underreported bias types," en, *PLOS ONE*, vol. 15, no. 4, C. Schwieren, Ed., e0231189, Apr. 2020, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0231189. [Online]. Available: https://dx.plos.org/10.1371/journal.pone.0231189 (visited on 11/25/2021).

[40] O. Levy, "Word Representation," en, in *The Oxford Handbook of Computational Linguistics 2nd edition*, R. Mitkov, Ed., Oxford University Press, Sep. 2018, ISBN: 978-0-19-957369-1. DOI: 10.1093/oxfordhb/9780199573691.013.57. [Online]. Available: https://oxfordhandbooks.com/view/10.1093/oxfordhb/9780199573691.001.0001/oxfordhb-9780199573691-e-57 (visited on 11/19/2021).

[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," en, *arXiv:1301.3781 [cs]*, Sep. 2013, arXiv: 1301.3781. [Online]. Available: http://arxiv.org/abs/1301.3781 (visited on 11/16/2021).

[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and their Compositionality," en, *Advances in Neural Information Processing Systems*, vol. 26, p. 9, Oct. 2013.

[43] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic Regularities in Continuous Space Word Representations," en, *N13-1090*, vol. Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, p. 6, 2013.

[44] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention Is All You Need," en, *arXiv:1706.03762 [cs]*, Dec. 2017, arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762 (visited on 11/19/2021).

[45] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A Unified Embedding for Face Recognition and Clustering," en, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, arXiv:1503.03832 [cs], Jun. 2015, pp. 815–823. DOI: 10.1109/CVPR.2015.7298682. [Online]. Available: http://arxiv.org/abs/1503.03832 (visited on 11/25/2022).

[46] *Lxml - Processing XML and HTML with Python*. [Online]. Available: https://lxml.de/index.html#introduction (visited on 11/13/2022).

[47] *Selenium Overview*, en. [Online]. Available: https://www.selenium.dev/documentation/overview/ (visited on 11/19/2022).

[48] *CSS: Cascading Style Sheets | MDN*, en-US, Apr. 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS (visited on 04/25/2023).

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] D. Jurafsky and J. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Feb. 2008, vol. 2.

[51] *6.2. Feature extraction*, en. [Online]. Available: https://scikit-learn/stable/modules/feature_extraction.html (visited on 04/24/2023).

[52] *1.14. Semi-supervised learning*, en. [Online]. Available: https://scikit-learn.org/stable/modules/semi_supervised.html#label-propagation (visited on 11/28/2022).

[53] *Semantic Textual Similarity — Sentence-Transformers documentation*. [Online]. Available: https://www.sbert.net/docs/usage/semantic_textual_similarity.html (visited on 04/25/2023).

[54] T. pandas development team, *Pandas-dev/pandas: Pandas*, version latest, Feb. 2020. DOI: 10.5281/zenodo.3509134. [Online]. Available: https://doi.org/10.5281/zenodo.3509134.

[55] N. Srivastava, *App Privacy Policy Generator*. [Online]. Available: https://app-privacy-policy-generator.nisrulz.com/ (visited on 11/28/2022).

[56] N. Srivastava, *Nisrulz/app-privacy-policy-generator*, original-date: 2017-02-21T17:38:38Z, Apr. 2023. [Online]. Available: https://github.com/nisrulz/app-privacy-policy-generator (visited on 04/25/2023).

[57] *Sklearn.manifold.TSNE*, en. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.manifold.TSNE.html (visited on 11/28/2022).

[58] *Sentence-transformers/all-mpnet-base-v2 · Hugging Face*. [Online]. Available: https://huggingface.co/sentence-transformers/all-mpnet-base-v2 (visited on 11/28/2022).

[59] *Sentence-transformers/all-distilroberta-v1 · Hugging Face*. [Online]. Available: https://huggingface.co/sentence-transformers/all-distilroberta-v1 (visited on 11/28/2022).

[60] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: Smaller, faster, cheaper and lighter,” *ArXiv*, vol. abs/1910.01108, 2019.

Appendix A – Parameter Tuning

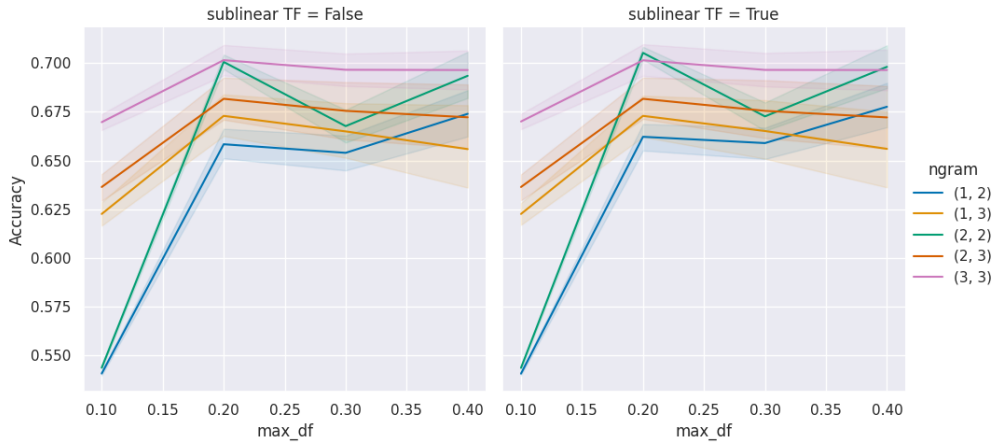


Figure A.1: TF-IDF vectorizer parameters for the source application (Logistic Regression)

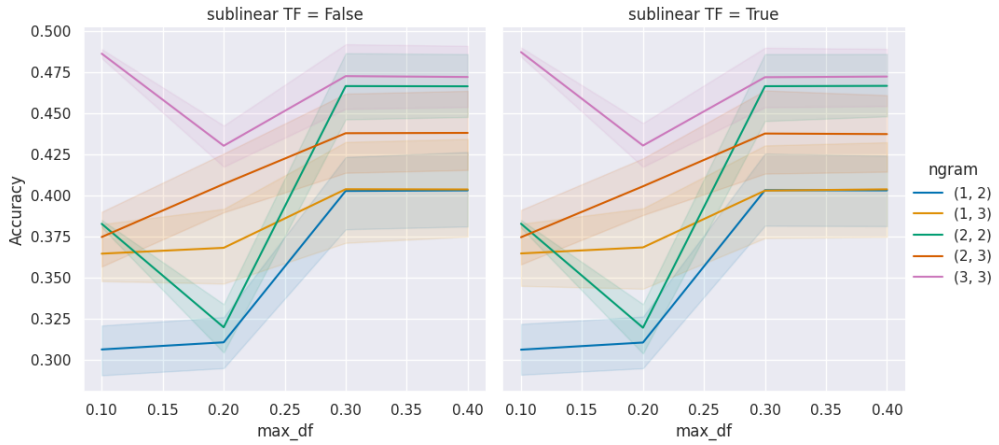


Figure A.2: TF-IDF vectorizer parameters for the target application (Logistic Regression)

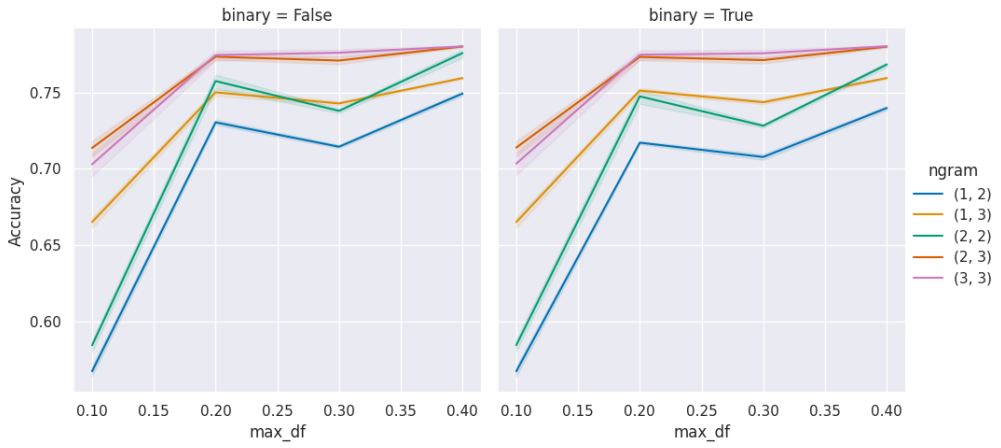


Figure A.3: Count vectorizer parameters for the source application (Logistic Regression)

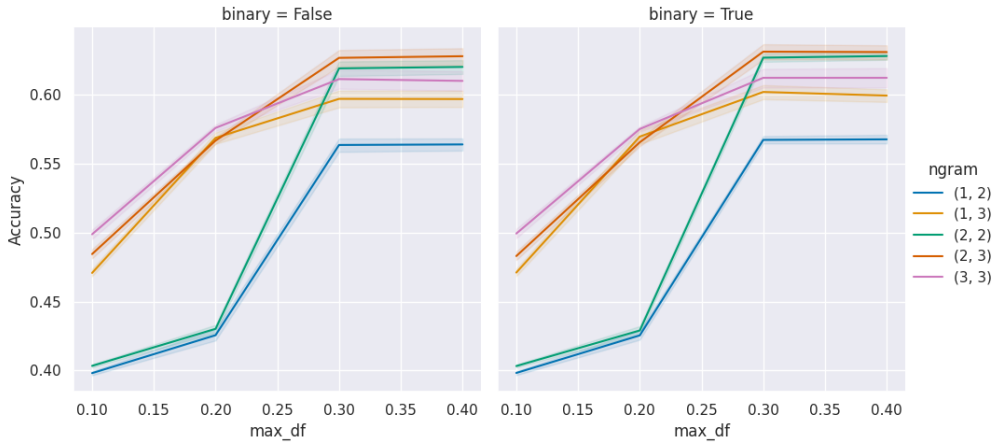


Figure A.4: Count vectorizer parameters for the target application (Logistic Regression)

Appendix B – Code

```
import numpy as np
import pandas as pd
from sklearn.semi_supervised import LabelSpreading,
    LabelPropagation
from sklearn.metrics import accuracy_score

token_pattern = r'(?u)\b\w+(?:\[.?\])?'

#Parameters to be tested
ngram = [(1,3),(2,2),(3,3),(2,3),(1,2)]
max_df = np.arange(.1,.5,.1)
bool_p = [True, False]
true_l = data['true_label']
labels = data['class']
gamma = np.arange(5,41,5)
alpha = np.arange(.01, .17, .05)

#Label Spreading with TF-IDF
def tune_labelspreadingTFIDF(data):
  results_tfidf = {'ngram':[], 'max_df':[],'sublinear_tf':[],'gamma
    ':[], 'alpha':[], 'accuracy':[]}
  for n in ngram:
    for m in max_df:
      for s in bool_p:
        preprocessorTFIDF = make_column_transformer(
          (TfidfVectorizer(token_pattern = token_pattern,
    ngram_range=n, sublinear_tf=s,max_df=m), 'xpath'),
          (MinMaxScaler(), ['x', 'y']),
          sparse_threshold=0)
        X1tfidf = preprocessorTFIDF.fit_transform(data)
        for i in gamma:
```

```
          for j in alpha:
            label_sp_model = LabelSpreading(gamma=i, alpha=j)
            label_sp_model.fit(X1tfidf, labels)
            y_pred = label_sp_model.transduction_
            results_tfidf['gamma'].append(i)
            results_tfidf['alpha'].append(j)
            results_tfidf['ngram'].append(n)
            results_tfidf['max_df'].append(m)
            results_tfidf['sublinear_tf'].append(s)
    results_tfidf['accuracy'].append(accuracy_score(true_l, y_pred))
  results_tfidf = pd.DataFrame(results_tfidf)
  return results_tfidf

#Label Spreading with CountVectorizer
def tune_labelspreadingCV(data):
    results_cv = {'ngram':[], 'max_df':[],'binary':[],'gamma':[], '
    alpha':[], 'accuracy':[]}
    for n in ngram:
     for m in max_df:
       for s in bool_p:
         preprocessorCV = make_column_transformer(
           (CountVectorizer(token_pattern = token_pattern,
    ngram_range=n, binary=s,max_df=m), 'xpath'),
           (MinMaxScaler(), ['x', 'y']),
           sparse_threshold=0)
         X1cv = preprocessorCV.fit_transform(data)
         for i in gamma:
           for j in alpha:
             label_sp_model = LabelSpreading(gamma=i, alpha=j)
             label_sp_model.fit(X1cv, labels)
             y_pred = label_sp_model.transduction_
             results_cv['gamma'].append(i)
             results_cv['alpha'].append(j)
             results_cv['ngram'].append(n)
             results_cv['max_df'].append(m)
             results_cv['binary'].append(s)        results_cv['
    accuracy'].append(accuracy_score(true_l, y_pred))
  results_cv = pd.DataFrame(results_cv)
  return results_cv

#Label Propagation with TF-IDF
def tune_labelpropagationTFIDF(data):
  results_tfidf = {'ngram':[], 'max_df':[],'sublinear_tf':[],'gamma
    ':[], 'alpha':[], 'accuracy':[]}
  for n in ngram:
```

```
    for m in max_df:
      for s in bool_p:
        preprocessorTFIDF = make_column_transformer(
          (TfidfVectorizer(token_pattern = token_pattern,
   ngram_range=n, sublinear_tf=s,max_df=m), 'xpath'),
          (MinMaxScaler(), ['x', 'y']),
          sparse_threshold=0)
        X1tfidf = preprocessorTFIDF.fit_transform(data)
        for i in gamma:
          label_sp_model = LabelPropagation(gamma=i, max_iter
   =10000)
          label_sp_model.fit(X1tfidf, labels)
          y_pred = label_sp_model.transduction_
          results_tfidf['gamma'].append(i)
          results_tfidf['ngram'].append(n)
          results_tfidf['max_df'].append(m)
          results_tfidf['sublinear_tf'].append(s)
   results_tfidf['accuracy'].append(accuracy_score(true_l, y_pred))
  results_tfidf = pd.DataFrame(results_tfidf)
  return results_tfidf

#Label Propagation with CountVectorizer
def tune_labelpropagationCV(data):
  results_cv = {'ngram':[], 'max_df':[],'binary':[],'gamma':[], '
   alpha':[], 'accuracy':[]}
  for n in ngram:
    for m in max_df:
      for s in bool_p:
        preprocessorCV = make_column_transformer(
          (CountVectorizer(token_pattern = token_pattern,
   ngram_range=n, binary=s,max_df=m), 'xpath'),
          (MinMaxScaler(), ['x', 'y']),
          sparse_threshold=0)
        X1cv = preprocessorCV.fit_transform(data)
        for i in gamma:
          label_sp_model = LabelPropagation(gamma=i, max_iter
   =10000)
          label_sp_model.fit(X1cv, labels)
          y_pred = label_sp_model.transduction_
          results_cv['gamma'].append(i)
          results_cv['ngram'].append(n)
          results_cv['max_df'].append(m)
          results_cv['binary'].append(s)           results_cv['
   accuracy'].append(accuracy_score(true_l, y_pred))
  results_cv = pd.DataFrame(results_cv)
```

```
  return results_cv
```

```
import susi

#Testing SOM
#Splitting data into training and test data sets
def split_data(data):
  unlabeled = data.index[data['class'] == -1].values
  #Every second unlabeled data point is placed in the training data
  unlabeled_train = unlabeled[0:-1:2]
  test_idx = np.setdiff1d(unlabeled, unlabeled_train)
  train_idx = np.concatenate((unlabeled_train, data.index[data['
   class'] != -1].values), axis=0)
  return train_idx, test_idx

train, test = split_data(data)

#Extracting input features
preprocessor = make_column_transformer(
    #(CountVectorizer(token_pattern = token_pattern, ngram_range
   =(3,3), binary=True, max_df=0.3), 'xpath'),
    (TfidfVectorizer(token_pattern = token_pattern, ngram_range
   =(3,3), sublinear_tf=True, max_df=0.3), 'xpath'),
    (MinMaxScaler(), ['x', 'y']),
    sparse_threshold=0
)
train_data = preprocessor.fit_transform(data.iloc[train])
test_data = preprocessor.transform(data.iloc[test])

#Fitting and predicting labels with SOM
def predict_SOM(train_data, test_data, train_labels, true_labels,
   columns, rows, rate_start, rate_end):
    som = susi.SOMClassifier(random_state=0,
   missing_label_placeholder=-1,
        n_columns=columns,
        n_rows=rows,
        learning_rate_start = rate_start,
        learning_rate_end = rate_end)
    som.fit(train_data, train_labels)
    y_pred = som.predict(test_data)
    accuracy = metrics.accuracy_score(true_labels, y_pred)
    print(accuracy)
```