Analysing Question and Answer Websites on the Example of Stack Exchange

David Stutz

November 24, 2014

Contents

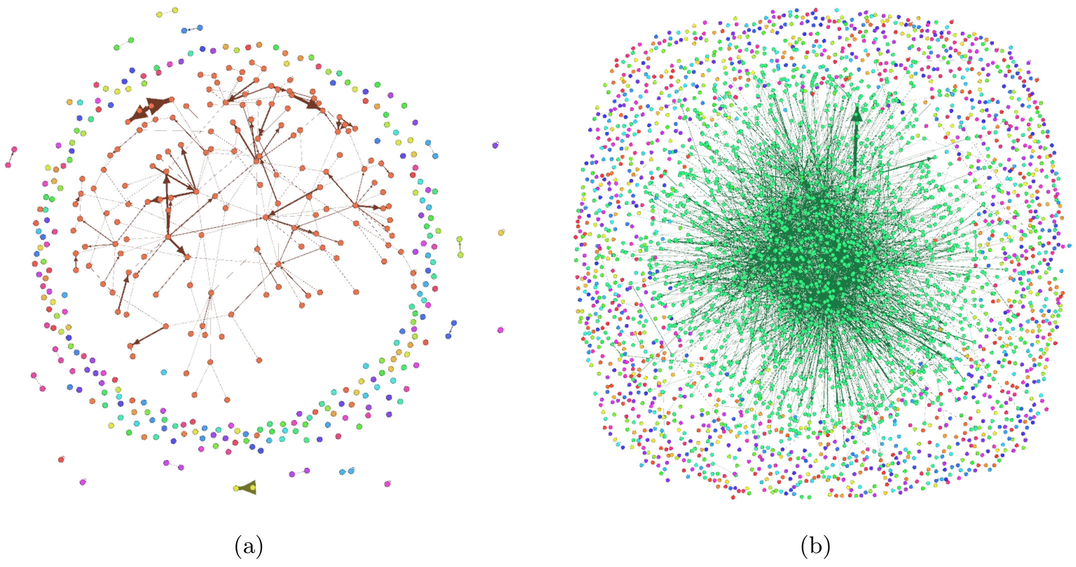

Figure 1: The datascience network (a) and the crypto network (b) where the nodes are colored according to the weak connected components.

1 Introduction

This document provides a brief analysis of two social networks taken from the question and answer website Stack Exchange[1]: the social network corresponding to `datascience.stackexchange.com` and the social network corresponding to `crypto.stackexchange.com`. Both networks are based on the latest data dump from Stack Exchange[2]. Based on posts (questions) and comments (answers), a directed graph was created where nodes represent users who either created a post or commented on a post, and a directed edge from user A to user B indicates that A commented on a post created by B.

The networks were created using PHP[3] scripts which are provided together with this documents. Due to processing constraints, only small networks could be generated. Network Analysis was mainly done using Gephi[4]. In this case, the datascience network has 417 nodes while the crypto network has 3426 nodes. Both networks – nodes colored according to their weak connected component – are shown in figure 1.

[1]Stack Exchange provides a network of question and answer websites on several topics: `http://stackexchange.com/`.

[2]The data can be found here: `https://archive.org/details/stackexchange`.

[3]See `http://php.net/`.

[4]See `http://gephi.github.io/`.

2 The datascience network

As mentioned above, the datascience network consists of 417 nodes and 216 edges. Note that the number of edges is less than the number of nodes, indicating that a bigger part of the network consists of isolated nodes. This means that plenty of posts (questions) have no comments and are therefore unresolved. This fact may be explained by several observations. First, the website is officially in beta status, see `datascience.stackexchange.com`. Second, data science is a relatively young field of study and therefore experts in this field are missing.

As shown in figure 1, the network was first segmented into weak connected components. There are 244 weak connected components and the average degree is 0.518. These facts underline the observation that most questions have not been answered. In addition, when examining figure 1 closely, there is only a weak community structure observable. This hypothesis is supported by the average clustering coefficient of 0.053. Furthermore, only 5 triangles are found within the network. Nevertheless we can identify several experts – or hubs – within the network, that is some users have an out-degree of 4 or more meaning that they have answered several questions.

As conclusion, it has been shown that the datascience network does not have a strong community structure yet. However, hubs seem to be emerging and the network may as well develop a stronger community structure with a rising number of experts in the field of data science.

3 The crypto network

The crypto network consists of 3426 nodes and 3293 edges. Again, the number of edges is less than the number of nodes, however, we can expect to have less isolated nodes when compared to the datascience network. Compared to the field of data science, cryptography is several years older and therefore there are more experts. This suggests that more posts have been answered. When examining figure 1, this hypothesis can be confirmed: the average degree is 0.961 and the average clustering coefficient is 0.388 with a total of 1828 triangles. This suggests that the crypto network has a stronger community structure with several big hubs: some users have answered 100 or more posts. This also suggests that the information exchange about the topic of cryptography is much more intensive when compared to the datascience network.

In conclusion, it can be observed that the community structure gets stronger when the research area gets older and more experts are able and willing to answer questions. Nevertheless, the network structure of the crypto network has similarities to the datascience network in that there is a “core” of several experts answering most of the questions, and a periphery where a lot of posts are unanswered.

4 Conclusion

After having analyzed the datascience and the crypto network, it can be concluded that both networks are similar in structure. The difference can easily be explained using the age of the research areas: while data science is a relatively young area of research, the field of cryptography is much older and therefore shows a stronger community structure with higher number of experts (hubs).

A PHP scripts

The `UserPostCommentGraph` class will create a `.gml` file given the latest data dump of `crypto.stackexchange.com` (can be similarly used to generate the `gml` file for `datascience.stackexchange.com`). The data can be found here: `https://archive.org/details/stackexchange`.

```
<?php

include 'Graph.php';

if (!function_exists('assert')) {
    /**
     * Assert fuctions.
     *
     * @param bool $bool
     * @param string $message
     * @throws Exception
     */
    function assert($bool, $message = NULL) {
        if (!$bool) {
            throw new Exception($message);
        }
    }
}

/**
 * Creates a graph where each user is a node and nodes are connected in the
 * following way:
 *
 *  user A -> user B iff A comments a post from B
 *
 * Edge weights correspond to the comment score. Additional attributes saved
 * for users are:
 *
 *  - Reputation
 *  - DisplayName
 *  - Views
 *  - UpVotes
 *  - DownVotes
 *
 * The unique identifier of every user is its respective id.
 *
 * Usage:
 *
 *  $postsFile = 'Posts.xml';
 *  $commentsFile = 'Comments.xml';
 *  $usersFile = 'Users.xml';
 *
 *  $userPostCommentGraph = new UserPostCommentGraph($postsFile,
     $commentsFile, $usersFile);
 *  // Creates the graph ...
 *  $userPostCommentGraph->process();
 *  // Gets the created graph, see the Graph class.
 *  $graph = $userPostCommentGraph->getGraph();
 *
 * @author David Stutz
 */
class UserPostCommentGraph {

    /**
```

```
         * The created graph.
         */
        protected $_graph;

        /**
         * XML of posts as string.
         */
        protected $_postsXML;

        /**
         * XML of comments as string.
         */
        protected $_commentsXML;

        /**
         * XML of users as string.
         */
        protected $_usersXML;

        /**
         * Construct a graph by providing Posts.xml, Comments.xml, Users.xml.
         *
         * @param string $postsFile
         * @param string $commentsFile
         * @param string $usersFile
         */
        public function __construct($postsFile, $commentsFile, $usersFile) {
            assert(file_exists($postsFile));
            assert(file_exists($commentsFile));
            assert(file_exists($usersFile));

            $postsXML = new XMLReader();
            $postsXML->open($postsFile);
            $postsXML->setParserProperty(XMLReader::VALIDATE, true);
            assert($postsXML->isValid());

            $commentsXML = new XMLReader();
            $commentsXML->open($commentsFile);
            $commentsXML->setParserProperty(XMLReader::VALIDATE, true);
            assert($commentsXML->isValid());

            $usersXML = new XMLReader();
            $usersXML->open($usersFile);
            $usersXML->setParserProperty(XMLReader::VALIDATE, true);
            assert($usersXML->isValid());

            $this->_postsXML = file_get_contents($postsFile);
            $this->_commentsXML = file_get_contents($commentsFile);
            $this->_usersXML = file_get_contents($usersFile);

            $this->_graph = NULL;
        }

        /**
         * Processes the XML files and sets up the actual graph.
         *
         * Use
         *
         *  $userPostCommentGraph->getGraph()
         *
```

```
     * to get the created graph (see Graph class).
     */
    public function process() {
        $postsXMLElement = new SimpleXMLElement($this->_postsXML);
        $commentsXMLElement = new SimpleXMLElement($this->_commentsXML);
        $usersXMLElement = new SimpleXMLElement($this->_usersXML);

        $this->_graph = new Graph();
        foreach ($postsXMLElement->row as $post) {
            $userPostId = (string) $post['OwnerUserId'];
            $postId = (string) $post['Id'];

            $userPost = $usersXMLElement->xpath('/users/row[@Id="' .
                $userPostId . '"]');

            if (sizeof($userPost) > 0) {
                $userPost = $userPost[0];

                if (!$this->_graph->nodeExists($userPostId)) {
                    $this->_graph->addNode($userPostId, array(
                        'Reputation' => (string) $userPost['Reputation'],
                        'DisplayName' => (string) $userPost['DisplayName'],
                        'Views' => (string) $userPost['Views'],
                        'UpVotes' => (string) $userPost['UpVotes'],
                        'DownVotes' => (string) $userPost['DownVotes'],
                    ));
                }

                foreach ($commentsXMLElement->xpath('/comments/row[@PostId
                    ="' . $postId . '"]') as $comment) {

                    $userCommentId  = (string) $comment['UserId'];
                    $userComment = $usersXMLElement->xpath('/users/row[@Id
                        ="' . $userCommentId . '"]');

                    if (sizeof($userComment) > 0) {
                        $userComment = $userComment[0];

                        if (!$this->_graph->nodeExists($userCommentId)) {
                            $this->_graph->addNode($userCommentId, array(
                                'Reputation' => (string) $userComment['
                                    Reputation'],
                                'DisplayName' => (string) $userComment['
                                    DisplayName'],
                                'Views' => (string) $userComment['Views'],
                                'UpVotes' => (string) $userComment['UpVotes
                                    '],
                                'DownVotes' => (string) $userComment['
                                    DownVotes'],
                            ));
                        }

                        assert(TRUE === $this->_graph->nodeExists(
                            $userCommentId));
                        assert(TRUE === $this->_graph->nodeExists(
                            $userPostId));

                        $this->_graph->addEdge($userCommentId, $userPostId,
                            (int) $comment['Score']);
                    }
                }
            }
        }
    }

    /**
     * Get the created graph.
     *
     * @return Graph
     */
    public function getGraph() {
        assert($this->_graph !== NULL);

        return $this->_graph;
    }
}

$userPostCommentGraph = new UserPostCommentGraph('../data/crypto.
    stackexchange.com/Posts.xml', '../data/crypto.stackexchange.com/Comments.
    xml', '../data/crypto.stackexchange.com/Users.xml');
$userPostCommentGraph->process();

ini_set('memory_limit', '2048M');
ini_set('max_execution_time', 600);

header('Content-type: text/gml');
echo $userPostCommentGraph->getGraph()->exportAsGML();
```

The `Graph` class is used by the `UserPostCommentGraph` to generate the `.gml` files.

```
<?php

if (!function_exists('assert')) {
    /**
     * Assert fuctions.
     *
     * @param bool $bool
     * @param string $message
     * @throws Exception
     */
    function assert($bool, $message = NULL) {
        if (!$bool) {
            throw new Exception($message);
        }
    }
}

/**
 * Class Graph represents an undirected or directed weighted graph. The
    edges are
 * stored in the form of an adjacency matrix. Each node can have arbitrary
 * attributes. Unweighted graphs have weight = 1 per default.
 *
 * @uathor David Stutz
 */
class Graph {

    /**
     * Array of nodes with corresponding attributes.
     */
    protected $_nodes;
    /**
     * Adjacency matrix.
     */
    protected $_edges;

    /**
     * Directed graph flag.
     */
    protected $_directed;

    /**
     * Construct a graph by an array of nodes, edges and set the graph
         directed
     * or undirected (this cannot be changed afterwards).
     *
     * @param array $nodes
     * @param array $edges
     * @param bool $directed
     */
    public function __construct($nodes = array(), $edges = array(),
        $directed = TRUE) {
        $this->_nodes = $nodes;
        $this->_edges = $edges;

        assert($directed === TRUE OR $directed === FALSE);
        $this->_directed = $directed;
    }

    /**
     * Add an directed edge $idA -> $idB, or the corresponding undirected
         edge
     * witht he given weight (default is 1 for unweighted graphs).
     *
     * @param mixed $idA
     * @param mixed $idB
     * @param double $weight
     */
    public function addEdge($idA, $idB, $weight = 1) {

        assert(!is_array($weight) AND !is_object($weight));

        assert(FALSE !== array_key_exists($idA, $this->_edges));
        assert(FALSE !== array_key_exists($idB, $this->_edges[$idA]));
        assert(FALSE !== array_key_exists($idB, $this->_edges));
        assert(sizeof($this->_edges) == $this->numNodes());
        assert(sizeof($this->_edges[$idA]) == $this->numNodes());
        assert(sizeof($this->_edges[$idB]) == $this->numNodes());

        $this->_edges[$idA][$idB] = (double) $weight;

        if ($this->_directed === FALSE) {
            assert(FALSE !== array_key_exists($idA, $this->_edges[$idB]));
            $this->_edges[$idB][$idA] = (double) $weight;
        }
    }

    /**
     * Add a node with the given id and array of attributes.
     *
     * @param mixed $id
     * @param array $array
     */
    public function addNode($id, $array) {

        assert(FALSE === array_key_exists($id, $this->_nodes));
        assert(FALSE === array_key_exists($id, $this->_edges));

        $this->_nodes[$id] = $array;
        $this->_edges[$id] = array();

        foreach ($this->_nodes as $nodeId => $array) {
            assert(FALSE !== array_key_exists($nodeId, $this->_edges));
            assert(FALSE !== array_key_exists($id, $this->_edges));

            assert(FALSE === array_key_exists($id, $this->_edges[$nodeId]));
            assert(FALSE === array_key_exists($id, $this->_edges[$nodeId]));

            $this->_edges[$nodeId][$id] = 0;
            assert(sizeof($this->_edges[$nodeId]) == $this->numNodes());

            $this->_edges[$id][$nodeId] = 0;
        }

        assert(count($this->_edges, 0) == $this->numNodes());
    }

    /**
     * Get the number of nodes.
     *
     * @return int
     */
    public function numNodes() {
        return count($this->_nodes, 0);
    }

    /**
     * Check whether a node with the given id extists.
     *
     * @param mixed $id
     * @return bool
     */
    public function nodeExists($id) {
        return array_key_exists($id, $this->_nodes);
    }

    /**
     * Export graph as GML file.
     *
     * @return string
     */
    public function exportasGML() {
        $return = 'graph [' . "\n";
        $return .= "\t" . 'directed ' . ($this->_directed === TRUE ? '1':
            '0') . "\n";

        foreach ($this->_nodes as $id => $array) {
            $return .= "\t" . 'node [' . "\n"
                    . "\t\t" . 'id ' . $id . "\n";
            foreach ($array as $key => $value) {
                $escapedValue = $value;
                if (!is_numeric($escapedValue)) {
                    $escapedValue = ’"’ . $escapedValue . ’"’;
                }

                $return .= "\t\t" . $key . ’ ’ . $escapedValue . "\n";
            }

            $return .= "\t" . ’]’ . "\n";
        }

        foreach ($this->_edges as $idA => $array) {
            foreach ($this->_edges[$idA] as $idB => $weight) {
                if ($weight != 0) {
                    $return .= "\t" . ’edge [’ . "\n"
                            . "\t\t" . ’source ’ . $idA . "\n"
                            . "\t\t" . ’target ’ . $idB . "\n"
                            . "\t\t" . ’weight ’ . $weight . "\n"
                            . "\t" . ’]’ . "\n";
                }
            }
        }

        $return .= ’]’;

        return $return;
    }
}
```