On Twitter Bots Behaving Badly: A Manual and Automated Analysis of Python Code Patterns on GitHub

Andrea Millimaggi and Florian Daniel*

Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy
E-mail: andrea.millimaggi@mail.polimi.it; florian.daniel@polimi.it
**Corresponding Author*

Received 04 December 2019; Accepted 18 December 2019;
Publication xx xxxxxxx xxxx

Abstract

Bots, i.e., algorithmically driven entities that behave like humans in online communications, are increasingly infiltrating social conversations on the Web. If not properly prevented, this presence of bots may cause harm to the humans they interact with. This article aims to understand which types of abuse may lead to harm and whether these can be considered intentional or not. We manually review a dataset of 60 Twitter bot code repositories on GitHub, derive a set of potentially abusive actions, characterize them using a taxonomy of abstract code patterns, and assess the potential abusiveness of the patterns. The article then describes the design and implementation of a code pattern recognizer and uses the pattern recognizer to automatically analyze a dataset of 786 Python bot code repositories. The study does not only reveal the existence of 28 communication-specific code patterns – which could be used to assess the harmfulness of bot code – but also their consistent presence throughout all studied repositories.

Keywords: Bots, Harm, Abuse, Code patterns, Pattern recognition, GitHub, Twitter, Python.

Journal of Web Engineering, Vol. 18_8, 1–36.
doi: 10.13052/jwe1540-9589.1883

1 Introduction

Social networks, microblogging or instant messaging services like Facebook, Twitter, Instagram, LinkedIn, WhatsApp, Telegram, and similar are the foundation of the Web 2.0, that is, the Web made of content and services provided by the users themselves. Over the last 15 years, these applications have enabled users all around the world to stay informed, share ideas and discuss opinions. In short, they revolutionized online communication to billions of humans.

In the recent years, a new phenomenon has arisen: *bots*, i.e., algorithmically driven entities that behave like humans in online communications and increasingly participate in conversations without the human participants necessarily being aware of communicating with a machine [8]. State-of-the-art artificial intelligence, speech technology and conversational technology enable the implementation of software agents whose communications are only hardly distinguishable from those by human agents. Combined with generally low transparency about the true nature of bot accounts, humans are easily fooled.

In [10], we have started asking ourselves whether the increasing presence of bots may lead to harmful human-bot interactions that may hurt the human participant in the conversation, and by searching for papers, news, blog posts, and similar we found a variety of anecdotal evidence that this may indeed happen. Of course, bots are not harmful in general. But sometimes, intentionally or unintentionally, software-driven conversations may just break common conversational rules, etiquette, or even the law. For instance, in the prior study we found anecdotal evidence of bots that used language that was not suitable to children, that pretended to be real women in online dating sites, or that engaged in conversations with other accounts that had offensive or discriminating account names. Note that in the former two examples it's the users of the bots that got hurt, while in the latter example it's the owner of the account that suffered a damage to its public image. It is important to acknowledge the problem, to be able to provide countermeasures and to prevent people or organizations from getting hurt.

As we show in our discussion of related works, most of the literature today focuses on the detection of bots and on telling bots and humans apart starting from the evidence (e.g., posts, comments, likes) that is observable and accessible online. There is very little information on assessing the harms caused by this presence of bots, even less so on the reasons that lead to harm. This article studies this latter aspect and aims to identify how harm is caused by bots to understand the likely, underlying intentions. Doing so requires looking behind the curtain, away from the content shared online and into the actual code implementing the bots' communication logic.

The contributions of this article are:

- The construction of a *dataset* of social bot GitHub code repositories for Twitter; the analysis focuses on code written in Python and on project metadata.
- An *abuse-oriented classification* of bot code repositories according to how the developers themselves advertise their projects.
- A *qualitative, systematic code review* that identifies 28 potentially *abusive code patterns* that may lead to harmful interactions with human users and a discussion of the *harmfulness* and possible *intentions* underlying these patterns.
- The implementation of an *automated pattern recognizer* for Python code repositories able to search for potentially abusive code patterns.
- A *large-scale analysis* of 786 code repositories providing empirical evidence for how widespread the use of the identified patterns is in state-of-the-art bot implementations.

Next, we elaborate on the background of the work, then in Section 3 we describe the dataset we use for our study and report on a preliminary analysis of the data. In Section 4, we detail the method underlying the analysis and describe the respective results: actions, patterns and possible consequences. In Section 5, we introduce the pattern recognizer for automated code pattern search and report on the results we obtained by applying it to a large dataset of code repositories. After summarizing the most related works, we discuss our findings, conclude the article and outline future works.

2 Background

2.1 Harm and Abuse in Human-bot Interactions

Harm occurs when someone suffers an injury or damage, but also when someone gets exposed to a potential adverse effect or danger. In prior work [10], we analyzed empirical evidence of harms caused by bots identified the following types of harm caused by bots:

- *Psychological harm*: it occurs when someone's psychological health or wellbeing gets endangered or injured. An example of a bot causing psychological harm is Boost Juice's Messenger bot that was meant as a funny channel to obtain discounts by mimicking a dating game with fruits but used language that was not appropriate for children (http://bit.ly/2zvNt0E).

- *Legal harm*: it occurs when someone becomes subject to law enforcement or prosecution. A good example is the case of Jeffry van der Goot, a Dutch developer who had to shut down his Twitter bot generating random posts, after it sent out death threats to other users (http://bit.ly/2Dfm71P).
- *Economic harm*: it occurs when someone incurs in monetary cost or loses time that could have been spent differently. For example, in 2014 the bot wise_shibe provided automated answers on Reddit and users rewarded the bot with tips in the digital currency Dogecoin, convinced they were tipping a real user (http://bit.ly/2zu2b6r).
- *Social harm*: it occurs when someone's image or standing in a community gets affected negatively. An example of a bot causing social harm was documented by Jason Slotkin whose Twitter identity was cloned by a bot, confusing friends and followers (http://bit.ly/2Dfq4DH).
- *Democratic harm*: it occurs when democratic rules and principles are undermined and society as a whole suffers negative consequences. Bessi and Ferrara [3], for instance, showed that bots were pervasively active in the on-line political discussion of the 2016 U.S. Presidential election.

These types of harm may happen while bots perform regular *actions*, such as posting a message or commenting a message by someone else, that are not harmful per se and that also human users would perform. What needs to happen in order to cause harm is the abusive implementation of these actions. *Abuses* we found are: *disclosing sensitive facts*, *denigrating*, *being grossly offensive*, *being indecent or obscene*, *being threatening*, *making false allegations*, *deceiving users*, *spamming*, *spreading misinformation*, *mimicking interest*, *cloning profiles*, and *invading spaces* that are not meant for bots. Some of these may be subject to legal prosecution (e.g., threatening people), others only breach moral, ethical or social norms, yet they still may be harmful to unprepared human users. Note that these abuses may happen in any of today's social networks and are not limited to any one in specific.

2.2 Platform Policies and Permissions

In order to properly assess the behavior of a bot, it is important to understand the position of the platforms targeted by bots in relation to automation through bots. For this purpose, we manually surveyed the *usage policies* of a selection of social networks (Facebook, Twitter, Tumblr), instant messaging platforms (Telegram, Whatsapp, Facebook Messenger), platforms for media sharing (Instagram, Pinterest), a professional network (LinkedIn) and Reddit.

All platforms provide developers with *programmable interfaces* (APIs) that can be used for the development of bots; Messenger and Telegram even come with APIs specifically tailored to bots, more specifically, chatbots. Whatsapp is the platform that is most restricted: its Business API allows the implementation of bots, but it seems limited to company use only; however, Android intents (https://bit.ly/2RwjScE) can be used locally on the mobile phone to interact with Whatsapp programmatically. Where an API is provided, it typically allows programmatic access to essentially *all functionalities* that would also be available to users via the platforms' user interfaces. Users of the APIs must *authenticate* with the platforms (the preferred protocol is OAuth) and obtain a *token* enabling programmatic access; only Telegram gives tokens without authentication. All of the studied APIs are *REST APIs*; Facebook and Twitter also provide access to *streaming, live data*. To ease development, some platforms (Facebook, Messenger, LinkedIn) are equipped with developer-oriented *software development kits* (SDKs), even in multiple programming languages. Others (Instagram) provide more basic programming *libraries*. Twitter provides both an SDK and a more basic library.

As for the usage policies, almost all platforms impose some kind of *limitation*. For instance, "200 calls per hour per user" per app on Facebook. Twitter uses message-level limits, e.g., to prevent aggressive following practices. Only Messenger does not explicitly limit usage and instead even states "you can safely send 250 requests per second." Some platforms impose specific *requirements*, such as "keep your app's negative feedback below our threshold" (Facebook) or "automated bots must respond to any and all input from the user" (Messenger). An explicit *code review* is needed for Facebook, Instagram and Messenger. *Automation* is generally allowed, although commonly limited to actions the target users have explicitly granted permission to; Twitter, for instance, disallows "sending messages in an aggressive or discriminate manner." Most policies even include *content restrictions* like "don't create fake accounts" (Facebook) or "don't send tweets containing links that are misleading." All surveyed platforms explicitly state that they may *suspend* accounts or apps if they violate their policies.

3 Dataset: Twitter Bot Code Repositories

This article follows a Data Science methodology [9] to extract new knowledge from data. We thus describe here the dataset underlying our study and provide a first analysis of how developers themselves describe their own bot projects.

3.1 Data Sources and Retrieval

In this article, we specifically focus on Twitter (https://twitter.com) and bots written in Python. The former is an opportunistic choice, shared by most literature on the topic (see Section 6 for related works) and is motivated by the openness of Twitter compared to other platforms. The latter stems from the observation that Python is the most used language for Twitter bot implementations in GitHub (35.4% of the repositories we analyzed for Twitter use it). GitHub (https://github.com) is the code hosting service we use for data collection; the choice is again driven by adoption: with about 31M users and 100M projects (or "repositories"), GitHub is today's most used code hosting service (https://www.alexa.com/topsites/category/Computers/Open_Source/Project_Hosting).

In order to identify candidate repositories for our analysis, we used GitHub's search API with a combination of two terms, "Twitter" and any among "bot," "automation," "auto" and "automated." Figure 1 shows the distribution of results obtained by the search considering still all programming languages. As the result of the query "Twitter bot" shows, the term "bot" is highly used for Twitter (we performed similar searches for all platforms mentioned in Section 2.2, and the results distributions do vary from platform to platform). The search represents the state of GitHub as of

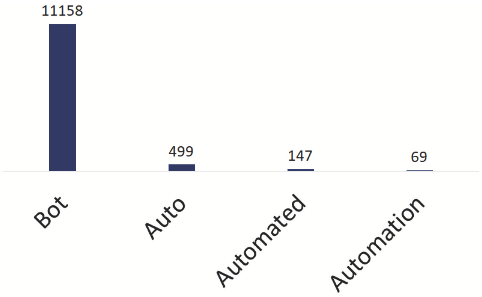


Figure 1 Distribution of GitHub search results by searched keywords (includes all programming languages.

October 29, 2018, the date the search was performed. For each identified repository, we collected all code files included in the repository as well as a subset of the respective project metadata: URL, programming language, description (a short line of text), and fork/subscriber/watcher counts.

The data collected following this methodology is by its very nature specific to Twitter, Python and GitHub. For other combinations of these three parameters, the same methodology can be used to collect and analyze similar data/metadata.

3.2 Preliminary Analysis

As the purpose of this article is to understand how bots implement their interactions with humans, the analysis necessarily requires a manual inspection. This, in turn, requires a careful selection of repositories, in order to keep the size of the dataset manageable and the selected repositories meaningful. Before choosing which repositories to keep and which not, we thus run a simple analysis based on the textual descriptions of the projects in order to obtain a preliminary understanding of which actions the repositories implement.

The analysis followed a top-down approach: We took as starting point the actions identified in our previous work [10], i.e., *talk with user*, *redirect user*, *write post*, *comment post*, *forward post*, *like message*, *follow user*, and *create user*, and matched the retrieved repositories with these action labels. In order to match repositories with action labels, we manually inspected the descriptions of the first 100 items as returned in order of relevance by the GitHub search API and extracted textual keywords from the descriptions. Examples of keywords are: *send messages*, *reply to messages*, *chat*, *post*, *tweet*, *tag*, *poke*, and similar. Then we mapped all keywords to respective action labels, such as $\{$*send messages, reply to messages, read messages, direct message, chat*$\} \rightarrow$ *talk*.

The mapping exercise produced evidence for the existence in the dataset of all the actions above, plus the addition of 3 new action labels: some projects explicitly claimed to implement a *spam* functionality; others implemented a *poke user* and a *recommend user* functionality.

According to [10], spamming is actually an abuse of the actions *write post* or *forward post*, but we kept it as the descriptions explicitly use the keyword. Poking and recommending users are not functionalities of Twitter: the former is a specific action of Facebook and the latter of LinkedIn, but they appeared anyway in the classification. Very likely the two actions refer to bots that provide cross-platform functionalities, starting from Twitter, which are however out of the scope of this article.

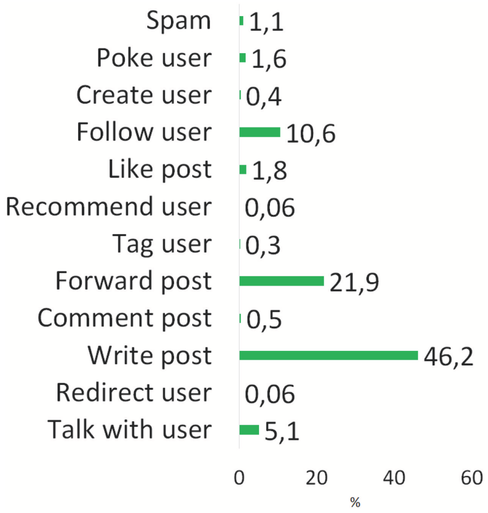


Figure 2 Labels of repositories.

The goal of this inspection was to enable the automatic labeling of the repositories with action labels by analyzing the keywords found in their descriptions and the informed selection of repositories for manual inspection. The automation was achieved by transforming all keywords (and their variants) into regular expressions that could easily be searched for in the repository descriptions. The results of the classification of all retrieved Twitter repositories is shown in Figure 2. It is evident that the most popular action labels are: *follow user*, *forward post*, *write post*, and *talk with user*. Interestingly, the label *like post* is not as important, while all other actions have very little support in the dataset.

Regarding the identified action types it is important to note that these actions per se do not yet imply any specific abuse or harm. However, if misused they all may be used to perform abuses. The labels in Figure 2, in particular, stem from a prior empirical analysis where we found at least one abuse per label.

3.3 Final Dataset

With the goal of maximizing the likelihood of being able to identify recurrent patterns in the code while guaranteeing diversity in the dataset, we applied the following criteria for the selection of the code repositories to be included in the study:

- Selection of repositories that use as main programming language *Python*.
- Exclusion of all those repositories that, after manual inspection, were considered *out of scope*, e.g., because not implementing bots at all or because not implementing any direct communication with other platform users.
- For each of the four most used actions (according to the preliminary analysis), selection of 5 repositories randomly chosen from the respective *best* repositories, according to the ranking provided by GitHub. The respective scores account for the number of forks, clones, likes, and similar. This choice assures that there is a minimum number of popular projects in the dataset for which we can assume to find code patterns with reasonable support.
- For each of the four most used actions, random selection of 5 repositories from the *rest* of the respective retrieved repositories. This choice aims to include also examples that are less popular, while still useful for our analysis.
- Selection of 10 repositories randomly chosen from the *best* repositories we could *not classify* automatically in the preliminary analysis. This assures the presence of a-priori unknown but popular repositories.
- Selection of 10 repositories randomly chosen from the *rest* of the *not classified* repositories, again to assure a representative selection of generic, a-priori unknown repositories.

The final dataset selected for analysis in this article is thus composed of 60 GitHub Twitter bot repositories whose main programming language is Python. In average, each repository comes with 3 files (standard deviation of 2.02) with an average number of lines of code of 192, an average size of 21.39 KBytes, an average number of subscribers of 3, and an average number of watchers of 13. The most popular repository (twitter-contest-bot, https://github.com/kurozael/twitter-contest-bot) has been forked 99 times, the least popular one (tweetpix, https://github.com/mseri/tweetpix) 0 times, with an average of 5 forks per project across all projects included.

4 Identification and Analysis of Abusive Code Patterns

4.1 Method

To the best of our knowledge, this is the first study that aims to understand and categorize how state-of-the-art social bots implement their interactions with human actors and whether it is possible to identify explicit intentions for the behaviors the bots exhibit in their social communications; no results exist yet. Starting from the dataset described above, we thus perform a manual, systematic review [11] of the code retrieved from GitHub, in order to (i) identify which code passages implement *interactions* with humans, (ii) categorize the concrete *actions* the bots use in their interactions (similar to [10]), and (iii) identify different implementation *patterns* for each categorized action, along with respective *examples* (green field analysis). Actions and patterns were first categorized by one of the authors and then agreed on and integrated by both authors. The result is a conceptual framework composed of actions, patterns and code examples that may allow us to infer the intention behind possible abuses; we did not specifically take into account code comments, as they were generally not used consistently throughout the repositories.

4.2 Actions: How Bots Participate in Communications

The preliminary analysis of our dataset in Section 3.2 has shown that Twitter bot developers promise almost all of the typical actions also human users can perform when using social networks in general. In order to understand which actions are really implemented in the repositories forming our dataset, and how, we reviewed all code files of the dataset manually looking for relevant code fragments. For a code fragment to qualify as *action* it either has to (i) implement some form of interaction by the bot with other users or (ii) implement application logic that manages content or user data fetched from the social network. An action thus represents a self-contained interaction of the bot with content and/or users.

The result of this iteration is summarized in Table 1, which describes the 9 actions that represent a consistent synthesis of all examples identified by this exercise using a Twitter-specific terminology. As expected, the bottom-up analysis brought up a set of typical *social network actions*, declined in Twitter terminology. Bots *follow* other users, *like* their tweets, *tweet* own content, *mention* other users in their tweets, or *retweet* tweets by others. Inside private chat rooms, they also *chat* with other users using instant messages. This result is in line with the actions identified in [10].

But there is more. Looking at the code of the bots further produced three *internal actions* that support their social network actions: bots heavily *search* Twitter for users or tweets, in order to harness accounts and content to work with; they intentionally *pause* or delay their interactions, in order to impersonate users; and they may *store* content they retrieve from the network for

Table 1 Synthesis of online communication actions implemented by Twitter bots

Action	Description	Visible
Follow	Follow users to establish social relationships	yes
Like	Like tweets by other users to endorse them	yes
Tweet	Post a new tweet to communicate content	yes
Mention	Mention other users in tweets using @ to attract attention	yes
Retweet	Re-post tweets by other users to endorse them	yes
Chat	Send direct messages to users to converse with them	yes
Search	Search users or tweets using names, keywords, hashtags, ids or similar or by navigating social network relationships (e.g., friends of friends, followers of friends, friends of followers, followers of followers)	no
Pause	Pause the conversation flow of the bot	no
Store	Store content retrieved from the social network for later use	no

later use. These internal actions are observed only in the code of the bots and would not be identifiable by looking at the externally visible communications of the bots only, as done by most literature on the topic. Later in this article, we will see that also internal actions that are not immediately visible to users may lead to abuses and harm.

4.3 Code Patterns: How Bots Implement Their Actions

Focusing on the code fragments considered relevant as communication actions, a second iteration of the code review aimed at synthesizing all examples of action implementations into a taxonomy of recurrent code patterns that explains how actions are implemented in practice. For a code fragment to qualify as a *pattern*, two requirements must be met: (i) it must be possible to abstract the fragment and to associate it to at least one action, and (ii) it must recur at least two times in different code repositories. A pattern thus represents the intended function of a set of instructions, not their syntactic manifestation in the code.

Even accounting for different names of identifiers in the code, without this type of semantic abstraction it would be necessary to perform a purely syntactic similarity search. However, given the diversity of the repositories and developers that characterize our dataset, only unlikely it would have been possible to spot two fragments that are syntactically equivalent.

For instance, it is possible to interact with the Twitter API using direct, low-level HTTP requests, or one can use a dedicated API wrapper library, such as (in order of use in our dataset): tweepy (http://www.tweepy.org), Twitter libraries (https://bit.ly/2Gg3WJC), TwitterAPI (https://bit.ly/2UwSZri), Twython (https://bit.ly/2aOjCnT), or own, proprietary libraries. Similarly, there are different options for the automatic generation of text for tweets or instant messages, such as nltk (https://www.nltk.org/) or seq2seq (https://bit.ly/2Ry2FQt). Patterns abstract away from these implementation choices and aim to capture the essence of what the developer wanted to implement.

The result of this analysis is reported in Table 2, which names and summarizes the identified patterns. These 28 patterns concisely represent the different interpretations of the 9 actions as implemented in the approximately 140 code examples collected and analyzed.

Table 2 Taxonomy of code patterns used for the implementation of actions

Action	Pattern	Description
Follow	*Unconditioned follow*	Follow users without checking suitability of users, usernames or content shared
	Whitelist-based follow	Follow only users whose attributes or tweets match some element of a given whitelist
	Blacklist-based follow	Don't follow users whose attributes or tweets satisfy one or more criteria specified in a blacklist
	Phantom follow	Follow users and unfollow them when condition is satisfied, e.g, a limit of friends reached or being followed back
Like	*Unconditioned like*	Like tweets without checking suitability of content, user or username
	Whitelist-based like	Like only tweets by users whose attributes or content match some element of a whitelist
	Blacklist-based like	Don't like tweets whose attributes or users match an element of a blacklist
	Mass like	Aggressively like tweets of given users
Tweet	*Fixed-content tweet*	The content of the tweet is taken from a fixed, static collection of predefined messages
	AI-generated tweet	The text of the tweet is automatically generated using AI/NLP tools
	Trusted source tweet	The content of the tweet is taken from a source that can be considered trusted
	Tweet with opt-in	Tweets are sent only to people who ask to interact with the bot, sending it a message or mentioning it in a tweet

Table 2 Continued

Action	Pattern	Description
Mention	*Unconditioned mention*	Mention other users without checking suitability of username or content shared
	Targeted mention	Classify users on the basis of their tweets and mention them in targeted messages
	Whitelist-based mention	Mention only users whose attributes match a whitelist
	Blacklist-based mention	Don't mention users whose attributes match a blacklist
Retweet	*Unconditioned retweet*	Retweet without checking content or username for suitability
	Whitelist-based retweet	Retweet content only from users whose attributes match some element of a whitelist
	Blacklist-based retweet	Don't retweet tweets whose attributes or users satisfy some condition expressed in a blacklist
	Mass retweet	Aggressively retweet multiple tweets by selected users
Chat	*Unconditioned chat*	Send direct messages to users without checking suitability
	Talk with opt-in	Reply only to messages sent to the bot (passive behavior)
	AI-generated chat	Generate messages using AI/NLP tools
	Fixed-content chat	Take message from a fixed list of predefined phrases
	Targeted chat	Classify users based on their tweets or attributes and target message accordingly
Search	*User search*	Search user account by name, keyword, id or similar
	Tweet search	Search tweets by keyword or hashtag
	Trend search	Search trending topics or hashtags by location
Pause	*Mimic human*	Use pauses in instant messages to deliver human-like conversation experience to other humans
	Satisfy API constraints	Use as short as possible pauses just to avoid being blocked by API usage limitations
Store	*Store persistently*	Store retrieved content or user information for later use

Example 1. Let us inspect the following two lines of code to understand the logic of the proposed patterns:

```
for tweet in tweepy.Cursor(api.search, q=QUERY).items():
     tweet.user.follow()
```

The code uses the tweepy library to interact with Twitter and implements two actions: *search* and *follow*. The *search* action is reified by the *search user* pattern (which exact feature is used for the search is unknown as the content of QUERY is not visible). The *follow* action is reified by the *Unconditioned follow* pattern, as line 2 follows all users without applying any filter on the users. ◁

Example 2. The following three lines of code show a concrete implementation of the *blacklist-based mention* pattern:

```
def mentions(count, max_seconds_ago, id_blacklist) :
   return [mention for mention in api.mentions_timeline
   (count=count)
      if not mention.id in id_blacklist ]
```

The code defines a function that returns all the ids of the users that have mentioned the bot in prior tweets (expressing some form of interest in the bot) and whose ids are not contained in the list of banned ids id_blacklist. ◁

Incidentally, these examples are also representative of two recurrent types of patterns across multiple actions: for all those actions that somehow endorse a user or a tweet (follow, like, mention, retweet), the analysis identified patterns that do so *unconditionally* or that do so by first checking if the involved user is *blacklisted* or not. Independently of these examples, the analysis also identified other recurrent types of patterns for these actions that endorse users only if they are *whitelisted*. Other notable patterns implement massively repeated actions like *mass like* and *mass retweet*, which aggressively endorse content by given users, or specially targeted actions like *targeted mention* and *targeted chat*, which instead carefully select the users to interact with (e.g., suicide candidates) and send them particularly tailored messages (e.g., to point to help and prevent suicide).

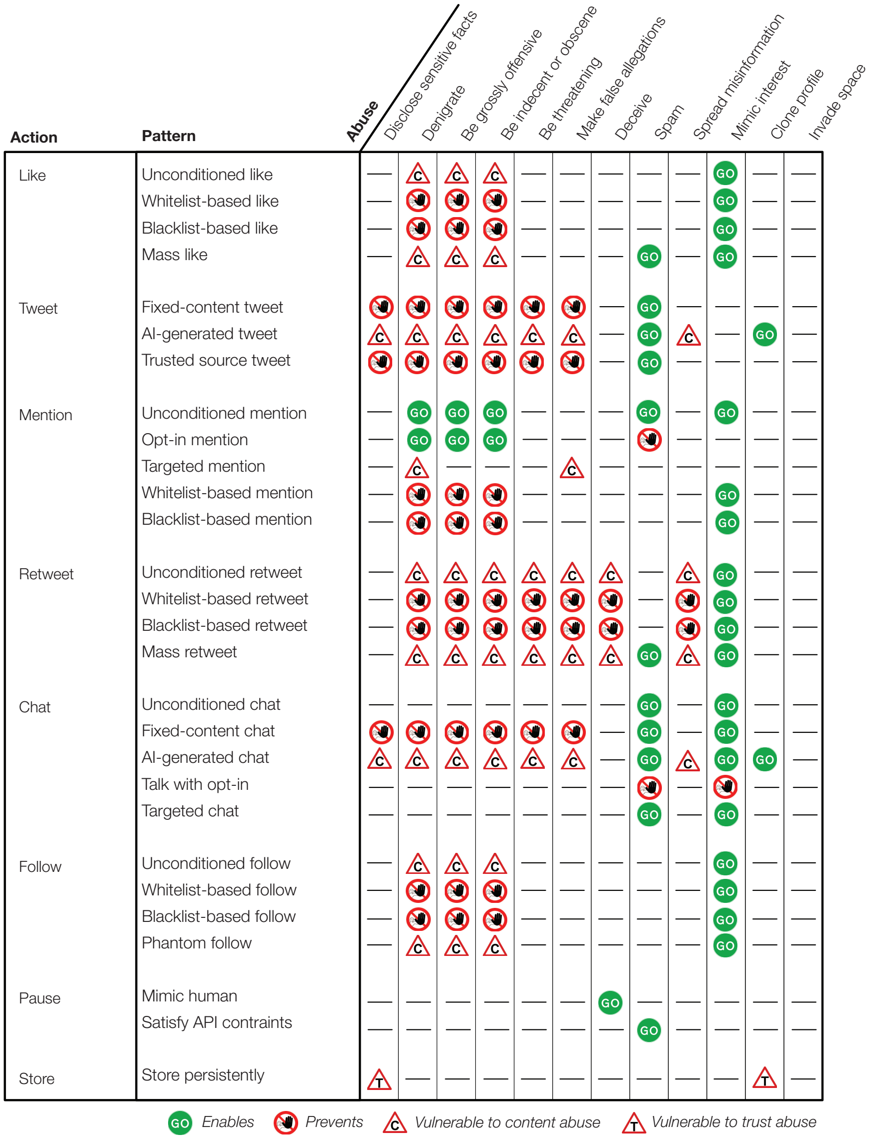
4.4 Effects of Actions: Assessing Potential Harmfulness

Considering again the unconditioned, blacklist and whitelist patterns, it is important to acknowledge that they implement different levels of sensibility of risk as perceived by the developer. Unconditionally retweeting content expresses either a high level of trust in the users who produce the retweeted content, or it expresses a lack of awareness of the risks that retweeting for example offensive, denigrating or obscene content may have on the reputation of the bot owner. Either way, it becomes evident that each pattern may have a different effect or impact on the users a bot interacts with.

In our prior work [10], we identified 12 major types of abuses bots have committed in the past (see the top-right list in Figure 3) and that have produced harm (remember Section 2.1). The first half of these abuses are legally prosecutable in most democratic countries (see, for example, New Zealand's Harmful Digital Communications Act of 2015 [15]). The typical question that remains unanswered when harm occurs is *why* the respective abuse was committed.

Some bots intentionally create spam messages, e.g., to influence political elections [3], but then there are bots like Microsoft's AI-based chatbot Tay that got trained by multiple colluding users, e.g., to offend Jews (http://bit.ly/2DCdqM4). Evidently, the bot was not ready for orchestrated attacks. From the outside, it is generally not possible to tell why abuse happens. This article provides a look inside the code that drives bots, and attempts a technical explanation for some of the abuses. In fact, patterns may have the following *effects*:

- *Enable an abuse*, if they implement logic that by design performs an abuse. For example, the *phantom follow* pattern enables mimicking interest for opportunistic reasons, e.g., to be followed back by users, or the *mass retweet* pattern enables artificially boosting the visibility of a user.
- *Prevent an abuse*, if they implement logic that aims to prevent the bot from performing an abuse. The *blacklist-based follow* pattern, for instance, prevents interactions with unwanted users, while the *tweet with opt-in* pattern prevents spamming users not interested in the bot.
- *Be vulnerable to content abuse*, if they implement interactions with users and/or content that may be inappropriate. The *unconditioned follow* pattern, for instance, causes the bot to follow users that may have inappropriate usernames or spread inappropriate content. The vulnerability may arise when endorsing content or users or when feeding user-provided content to AI algorithms without proper prior checks (see the example of Tay).
- *Be vulnerable to trust abuse*, if they forward, store or analyze content retrieved from users. The *store persistently* pattern is an example of this threat. A user sharing, for instance, sensitive information via personal messages is vulnerable if stored data are leaked to unintended audiences.



Action	Pattern	Disclose sensitive facts	Denigrate	Be grossly offensive	Be indecent or obscene	Be threatening	Make false allegations	Deceive	Spam	Spread misinformation	Mimic interest	Clone profile	Invade space
Like	Unconditioned like	—	C	C	C	—	—	—	—	—	GO	—	—
	Whitelist-based like	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
	Blacklist-based like	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
	Mass like	—	C	C	C	—	—	—	GO	—	GO	—	—
Tweet	Fixed-content tweet	⛔	⛔	⛔	⛔	⛔	⛔	—	GO	—	—	—	—
	AI-generated tweet	C	C	C	C	C	C	—	GO	C	—	GO	—
	Trusted source tweet	⛔	⛔	⛔	⛔	⛔	⛔	—	GO	—	—	—	—
Mention	Unconditioned mention	—	GO	GO	GO	—	—	—	GO	—	GO	—	—
	Opt-in mention	—	GO	GO	GO	—	—	—	⛔	—	—	—	—
	Targeted mention	—	C	—	—	—	C	—	—	—	—	—	—
	Whitelist-based mention	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
	Blacklist-based mention	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
Retweet	Unconditioned retweet	—	C	C	C	C	C	C	—	C	GO	—	—
	Whitelist-based retweet	—	⛔	⛔	⛔	⛔	⛔	⛔	—	⛔	GO	—	—
	Blacklist-based retweet	—	⛔	⛔	⛔	⛔	⛔	⛔	—	⛔	GO	—	—
	Mass retweet	—	C	C	C	C	C	C	GO	C	GO	—	—
Chat	Unconditioned chat	—	—	—	—	—	—	—	GO	—	GO	—	—
	Fixed-content chat	⛔	⛔	⛔	⛔	⛔	⛔	—	GO	—	GO	—	—
	AI-generated chat	C	C	C	C	C	C	—	GO	C	GO	GO	—
	Talk with opt-in	—	—	—	—	—	—	—	⛔	—	⛔	—	—
	Targeted chat	—	—	—	—	—	—	—	GO	—	GO	—	—
Follow	Unconditioned follow	—	C	C	C	—	—	—	—	—	GO	—	—
	Whitelist-based follow	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
	Blacklist-based follow	—	⛔	⛔	⛔	—	—	—	—	—	GO	—	—
	Phantom follow	—	C	C	C	—	—	—	—	—	GO	—	—
Pause	Mimic human	—	—	—	—	—	—	GO	—	—	—	—	—
	Satisfy API contraints	—	—	—	—	—	—	—	GO	—	—	—	—
Store	Store persistently	T	—	—	—	—	—	—	—	—	—	T	—

Abuse
GO *Enables* ⛔ *Prevents* C *Vulnerable to content abuse* T *Vulnerable to trust abuse*

Figure 3 Potential effects of actions and patterns on the users in online communications: patterns either enable, prevent or are vulnerable to abuses. For example, following an account with a denigrating or offending username may perpetuate and endorse the denigration or offense.

These four effects may translate into human users of the social network getting harmed or not. But harm in this context has two sides: if a user gets harmed through interaction with a bot, this may also affect and possibly harm the owner of the bot himself. If a bot threatens someone or discriminates people, the owner may become subject to legal prosecution. If it leaks private data, it may be suspended by the social network, as this violates the usage policies.

Figure 3 graphically summarizes for the patterns in Table 2 (except the *search* patterns without side-effects) which abuses they may enable, prevent or risk to commit. It is meant to create awareness in bot developers of the effects the code they write may have once their bot is deployed and interacting with people.

It is important to note that the analyzed dataset features only a few bots that implement patterns that aim to prevent abuses, which testifies a generally low awareness of the problem and commitment to mitigate risk by developers. Specifically, only 5 repositories implement blacklist-based patterns, 2 control if the user is verified by a whitelist (implementing multiple patterns), 6 use opt-in verification, 4 use a trusted source for tweets, and 8 use fixed content instead. Finally, we did not find any indication of effects of the identified patterns on the abuse *invade space* (it refers to bots invading spaces, e.g., online discussion groups or social networks, that are not meant for bot participation), as Twitter is generally open to bots.

5 Large-Scale Analysis: Automated Recognition of Code Patterns

While the patterns described above have an informative and educational value on their own, they can also be used as input for the implementation of an automated classifier to systematically search for patterns in bot code repositories. We call this classifier *Pattern Recognizer* (PR).

The implementation of the PR has taken inspiration from the work by Santanu and Atul [13], who proposed a framework for the extraction of generic fragments from code. In their work, they designed a pattern language for C and PL/AS (a variant of PL/1, a programming language developed by IBM), while the underlying approach is applicable to any language. The pattern language is an extension of the target programming language. The extension makes use of wildcards, which are special symbols used to match code elements. Patterns written with this language are transformed into an Abstract Syntax Tree (AST), which is used to build a special finite state automaton (FSA) that, in turn, takes as input the AST of the program of interest. If the FSA reaches a final state, one or more patterns have been recognized.

We have adapted the work by Santanu and Atul to the special case of pattern search in Python-based bot code repositories and implemented the code patterns described in the previous section. We summarize the implementation and report on the application of the PR in the following.

5.1 Pattern Language

The first ingredient of the PR is the pattern language to formally describe the patterns of interest. This language is an extension of the standard Python language. It has all the syntactic elements of Python plus the addition of some wildcards, which are special symbols used to match one or more instances of syntactic elements. For example, the wildcard corresponding to a variable matches every instance of a variable in a program, regardless of the respective name. Ideally, a pattern language is equipped with a wildcard for each syntactic element; in this work, we limit the wildcards to the elements that are needed for the recognition of the code patterns identified in the previous section. The specific wildcards implemented are detailed in Table 3.

Writing effective patterns can be a complex endeavor. To write a pattern, it is good to observe that a pattern is not a monolith but that it can be seen as a combination of building blocks that can be written independently and then combined. For example, retweeting a tweet can be seen as a sequence of the following blocks:

1. Retrieve tweets
2. Loop over the tweets
3. For each tweet, retweet it

Assuming that a program contains a function `retrieve_tweets()` that returns a list of tweets, the retweeting logic above could be written as follows:

```
_VAR_TWEETS_ = retrieve_tweets
_STAT_MULTI_
for _VAR_TWEET_ in _VAR_TWEETS_:
    api.retweet(_VAR_TWEET_)
```

5.2 Pattern Recognition

For the PR to be able to parse and match patterns, it needs a representation of both the program to be scanned and the pattern to be searched for. The

Table 3 Wildcards of Python-based pattern language for abuse code pattern specification

Name	Symbol	Description
Generic Variable	_VAR_	matches any variable
Generic Multi Variable	_VAR_MULTI_	matches any variable, with any number of attribute accesses and function calls
Generic Function Definition	_FUN_()	matches any function definition
Generic Function Call	_FUN_()	matches any function call
Generic Assignment	_ASSIGN_	matches any assignment
Generic Number	_NUM_GT_n_LT_m	matches any number. The tokens "_GT_" and "_LT_" are optional and specify whether the number must be greater than n or smaller than m
Generic Argument	_ARG_	matches a generic argument in a function call
Generic Arguments	_ARGS_	matches any number of arguments in a function call
Generic Statement	_STAT_	matches a generic statement
Generic Multi Statement	_STAT_MULTI_	matches any number of consecutive generic statements

representation used here for this purpose is an *Abstract Syntax Tree* (AST), i.e., a tree where each node is an abstract syntactic unit of the program. To build the AST of the program and the pattern we use a built-in module of Python called "ast" (https://docs.python.org/3/library/ast.html) that, given a Python program in input, produces the respective AST in output. In particular, the output is a list of nodes, where each node has a list, the so-called `body`, that contains all its children. Figure 4 illustrates an example of AST for a simple function definition.

Of course, the AST provided by the Python module does not have nodes representing the wildcards of our pattern language. We thus constructed a new module on top the built-in library, in order represent also wildcards.

The automaton used to identify patterns is a standard *Finite State Automaton* (FSA) where each state represents a Python statement[1]. The automaton take as input the AST of the pattern to be searched for and the AST of the

[1] Supported statements are: generic expressions, conditional statements, `while`, `for`, `continue`, `return`, `break`, `delete`, asynchronous function definitions, function definitions, class definitions, asynchronous `for` loops, `try/except` statements, asynchronous `with` statements, `with` statements, `lambda` statements, `yield` statements, `yield from` statements.

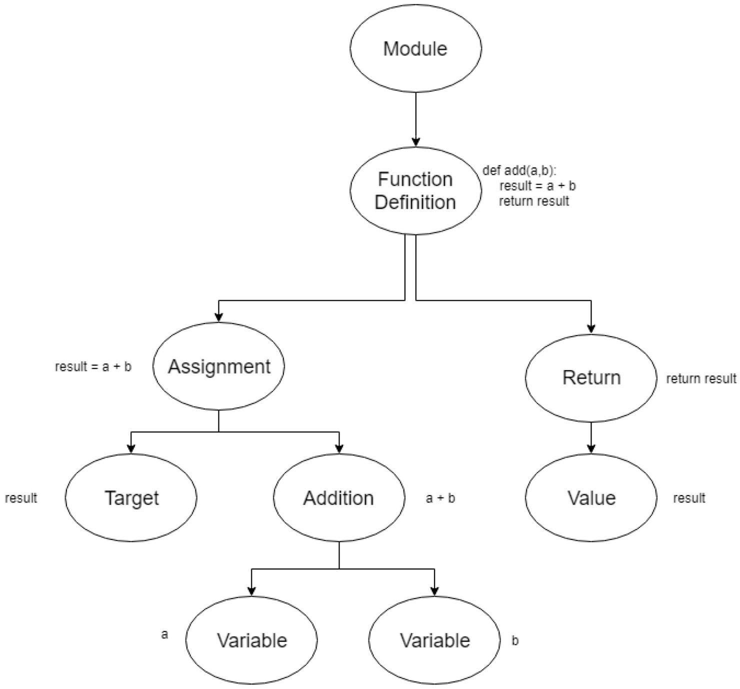


Figure 4 Example of an AST for the definition of a generic function `add(a,b)`

program to be scanned. While the FSA goes from one state to another of the pattern's AST, the automaton scans the AST of the scanned program. In other words, every time it performs a transition, it passes to the next statement of the program. Each state of the automaton is a final one, and it outputs a result. All of them but one output a negative result, if the comparison between the statement represented by the state and the current statement of the scanned program gives a negative output. The comparison is done splitting the statements into their basic syntactic units, and comparing them. For "standard" units, the comparison is a strict check for equality, while when the comparison is between a wildcard and a "standard" unit, the check is done following the rule of wildcards explained above. So, if the comparison between statements has a positive output, the automaton goes to the next state. The only state that can output a positive result is the last one. If the automaton reaches the last state, and the last comparison is positive, then it outputs the piece or the pieces of code that match the given pattern.

The source code of the PR can be found on Github at `[URL hidden for review]`.

5.3 Analysis

We now describe the implementation of a set of code patterns and apply the pattern recognizer to the full dataset of Python bot code repositories retrieved in Section 3, going beyond the 60 repositories analyzed manually.

5.3.1 Pattern implementation

Before talking about the actual implementation of patterns for pattern search, it is important to note that the problem is very complex and not solvable in general. In Section 4.3 we intentionally provided a conceptual, abstract representation of the abusive code patterns we encountered in our analysis, in order to make that knowledge reusable across code repositories and programming languages. In order to search for patterns, it is instead necessary to provide a representation of the patterns that is able to capture programming language-specific syntax and solutions.

The analysis of our manually labeled dataset has shown that for each pattern there is a large variety of ways the patterns may be implemented – too many varieties to be manageable if we wanted to capture them all. The following analysis is thus based on patterns that we considered simple enough and affordable (in terms of implementation time needed) and, hence, represents a best effort analysis. Out of the patterns described in Section 4.3, we implemented the following patterns: unconditioned / blacklist-based / whitelist-based / phantom *follow*, unconditioned / blacklist-based / whitelist-based / mass *retweet*, unconditioned / blacklist-based / whitelist-based / mass *like*, *satisfy API constraints*, and *mimic human*. Refer to the Appendix for the details of the implementation in the developed pattern language.

The implementation of the patterns, is based on three techniques: observation of dataset, creative thinking, and reification of abstract Python libraries. *Observing the dataset* means manually reading code projects (the 60 projects we derived the patterns from), look at how the patterns are implemented, and infer specific descriptions in the pattern language. The challenge is specifying patterns that are as abstract as possible, while still able to capture specific cases. *Creative thinking* means reflecting on how else develops could achieve the realization of the patterns, in terms of basic actions, and describing these actions in the pattern language. The challenge here is going beyond what is visible in the code and bringing in own programming expertise. *Reifying*

Python library calls is needed to take into account that the interactions with the Twitter API may be performed using different libraries, whose use however does not affect the meaning of patterns. The APIs and libraries contemplated are:

- Tweepy (http://docs.tweepy.org/en/v3.5.0/api.html)
- Python Twitter API (https://github.com/sixohsix/twitter/tree/master)
- Python-Twitter (https://python-twitter.readthedocs.io/en/latest/twitter.html)
- TwitterAPI (https://github.com/geduldig/TwitterAPI)
- Twython (https://twython.readthedocs.io/en/latest/api.html)

5.3.2 Dataset and metrics

The study presented in this section is based on two different datasets: the first dataset, which we call the *gold dataset*, is the one used to manually derive the patterns and is composed if 60 projects; the second dataset, which we call the *test dataset* is a dataset of 786 projects randomly chosen from the results of the bot code repository search on Github described in Section 3.

The *metrics* used to evaluate the state of the art are:

- True Positives, TP = # correctly identified patterns
- False Positives, FP = # incorrectly identified patterns
- False Negatives, FN = # incorrectly not identified patterns
- Precision, P = $\frac{TP}{TP + FP}$
- Recall, R = $\frac{TP}{TP + FN}$
- Number of projects implementing a pattern, NP = $\frac{TP + FN}{Total\# \; of \; projects}$

TPs, FPs and FNs were assessed manually by going through the source code of the projects looking for code patterns.

5.3.3 Results

Table 4 reports the performance of the pattern recognizer using the gold dataset from which we derived patterns discussed in the previous sections of this article. In line with our manual analysis, the most implemented operational patterns (patterns involving tweeting, following or liking) are the *unconditioned* patterns. They simply look for tweets or users using keywords and they retweet or like the tweets, or follow the users or the authors of the tweets. Few bots implement filters (*blacklist* or *whitelist*),

Table 4 Results of automated pattern recognition using the gold dataset. Percentages of NP refer to the size of the gold dataset (60 repositories)

Pattern	TP	FN	FP	P	R	NP
Unconditioned Follow	14	4	0	100%	88%	16 (26.7%)
Blacklist-based Follow	2	1	0	100%	67%	3 (5%)
Whitelist-based Follow	1	1	0	100%	50%	2 (3.3%)
Phantom Follow	1	1	0	100%	50%	2 (3.3%)
Unconditioned Retweet	11	3	0	100%	79%	14 (23.3%)
Blacklist-based Retweet	1	3	1	50%	25%	4 (6.7%)
Whitelist-based Retweet	0	0	0	0	0	0
Mass Retweet	1	0	0	100%	100%	1 (1.7%)
Unconditioned Like	10	1	0	100%	91%	11 (18.3%)
Blacklist-based Like	0	0	0	0	0	0
Whitelist-based Like	0	0	0	0	0	0
Mass Like	1	0	0	100%	100%	1 (1.7%)
Satisfy API Constraints	16	6	0	100%	72.7%	22 (36.7%)
Mimic humans	4	6	0	100%	40%	10 (16.7%)

phantom follow or *mass* patterns. For four of the implemented patterns, the PR was not able to retrieve any specific instance, although we know that at least one representative of the pattern must be present in the gold dataset. Excluding these patterns (for which it is not possible to calculate P/R), the macro-averages [16] of the precision/recall values are 95.5% and 69.3%, respectively. Globally (micro-average), the precision is 98.4% and the recall is 70.5%.

Table 5 reports the results of the same experiment with the full test dataset (786 bot repositories). The first result that can be noticed looking at the numbers is that the number of projects that implement any of the patterns is generally low. The patterns which have the highest presence are the temporal ones (*satisfy API constraints* and *mimic human*). This tells us that being limited by Twitter's API constraints is one of the most important concerns of bots.

Regarding the operational patterns, they are not as widespread as expected. Reading the code of bots to validate the results, we discovered a plausible reason, which is that most of the bots are designed to post simple tweets on Twitter, while bots which make use of retweet, follow, and like are a minority. This is somewhat surprising. Among the operational patterns, the majority of the bots use again the *unconditioned* patterns. Looking at

Table 5 Results from test dataset. In round parenthesis there is the quantity of projects corresponding to the percentage

Pattern	TP	FN	FP	P	R	NP
Unconditioned Follow	46	24	6	88.4%	65.7%	50 (6.4%)
Blacklist-based Follow	4	2	0	100%	67%	6 (0.7%)
Whitelist-based Follow	1	1	0	100%	50%	2 (0.25%)
Phantom Follow	4	18	0	100%	18%	22 (2.8%)
Unconditioned Retweet	50	15	5	90.9%	76.9%	65 (8.3%)
Blacklist-based Retweet	1	1	1	50%	50%	2 (0.25%)
Whitelist-based Retweet	0	2	3	0	0	2 (0.25%)
Mass Retweet	1	0	0	100%	100%	1 (0.13%)
Unconditioned Like	32	12	5	86.4%	72%	44 (5.6%)
Blacklist-based Like	13	1	1	93%	93%	14 (1.8%)
Whitelist-based Like	0	2	0	0	0	2 (0.25%)
Mass Like	0	0	0	0	0	0
Satisfy API Constraints	198	16	0	100%	92.5%	214 (27.2%)
Mimic human	84	11	0	100%	88.4%	95 (12.1%)

blacklist-based and *whitelist-based* patterns, we can see that the bots which implement them are a minority. Only the *blacklist-based like* is implemented by more than the 1% of the bots. This result provides evidence that there is still a lack of awareness among developers that unconditioned patterns may expose the bot to unwanted and unexpected consequences or that prevention is simply not a priority.

A small but significant part of the bots do implement the *phantom follow* pattern. This was instead expected, as Twitter poses severe limits on the amount of users that can be followed. It is therefore a good rule for a bot to unfollow users who don't follow back, in order to be free to follow other users, hoping that they will eventually follow back (which is the typical goal when following other users). The *mass* pattern is the least supported one. Almost no one is interested in targeting specific users and liking or retweeting all of his/her tweets, which is positive.

Considering the performance of the PR on the test dataset, the small number of samples doesn't allow us to drive final conclusions about the quality of the system, but still the precision and recall obtained in this test are promising and deserve a discussion. Looking at Table 5, the precision is generally very high, except for *blacklist-based retweet*: the macro-average of the pattern-specific precision values is 77.6%. This means that when the system outputs a positive result for a certain pattern, it is very likely that that pattern is really implemented in the repository. The micro-average of the precision is instead 95.4%, which is in line with the micro-precision obtained also with the gold dataset.

The recall of the system has no precise trend. The macro-average across patterns is 59.5%, which is lower as with the gold dataset (expected). So, while the recall can be high for some of the patterns, like *blacklist-based like* or *mimic human*, on average the system is likely to return some false positive when searching for patterns. The micro-average for the recall is 80.5%, which is again in line with the performance of the PR on the gold dataset. The decrease of the macro-average can be explained by the fact that some patterns, e.g., the *whitelist-based* ones, perform poorly and, hence, lower the average. Globally, the micro-average remains however high.

6 Related Works

As already hinted at in the introduction, the topic of social bots has so far been approached mostly from the perspective of telling humans and bots apart, that is, with the intention of *detecting* bots. The work that is most closely related to this aspect is Botometer, formerly known as BotOrNot [7, 8], an online tool that computes a bot-likelihood score for Twitter accounts and allows one to tell bots and genuine user accounts apart. The tool builds on more than 1000 features among network, user, friends, temporal, content and sentiment features, and uses a random forest classifier for each subset of features. The training data used is based on bot accounts collected in prior work by Lee et al. [12], who used Twitter honeypots to lure bots and collected about 36000 candidate bot accounts following or messaging their honeypot accounts.

Some works go further and turn their focus to *specific types* of social bots and, thereby, harms. For instance, Ratkiewicz et al. [14] studied the phenomenon of *astroturfing*, i.e., political campaigns that aim to fake social support from people for a cause, and showed that bots play a major role in astroturfing activities in Twitter. Cresci et al. [6] specifically focused on the problem of *fake followers*. They constructed a dataset of human accounts (manually and by invitation of friends) and bought fake followers from online services like http://fastfollowerz.com. The work compares two types of automatic classifiers, classifiers based on expert-defined rules and feature-based classifiers (machine learning), and shows (i) that fake followers can indeed be spotted and (ii) that black-box, feature-based classifiers perform better than white-box, rule-based classifiers. In addition, the work also produced a publicly available, labeled dataset that can be used for research purposes. Varol et al. [17] propose a bottom-up approach to the identification of bots with similar online behavior. The classifier used is the one adopted by Botometer, while the dataset used also included a manually annotated collection of Twitter accounts. After classifying accounts into bot or not, the authors further clustered the bot accounts into three types of bots: *spammers*, *self promoters*, and accounts that *post content from applications*. Chu et al. [5] coined the term *cyborg* to refer to bot-assisted humans in social networks and used a manually labeled dataset of 6000 randomly sampled Twitter accounts and a random forest classifier plus entropy measures to classify accounts into bots, cyborgs and humans.

In terms of *datasets* analyzed for bot detection, Beskow and Carley [2] propose four tiers of data for the classification of Twitter accounts: single tweet text (tier 0), account + one tweet (1), account + full timeline (2), and account + timeline + friends timelines (3). The assumption is that bot detection is achieved using feature-based classification or AI algorithms. In fact, with their tool bot-hunter, the authors study different machine learning techniques for tier-1 datasets. Differently from these classification-based approaches, Cao et al. [4] describe SybilRank, a tool for the detection of sybil accounts (bots) in social networks by analyzing the social graph (of Facebook, in the specific study). The study in this article focuses on a different type of dataset, i.e., code, to understand the internals of bots, not their externally visible behavior or traces.

Little or no work has been done so far on the analysis of *harms* and *abuses*, as proposed in this article. Perhaps the work by Varol et al. [17] can be seen as an ethical alarm: it estimates that between 9% and 15% of all accounts in Twitter are likely automated accounts and shows that bots are able to apply sophisticated communication tactics, distinguishing between humans and bots.

7 Discussion and Outlook

This article proposes an original perspective on bots for online communication: instead of looking at messages or network activity, which is the typical practice in literature, it analyzes the code that produces them. To the best of our knowledge, this is the first study of its kind in this area.

7.1 Summary of Findings

The study contributes to the state of the art in a twofold fashion: a manual, qualitative study characterizing the problem and an automated, quantitative study backing the qualitative study with suitable evidence.

The qualitative study identified *31 patterns* that implement different variants of 9 communication actions from a dataset of 60 GitHub Twitter bot repositories (approximately 75–80 hours of manual code inspection). Then, it discussed the *effects* the patterns may have at runtime and provides a systematic mapping of patterns to potential abuses as a reference for developers. The hope is that the awareness of potential abuses may help prevent bots from causing psychological, legal, economic, social and democratic harm.

This allows us to finally come back to the *why* question, which is still open, and attempt a technical interpretation of why abuses may happen. In fact, it seems plausible to assume: (i) that bots that explicitly enable abuses *intentionally* try to do harm or at least accept the possibility to do so; (ii) bots that are vulnerable to content abuse by other users may *unintentionally* cause harm, while still being responsible for the content they endorse or spread; and (iii) bots that are vulnerable to trust abuse, if they leak data, may do so intentionally (e.g., it they sell data) or unintentionally (e.g., if intruders steel data). Regarding this last case, we did not find any hint for intentional leaks in our dataset. These ethical aspects are particularly relevant to web engineering if we consider that many understand bots as the apps of tomorrow. As a possible usage scenario, social network providers that host third-party bot code (e.g., Facebook) could use these patterns to implement early warning systems to prevent harm.

The quantitative study carried out using the automated pattern recognizer is based on the implementation of 14 out of the 31 identified patterns and was applied to 786 Python bot code repositories taken from GitHub. The key findings are, first of all, that it is possible to automatically identify code patterns inside generic code repositories (very good precision and recall values) and that: (i) the key concern by bot developers is tricking Twitter's API constraints to maximize bot productivity, and (ii) preventing vulnerabilities using blacklists or whitelists is not a major concern to developers, which may expose the bots to content abuse. Whether this lack of preventive mechanisms is intentional or not remains an open question; in any case, it expresses some level of negligence.

7.2 Limitations

The findings of this article stem from a careful, manual systematic code review. They are thus limited by nature. As for the *internal validity*, the study suffers of course from the limited size of the dataset; perhaps more repositories would have allowed us to identify more patterns. Also, the open-source nature of the projects may provide a limited view on the possible patterns, as developers of intentionally malicious bots may not share their code. The focus on Python was needed to keep the dataset manageable. The careful, randomized selection of repositories aimed to increase internal validity. As for the *external validity*, different programming languages and communication platforms may behave differently. However, the core of the actions and patterns proposed in this article are similar in other platforms and programming languages. These may differ in platform-specific functionalities (e.g., poking a user in Facebook), but the abstractions of this article make the actions and patterns portable.

As for the pattern recognizer, which we consider a first attempt at automating the manual work above, we would like to note that recognizing patterns is based on two distinct ingredients: the formal specification of the patterns and their interpretation and matching against the source code of the bot repositories. This latter is based on the prior work by Santanu and Atul [13] and works satisfactorily. The tricky part are the patterns, whose general implementation, as explained earlier, is very hard and represents a best effort. This is of course a limitation that affects the recall of the pattern recognizer. Inspecting more pattern examples and implementing more patterns would help increase recall, yet this would still represent a lower bound for the recall, as you can always do more and better.

Regarding the precision of the pattern recognizer, the key current limitation is that is works only at a syntactic level and neglects possible semantics of the code. If a pattern, for instance, specifically looks for variables that contain tweets, the current implementation matches just variables of any name and meaning; it is not able to discriminate variables based on their name (e.g., “tweets” vs. “users”) or based on how variables have been initialized (e.g., by assigning values returned from a specific API call that produces tweets). This limitation may produce false positives in the search.

It is important to note that the quantitative study and the implementation of patterns are both based on a manual analysis of code and are, therefore, at most able to represent a subset of the patterns that may actually present abusive behaviors. All findings are thus to be read as lower bounds of the phenomenon. For instance, our dataset did not include bots making use of deep learning or reinforcement learning to evolve behavior over time. These types of bots thus require a dedicated analysis.

7.3 Outlook

In order to assist developers in finding good pattern representations and to partly automate the process of pattern authoring, we are considering using machine learning algorithms to find a pattern representations from code samples and to find representations of lower-level, building blocks that can again be combined into patterns. There are studies in literature on the construction of a representation of source code using machine learning. For instance, Alon et al. [1] built a system that builds a vector representation of source code that can also give meaning to the code. We are considering building on that effort in our future work, and we are considering taking into account code semantics to improve the precision of the pattern recognizer. Of course, also expanding the horizon of the investigation beyond Twitter and Python represents a likely future development.

Appendix: Implementation of Potentially Abusive Code Patterns

In the following paragraphs, the implementation of the patterns used in the study described in Section 5 is described, including details and examples in pattern language and Python.

Mimic Human Behaviour

This pattern has been elaborated using the assumption that a real user usually interposes long pauses between actions on a social network. When using the adjective “long”, we intend an order of magnitude of minutes. So, to mimic the behaviour of real people, a bot must stop his activity from time to time for a long period of time.

Example of description in pattern language

```
sleep(random._VAR_MULTI_._FUN_(_NUM_GT_3000, _NUM_GT_3000)
```

This description is intended to look for calls of the *sleep* function, which take as argument an amount of time in seconds that is a random number between two numbers greater than 3000.

Example of implementation in Python

```
if wait_time > 0:
  print("Choosing time between %d and %d -
  waiting %d seconds before action" %
  (min_time, max_time, wait_time))
  time.sleep(wait_time)
```

Satisfy API Constraints

If a bot doesn't want to pretend to be human, but it wants only to do the maximum number of actions it can, without being limited by Twitter, it will continuously check if it has reached the limits imposed by Twitter, and it will pause the script in these cases. An alternative and less effective way for the bot to avoid to be limited is to sleep a small amount of time after each action it does.

Example of description in pattern language

```
if _VAR_1 < _VAR_2:
    _STAT_MULTI_
    _VAR_MULTI_sleep(_ARGS_)
```

The pattern has the following meaning: search a piece of code that is a generic condition that tests if a variable is less than another variable and whose body is a call to the sleep function. The reasoning behind the elaboration of this pattern is that if the bot is checking if a variable is less than other, it is probably checking if the number of actions it has done is less than the permitted ones.

Example of implementation in Python

```
if ratelimit[2] < min_ratelimit:
    time.sleep(200)
```

Unconditioned Follow, Like, and Retweet

These three patterns are grouped together because they have been elaborated using the same reasoning. The same applies to blacklist-based patterns and mass patterns. These patterns are the simplest ones: if the bot doesn't use a blacklist or a whitelist to filter users and tweets, it is doing an unconditioned action. So, the system first checks blacklist-based and whitelist-based patterns, and, if it doesn't find matches for them, it uses a description that includes only the action of retweeting, liking, or following. If it finds a match for those base patterns, it can conclude that the bot is performing unconditioned actions.

Example of description in pattern language

```
_VAR_MULTI_.create_friendship(_ARGS_)
```

In this example, we use the description to look for pieces of code where there is a simple call to a function of the API that allows the bot to follow a Twitter account.

Example of implementation in Python

```
try:
    api.create_friendship(my_id, follow)
except:
    print('Already requested %s' % current_screen_name)
```

Blacklist-based Follow, Retweet, and Like

Performing blacklist-based actions basically means filtering out the users to follow and the tweets to like or retweet.

Example of description in pattern language

```
_VAR_USERS_ = set(_VAR_1) - set(_VAR_2)
_VAR_MULTI_
for _VAR_USER_ in _VAR_USERS_:
    _STAT_MULTI_
    _VAR_MULTI_.create_friendship(_ARGS_)
```

In this example, the filter is implemented using a set difference. The idea behind this description is that if the bot is taking out something from the set of users to follow, probably it is filtering out unwanted users, so it is blacklisting users.

Example of implementation in Python

```
RTUsers = set(RTUsers) - set(blacklisted_users)
...
for f in RTUsers:
try:
    api.create_friendship(f)
```

Phantom Follow

Performing a Phantom Follow means removing the friendship relation with a user once it is no longer needed. Usually, bots follow users to get their attention, obtain their friendship, and thereby raise their numbers of followers. When users have followed back the bot, there is no need to follow them any longer, on the contrary, the bot has to unfollow them to be free to follow new users. So, from time to time, the bot takes the list of its friends, and unfollows a part or all of them.

Example of description in pattern language

```
_VAR_FRIENDS_ = _VAR_MULTI_.friends\_ids(_ARGS_)
_STAT_MULTI_
for _VAR_FRIEND_ in _VAR_FRIENDS_:
    _STAT_MULTI_
    _VAR_MULTI_.destroy_friendship(_VAR_USER_)
```

In this example, we look for a piece of code where there is the collection of the friends of the bot, followed by a loop over friends where, for each friend, the bot unfollows him/her.

Example of implementation in Python

```
my_following = api.friends_ids(my_id)
...
unfollowed = 0
count = 0
for following in my_following:
    ...
    try:
        if skipMutuals:
            if following not in my_followers:
                api.destroy_friendship(my_id,
                                        following)
        else:
            api.destroy_friendship(my_id,
                                    following)
            unfollowed+=1
            count+=1
```

Mass Like and Retweet

To massively like or retweet a post, a bot has to collect a bunch of tweets from a specific user, and, then, attempt to like or retweet all of them. The Twitter API offers a method to collect the tweets of specific users, and, in turn, libraries offer the same methods.

Example of description in pattern language

```
_VAR_TWEETS_ = _VAR_MULTI_.get_user_tweets(_ARGS_)
_STAT_MULTI_
for _VAR_TWEET_ in _VAR_TWEETS_:
    _STAT_MULTI_
    _VAR_MULTI_.retweet(_ARGS_)
```

In this example, we look for pieces of code where there is, first, the collection of the tweets of the user, and, then, a loop over the tweets where there is the attempt to retweet the tweets.

Example of implementation in Python

```
tweets = TwitterAPI.get_user_tweets(twitter_object,
         screen_name,paginate_older_attribute)
...
if (len(tweets) > 0):
 for tweet in tweets:
        try:
 TwitterAPI.bot_retweet(twitter_object,
                     str(tweet[‘id_str’]))
 TwitterAPI.bot_like(twitter_object,
                     str(tweet[‘id_str’]))
```

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[2] David M Beskow and Kathleen M Carley. Bot-hunter: A tiered approach to detecting & characterizing automated activity on twitter. In *SBP-BRiMS 2018*, 2018.

[3] Alessandro Bessi and Emilio Ferrara. Social bots distort the 2016 us presidential election online discussion. *First Monday*, 21(11), 2016.

[4] Qiang Cao, Michael Sirivianos, Xiaowei Yang, and Tiago Pregueiro. Aiding the detection of fake accounts in large scale social online services. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 15–15, 2012.

[5] Zi Chu, Steven Gianvecchio, Haining Wang, and Sushil Jajodia. Detecting automation of twitter accounts: Are you a human, bot, or cyborg? *IEEE Transactions on Dependable and Secure Computing*, 9(6):811–824, 2012.

[6] Stefano Cresci, Roberto Di Pietro, Marinella Petrocchi, Angelo Spognardi, and Maurizio Tesconi. Fame for sale: efficient detection of fake twitter followers. *Decision Support Systems*, 80:56–71, 2015.

[7] Clayton Allen Davis, Onur Varol, Emilio Ferrara, Alessandro Flammini, and Filippo Menczer. Botornot: A system to evaluate social bots. In *WWW 2016*, pages 273–274, 2016.

[8] Emilio Ferrara, Onur Varol, Clayton Davis, Filippo Menczer, and Alessandro Flammini. The rise of social bots. *Communications of the ACM*, 59(7):96–104, 2016.

[9] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[10] hidden.

[11] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[12] Kyumin Lee, Brian David Eoff, and James Caverlee. Seven months with the devils: A long-term study of content polluters on twitter. In *ICWSM*, pages 185–192, 2011.

[13] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[14] Jacob Ratkiewicz, Michael Conover, Mark R Meiss, Bruno Gonçalves, Alessandro Flammini, and Filippo Menczer. Detecting and tracking political abuse in social media. In *ICWSM*, pages 297–304, 2011.

[15] The Parliament of New Zealand. Harmful Digital Communications Act 2015. Public Act 2015 No 63, 2015.

[16] Grigorios Tsoumakas and Ioannis Vlahavas. Random k-labelsets: An ensemble method for multilabel classification. In *European conference on machine learning*, pages 406–417. Springer, 2007.

[17] Onur Varol, Emilio Ferrara, Clayton A Davis, Filippo Menczer, and Alessandro Flammini. Online human-bot interactions: Detection, estimation, and characterization. *arXiv preprint arXiv:1703.03107*, 2017.