Captcha Service

Documentation

Potsdam, February 2017

Supervisor

Prof. Dr. Christoph Meinel,
Christian Bartz

Internet-Technologies and Systems Group

Abstract

The increasing numbers of bots, especially crawlers, within the World Wide Web has been a major concern for several years now. Over the years, different approaches to tackle bots were implemented and tested. Nowadays, one of the most widespread solutions for dealing with bots are Captchas. Originally, the main idea was to give users specific tasks to solve, which bots would be unable to solve. Through distortion and other obstacles, Captchas were improved against algorithmic solutions.

The potential of million online users solving Captchas was quickly noticed. Difficulties in identifying words or objects in images using computers, could be solved using the combined solutions of Captcha users. We implemented our own Captcha Service in order to allow researchers and scientists to get their own datasets labeled using on line users.

Contents

1 Motivation

Machine and deep learning are great tools for tackling more and more complex tasks and problems. However, therefore a great number of data is required in order to train classifiers. Researchers and scientists are lacking the time and capacities to label data on their own, which is often further needed to advance and improve other technologies.

Using required authentication processes for online users, we are able to utilize huge amounts of free labor. Our main goal was building a straightforward service for researchers and scientists to allow precise data labeling. A Captcha service is an ideal solution for data labeling, while at the same time providing additional benefit by providing security from bot attacks. We had to find a trade-off between a convenient, secure solution and a way to label data reliably. A main criteria was also the expandability of the system, allowing to easily integrate further Captcha types for various data.

In order to allow a user friendly usage of the service, a web interface was required for uploading and downloading data.

Last but not least, a simple integration for web services was needed to allow for a fast and widespread usage of the Captcha service.

2 Related Work

In preparation of creating an own Captcha service we searched the Internet for existing ideas and implementations of systems used for data labeling and machine learning purposes. The first popular approach was the Soylent Grid paper, which was published in 2007 [1]. Although there were never any popular implementations of the ideas, the paper provided several different approaches for data labeling. They were mostly based around the idea of object recognition in images, e.g. by clicking on objects or drawing rectangles around it. Other proposals were directed to object recognition, where users had to name objects displayed in certain images.

Another paper, which was published just a year later, dealt with text recognition and described the concept, which was implemented in reCAPTCHA v1 [2]. The system was created by Google in order to solve problems in the "Google Book Project". Two different optical character recognition (OCR) algorithms were used to translate images of scanned pages into digital texts. The solving of Captchas was used to identify words, which could not be deciphered clearly by the OCR algorithms. Because of its detailed documentation and its successful usage this method was ideal to be implemented and therefore the first Captcha type, which is supported by our system.
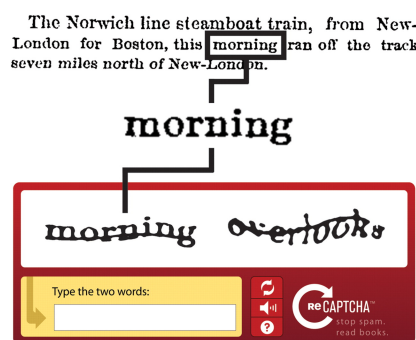

Figure 1: The reCAPTCHA approach explained in one picture

---

[1] http://vision.ucsd.edu/sites/default/files/icv2007.pdf

[2] http://science.sciencemag.org/content/321/5895/1465.full

3 Architecture

The Captcha service uses a Model-View-Controller to distinguish between data and its representation. The views are included in the `views.py`-file and process requests made by the client, the third party web app and the web interface. The data model is represented in the `models.py`-file and manages *CaptchaTokens*, *CaptchaSessions* and the connection to the database. The System is designed for simple expandability and uses inheritance to simplify the introduction of new Captcha types.

3.1 Models (models.py)

The models consist of two main classes, the *CaptchaToken* and *CaptchaSession*. An overview is given in the class diagram in figure 2.

The class *CaptchaToken* represents a single image, that is part for a Captcha, e.g. a single word, that needs to be written down by the user in order to solve the Captcha. The class *CaptchaSession* represents a complete Captcha challenge a user has to solve, e.g. writing down the words shown on all images. Each type of Captcha challenge provided by the service is represented by a subclass of *CaptchaSession* and *CaptchaToken*. Currently two kinds of Captchas, Image-Captchas and TextCaptchas, are supported.

All that needs to be done for implementing a new type of Captcha challenge is to create a new subclass for *CaptchaToken* and *CaptchaSession* and implement specific functionality in these subclasses. Which methods and attributes need to be added in the new subclasses is listed in the "Attributes and Methods implemented in the subclass"-paragraph.

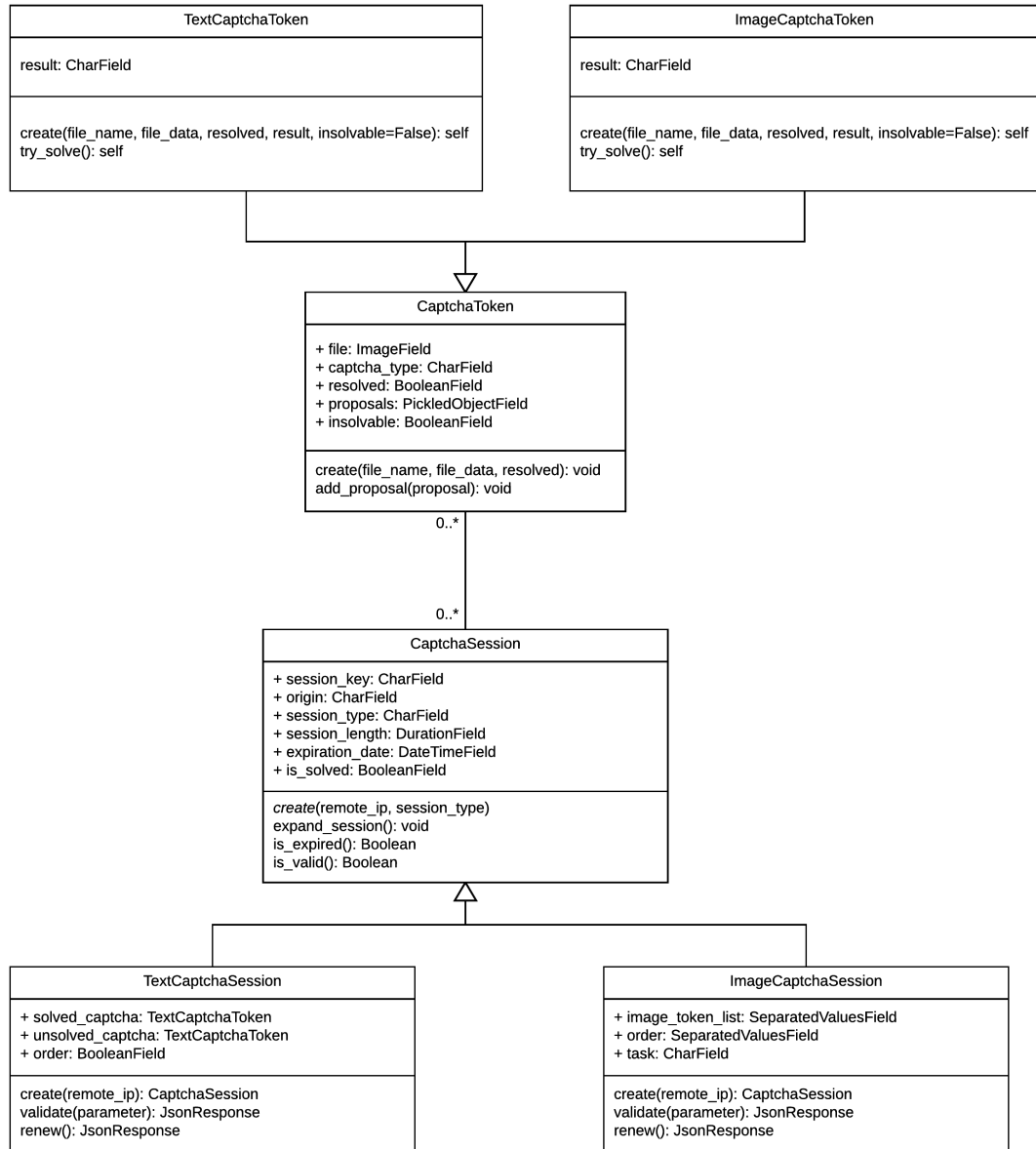All instances of a *CaptchaToken* or *CaptchaSession* are saved in the `db.sqlite3`-Database.

Figure 2: Class diagram representing the classes used for the generation of Captchas. The two main classes *CaptchaToken* and *CaptchaSession* are shown in the center. All other classes inherit from one of the superclasses.

3.1.1 CaptchaToken

The class *CaptchaToken* is the basic unit of the Captcha service. Data, which is supposed to be labeled by the Captcha service is saved as a *CaptchaToken*. Multiple *CaptchaTokens* are combined to a Captcha challenge, when a new *CaptchaSession* is created.

Attributes and Methods implemented in the superclass

Attributes:

- `file`: Image, that is represented by the CaptchaToken.

- `captcha_type`: String, that defines the type of Captcha the token can be used for. Currently "text" for TextCaptchas and "image" for Image-Captchas are supported.

- `resolved`: Boolean, that indicates, if the solution for a *CaptchaToken* is known or not. A `0` means the token is unsolved and a `1` means the Token is solved.

- `proposals`: Dictionary, that stores the possible solutions suggested by users of the Captcha service and how often each solution was suggested.

- `insolvable`: Boolean which indicates, that a token is not solvable by clients of the Captcha service. This value is set to `True`, if there are many proposals for a *CaptchaToken* without one proposal having the majority of the votes. For more information see section 6 (solving algorithm).

Methods:

- `create(file_name, file_data, resolved)`: Responsible for basic configuration, that need to be done for all kinds of tokens, when they are created. Only used for super-calls in the `create()`-method of subclasses.

- `add_proposals(proposal)`: Adds a new suggested solution to the `proposals`-dictionary, or increments the counter for an already suggested proposal.

Attributes and Methods implemented in the subclass

Attributes:

- `result`: Saves the correct solution for a token. Data type differs between different subclasses, e.g. *TextCaptchaToken* saves a string and *ImageCaptcha-Token* saves a boolean.

Methods:

- `create(file_name, file_data, resolved, result, insolvable=False)`: Responsible for configuration of all attributes of the *CaptchaToken*. Returns a *CaptchaToken*.

- `try_solve`: Responsible for finding the correct solution for a *CaptchaToken* based on the values saved in the `proposals`-attribute.

3.1.2 CaptchaSession

Represents an instance of a Captcha challenge, that needs to be solved by a certain client. A *CaptchaSession* consists of multiple *CaptchaTokens*, that are chosen randomly in order to create different challenges dynamically. Each Session corresponds to one of the supported types of *CaptchaTokens*. The tokens chosen are a mix of solved and unsolved tokens, in order to make it possible to validate the session and label new data.

Attributes and Methods implemented in the superclass

Attributes:

- `session_key`: String, that serves as primary key to identify each session.

- `origin`: String, that holds the IP address that requested the Captcha challenge. It is used to match requests made by the client to the corresponding session.

- `session_type`: String, that defines the kind of Captcha challenge, the client has to solve. Currently "text" for TextCaptchas and "image" for ImageCaptchas are supported.

- `session_length`: Timedelta, that stores the amount of time in which a session can be validated. The default value is 30 minutes, which is initialised in the `create()` method of *CaptchaSession*.

- `expiration_date`: Datetime, that stores the date, after which a *CaptchaSession* can not be validated anymore.

- `is_solved`: Boolean, that stores if a *CaptchaSession* was already solved by the client.

Methods:

- `create(remote_ip, session_type)`: Responsible for basic creation of a *CaptchaSession* of the requested type for the given IP address. Only used for super-calls in the `create()`-method of subclasses.

- expand_session(): Responsible for changing the expiration_date of a *CaptchaSession* so that the session is valid for another session_length.

- is_expired(): Responsible for checking, if a session is expired. Returns True when the current date is before the expiration_date.

- is_valid(): Responsible for checking, if a *CaptchaSession* was successfully validated by a client and is not expired. In that case True is returned.

Attributes and Methods implemented in the subclass

Attributes:

Each session needs to store the tokens, which were used for creating the session as well as additional information, that is needed for validating the answer given by the client. This can differ for every Captcha type.

TextCaptchaSession:

- solved_captcha_token: *TextCaptchaToken*, that is already solved and is used as a control word for the session.

- unsolved_captcha_token: *TextCaptchaToken*, that is not solved and shall be identified by the client.

- order: Boolean indicating the order, in which the two tokens are displayed to the client (0 -> solved, unsolved 1 -> unsolved, solved). It is needed to map the answers given by the client to the right tokens.

ImageCaptchaSession:

- image_token_list: List of *ImageCaptchaTokens*, where all tokens used for the session are saved.

- order: List of Booleans, that indicates, which token in the image_token_list is solved (0 -> unsolved, 1-> solved).

- task: String, that saves the object that should be detected in the Captcha challenge, e.g. cat, if cats should be identified.

Methods:

- `create(remote_ip):` Responsible for creating a *CaptchaSession* and returning the created session to the corresponding `view`, and a JsonResponse with the parameters needed to render the Captcha challenge in the front end (e.g. urls of pictures that are shown in the challenge). Depending on the type of the *CaptchaSession*, the function chooses a mix of multiple *CaptchaTokens* with some of them being unsolved and some of them being solved. The unsolved *CaptchaTokens* are data that shall be labeled and the solved *CaptchaTokens* are used for checking, if the answer for the *CaptchaSession* is correct.

- `validate(parameters):` Responsible for validating the solution for a CaptchaSession and returning the created session to the corresponding `view`. The solution suggested by the client is included in the parameters. The method checks, if the suggested solutions matches the solution for the solved *CaptchaTokens*. If the solution given by the client is correct, `proposals` are added for the unsolved *CaptchaTokens* and `try_solve()` is called on these tokens. If the solution is false new *CaptchaTokens* are chosen for the *CaptchaSession* to prevent brute forcing. Returns a JsonResponse with information, whether the session is valid or not and a list of image-URLs that shall be rendered in the session.

- `renew():` Responsible for exchanging the *CaptchaTokens* of a *CaptchaSession*, to create a new challenge or the same session. Returns a JsonResponse of the updated information needed to render the session in the front end.

Session length and expiration

A *CaptchaSession* expires after a certain amount of time. The default time for the `session_length` is 30 minutes. It can be changed in the `create()`-method of the *CaptchaSession* class. The expirations of sessions is done in order to prevent replay attacks. For this reason `is_valid()` always returns false after the expiration date is over. The *CaptchaSession* however remains in the database indefintely, because Django does not provide an elegant method to automatically delete session after expiry. Therefore the Captcha service provides a Django command, which is called `delete_timeouted_sessions` and on execution removes all timed out *CaptchaSessions* in the Database. This can be paired with a cron job so that timed out sessions will be deleted in short intervals.

3.2 Views (views.py)

The views handle POST- and GET-Requests made by the Client, third party web application and the web interface. List of Requests handled by views:

- `request(request)`: GET-Request called by the client, when a new session is requested. Chooses a type for the session randomly, calls create-Method for the session implemented in `models.py` and directs the response of session.create() to the client.
  Corresponding URL: hostname/captcha/request

- `validate(request)`: POST-Request called by the client, when a solution for a Captcha challenge is submitted. Retrieves the the corresponding *CaptchaSession* from the database, calls the validate-Method of the session and directs the response to the client.
  Corresponding URL: hostname/captcha/validate

- `renew(request)`: POST-Request called by the client, when a new challenge for an existing *CaptchaSession* shall be provided. Retrieves the corresponding *CaptchaSession* from the database, calls the renew-Method of the session and directs the response to the client.
  Corresponding URL: hostname/captcha/renew

- `upload(request)`: POST-Request called by the web interface, when new files are uploaded to the Captcha service. Extracts the files from the zip-file and creates tokens corresponding to the provided data.
  Corresponding URL: hostname/captcha/upload

- `download(request)`: GET-Request called by the web interface, to retrieve tokens and solutions from the database. Collects *CaptchaTokens*, that meet the requirements specified in the web interface, compresses them to a zip-file and returns the file.
  Corresponding URL: hostname/captcha/download

- `getTask(request)`: GET-Request called by the web interface, to get all possible tasks for an *ImageCaptchaSession*. Gets all tasks from the database and returns them as a list in a JsonResponse.

Corresponding URL: hostname/captcha/getTask

- `validate_solved_session(request)`: GET-Request called by the third party web application, to check if a *CaptchaSession* was successfully validated by a client. This prevents clients from being able to bypass the Captcha challenge. Corresponding URL: hostname/captcha/validate-solved-session

4 Image Distortion

The images being provided by users for text Captchas, makes it impossible to tell if those images are easy to recognize for bots. Therefore, they might be unsuited for being used as a Captcha token, in there original form. In order to complicate the recognition of the Captcha token, the system uses an image distortion algorithm, which is automatically applied to all uploaded text Captcha tokens.

The image distortion algorithm consists of two steps: the drawing of a horizontal line and a wave transformation.

In the first step it places a horizontal line in the middle of the image, which is colored with the dominant color of the whole image. Afterwards this line will be transformed together with the rest of the image. The algorithm works via the k-means clustering algorithm implemented in the scipy package. The implementation was taken from here.

In the second step, a standard sinus wave is used which will be modified in frequency and amplitude by according modifiers. The frequency as well as the amplitude of the wave, which will be applied, are dependent on the height of the image. Furthermore, the frequency depends on the width of the image so that one wavelength is at least as wide as the image itself. In addition, the frequency and the amplitude are modified by a random value in order make every transformation unique.

```
frequency_modifier = Decimal(random.uniform(0.6, 0.8) *
  (width / height))
frequency = frequency_modifier *
  (Decimal(1) / Decimal((width / 2))) * Decimal(math.pi)

amplitude_modifier = random.uniform(5, 7)
amplitude = (height / amplitude_modifier)
```

The wave is then used to determine an vertical offset for every pixel in the image. Each point then will be shifted either up or down by the offset so that the new y-position is a point on the wave.

```
for x in range(width):
  for y in range(height):
    shift = math.sin(frequency * x) * amplitude
    new_y = y + shift
    if new_y < height and new_y > 0:
      output_data[x, y] = input_data[x, new_y]
```

Every pixel that would be shifted out of bounds will be cut off. Additionally the pixels, which were located at the bottom and the top of the the original image will be stretched out vertically to fill the space, which was emptied due to the transformation.

```
elif shift < 0:
  output_data[x, y] = input_data[x, 0]
elif shift >= 0:
  output_data[x, y] = input_data[x, (height - 1)]
```

5 Integration

5.1 Client Side

The Captcha service integrates seamlessly into existing web applications. It basically works by appending an invisible overlay to the body element of the existing website. The overlay fades in when the user hits the submit button of the captcha protected form. Thereby it does not affect the layout of the existing websites.
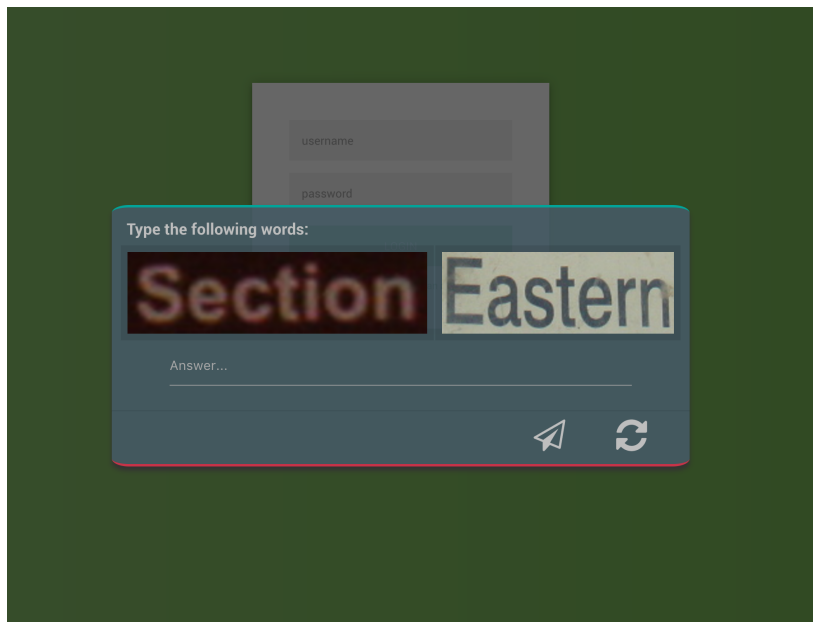
Figure 3: Captcha card contents

The overlay consists of a *Captcha card*, which in turn consists of a *task*, to be solved *Captcha tokens* and *submit* and *refresh* action elements, as shown in fig. 3. As mentioned earlier, we currently support two different Captcha types - namely *text* and *image captchas* - which are randomly delivered to the client.
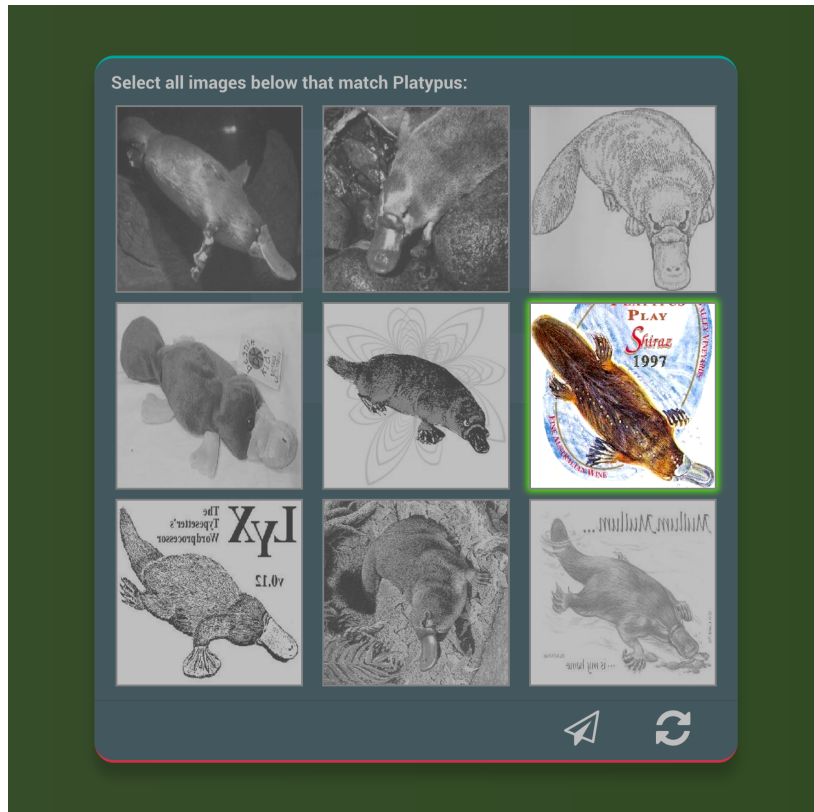
Figure 4: Image captcha session

As soon as the user visits the page, the browser opens a session at the Captcha service via REST API. The browser receives a *session key*, the *Captcha type*, a list of *Captcha tokens* and - in case of an *image Captcha session* - a *task*. The *session key* gets stored in a hidden input field in the Captcha protected form. As a result the *session key* is sent to the web application as soon as the form is submitted, in order to make sure that the client properly solved the Captcha at a later point in time.

Determined by the *Captcha type* the *Captcha tokens* get either rendered as *text Captcha tokens*, as shown in fig. 3, or as *image Captcha grid*, as depicted in fig.4. In case the *session type* is an *image Captcha session*, the task is generated dynamically by inserting the delivered *task* into the HTML, unlike the text Captcha task, which is static.

When the client hits the reload button, the Captcha service is asked via REST API to renew the *session's Captcha tokens*. The response consists of a new list of *tokens* of the same type as before. Subsequently, the old *Captcha tokens* are replaced by the freshly received *Captcha tokens*. The *submit action* triggers the server-side solution validation via REST API. In case the solution was correct the form gets submitted including the *session key*. When the solution is incorrect, the *Captcha card* shakes in order to visually indicate the wrong solution and all newly assigned *Captcha tokens* are inserted. This happens in order to prevent brute force solving. As soon as a wrong solution is submitted, the Captcha service assigns new *Captcha tokens* to the *session* and returns them as response to the client. As a result, you can not restore the old *session tokens* and try to solve it by using all possible inputs.

5.2 Captcha Integration Workflow

Integrating the Captcha service into your existing web application is fairly easy. You have to take the following steps:

1. inject `captcha.min.css` and `captcha.min.js` to your HTML skeleton

2. Make sure the to be protected captcha form has the class *captcha-form* and the corresponding submit button possesses the class *captcha-button*

3. Finally, integrate an additional POST request to our captcha service including the clients *session key* into your existing server-side code, that receives the form data, in order to assure that the user solved the Captcha correctly and did not modify the javascript code.

`captcha.min.js` includes the client-side code, whose behaviour is described in the last section. `captcha.min.css` provides the initial styling. Feel free to modify it, such that it fits your design.

6 Solving Algorithm

In order to provide a value for researchers, who upload data to the Captcha service, the system has to label the uploaded images. Thereby, our solving algorithm determines the label based on the given user inputs.

In case of text Captchas the algorithm needs at least three users, who solved the Captcha correctly. If three or more suggestions match, the image is marked as solved and labeled accordingly. However, the token is identified as unsolvable if there are six or more proposals but no more than two of them match. This approach is relatively similar to the concept of reCAPTCHA. In a published paper [3] it was proven, that three human resolutions are enough to label an image reliably.

The method for labeling image Captchas is similar to the one used for texts. The main difference is the fact that the proposals for these are limited to *true* and *false*, if they are suiting the specified task or not. Therefore, the algorithm checks if at least four resolutions match and also declares a token as unsolvable if more than six suggestions are given but failed to produce four that match. It was decided to raise the bar for labeling a picture from three to four, because it is more likely to falsely select an image due to a wrong click.

SOLVING ALGORITHM

	Text-Captcha	Image-Captcha
solved	3 identical sol.	4 identical sol.
unsolvable	6 unique sol.	6 attempts

Figure 5: Solving algorithm for text and image Captchas based on user input

---

[3] http://science.sciencemag.org/content/321/5895/1465.full

7 Web Interface

The web interface is a simple Django app, where researchers and scientists can register, upload and download images for the Captcha service to use.

7.1 Upload

The upload functionality allows for data to be uploaded for labeling or to feed tasks with already labeled data. The upload of pre-labeled data is necessary to guarantee the functionality of Captchas, thus the validation of a Captcha challenge. First of all, the user chooses for which Captcha type he wants to upload his data and whether the data is already labeled or not. For image Captchas, the user has to choose between an already existing task or he can add a task himself. The task declares which images are to be selected when solving the specific Captcha challenge.

For unsolved Captchas, the user simply has to upload a .zip-file with a folder containing his images. The .zip-file and the included folder need to have the same name. For solved Captchas, the .zip-file must also contain a .txt-file (CSV). The .txt-file needs to containing each image name, followed with `True` or `False` for image Captchas, or their solution word for text Captchas, separated by a comma. If one of the images submitted is not included in the .txt-file it will be ignored. If the .txt-file contains an entry without a corresponding image in the image folder the upload will fail.
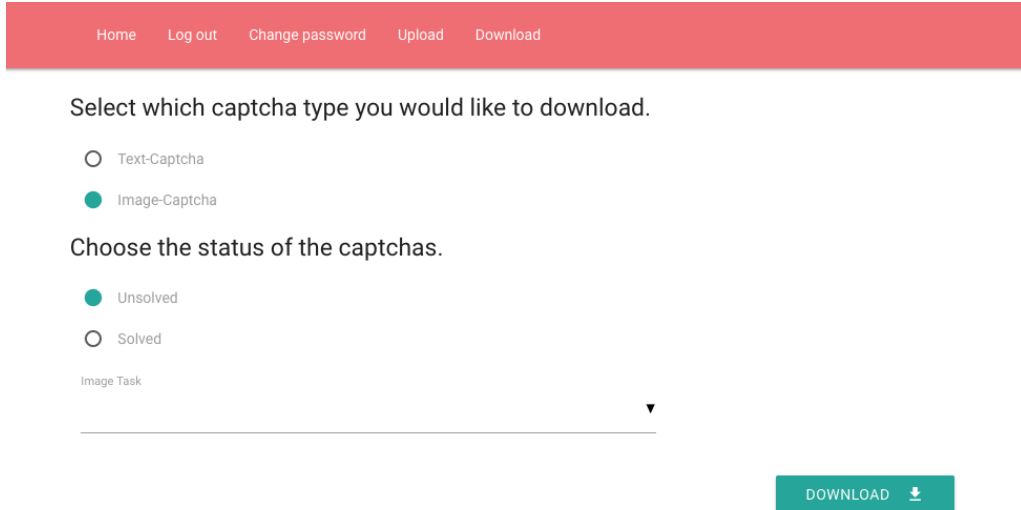
Figure 6: Upload page from web interface

7.2 Download

The download functionality allows for data to be downloaded by choosing the Captcha type. The researcher can specify his task for image Captchas, whereas that is not possible for text Captchas. By pressing the download button, all Captchas matching the chosen type and/or task are downloaded into a .zip-file, together with a .txt-file. For text Captchas, the .txt-file contains all images names followed with their labeled word. For image Captchas, the .txt-file contains all images names followed with `True` or `False`, whether they match their specific task.

Figure 7: Download page from web interface

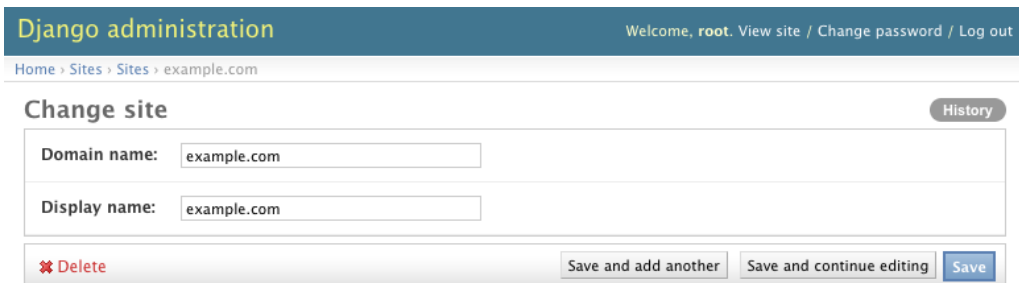## 7.3 User authentication and registration

The user authentication and registration is built using the Django authentication system [4]. Users can register, whereby an Email-address is needed in order to minimize Spam. The authentication Email is sent from an Email-address, which is configured in settings.py. The user receives an activation link, which is active for 7 days. After clicking the activation link, the user is able to login into the web interface.

---

[4]https://docs.djangoproject.com/en/1.10/topics/auth/default/

```
# Email-address to sent authentication Email
REST_SESSION_LOGIN = False
ACCOUNT_AUTHENTICATION_METHOD = 'username'
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_HOST_USER = 'hpicaptcha@gmail.com'
EMAIL_USE_TLS = True

SITE_ID = 1
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_CONFIRM_EMAIL_ON_GET = True
WSGI_APPLICATION = 'wsgi.application'

ACCOUNT_ACTIVATION_DAYS = 7  # One-week activation window
```

Figure 8: Email settings in settings.py

In order for the activation link to work, the site has to be configured within the Django administration. The site has to match the server address, where the Captcha service runs.

Figure 9: Configure site to match server address in the Django administration

8 Evaluation

As proven by reCAPTCHA, the solving algorithm is accurate most of the time. However, there remains a small chance that images are labeled incorrectly. Certainly there is no way to completely eliminate the possibility of inaccurate labeled images. It would be possible to minimise this probability by further increasing the number of needed proposals, but this would slow the labeling process down and therefore result in fewer labeled images over time. The current implementation provides an acceptable trade-off in order to offer satisfying results for researchers, who rely on this service for data labeling.

Another major feature of the CaptchaService is the ability to reliably combat bots and spammers. The security features, which are in place right now will be enough to hold off the majority of all attacks. However one could argue, that it is not sufficient to renew the Captcha after a wrong proposal and validate the solved session of a user. Further security mechanism, such as blocking of ip addresses after consecutive inaccurate solutions could further increase the security of the system. Nevertheless, it will also worsen the user experience of clients that fail to solve Captchas reliably. This could ultimately lead to users not visiting the website again. Therefore the current implementation does not rely on any further security mechanisms and thus offers an optimal trade-off between user experience an security.

9 Future Work

With the thought in mind of building an easy expandable service, the logic consequence would be on focusing on different Captcha types. Another important Captcha type would be the labeling within one image. Therefore, an uploaded image could be split up into nine smaller images and like in our already implemented image Captcha, the user has to select the images with a specific object in it. In the process, key factors such as access for disabled users can be tackled, e.g. by implementing audio Captchas. The accessibility for all users is important to allow for an unrestricted usage of the web services using the Captcha service.

Another aspect would be expanding the web interface. The option of downloading solved and unsolved Captchas should be further able to specialize, e.g. by selecting specific upload times or certain time spans. Since newly uploaded data might extend existing tasks, a researcher might only want to extract his uploaded Captchas. The option for uploading images could need some improvements concerning it's usability, since their are strict requirements for the structure and naming of the uploaded .zip-files. More flexibility in choice about how to upload the data and error feedback would simplify the usage of the captcha service.

Integrating feedback of the labeling progress within a task or uploaded data, is another feature necessary to improve the Captcha service.