Butterfly Annotator — Report

Oscar Davis, Boan Zhu, Yeye Chen, Zhili Tian,
Xingzhi Qian, Siran (Claire) Shen

January 2022

Part 1

Introduction

Machine Learning and Deep Learning are two blooming Computer Science fields that allow to infer, through their different algorithms and techniques, information about an input dataset. This information could be a discrimination of the data into categories (a classification problem), or, for instance, it could also be that of recognizing either certain parts of images or even what they represent entirely. Yet, the methods employed usually employ some form of "statistical" inference: the Machine/Deep Learning model infers what the correct answer is to any input by learning through what it knows to be correct already. But it then requires that the model has input data and output answers that it can assume to be valid (ground truth).

Mapping the input data to correct output answers is called annotating the dataset, and this task can become easily quite tedious. When trying to recognize various parts of an image, the model would require of course the images themselves and regions with labels describing the outlined area of the picture, which would serve as the ground truth. It is quite easily understandable that if one had to annotate hundreds of images on their own, it would be not only cumbersome, but also very time-consuming.

This is where we introduce the software we have been working on: Butterfly Annotator. What we have tried to create is a tool that allows to efficiently annotate large datasets of images, provided a textual description of them by the user. When done, the users can then export their annotations to a JSON file containing a description of each region and its label.

The main goal of the software is to accelerate the workflow of annotators, by making the software as convenient as possible to avoid wasting time on trivia. To do so, we have first decided to implement our software as a web application; this indeed is already quite practical as the user does not need to install anything supplementary and can just use their preferred browser. Furthermore, for a better navigation throughout the website, we have decided to create a "single-page web app." What this means is that instead of loading different pages every time the user wishes to consult a certain part of the website, they do not need to reload the page and can just quickly go to whatever it is they wanted. This was easily realized thanks to the Vue.js framework, which enables this kind of application quite easily, and which just then made HTTP requests to our server written in Python using the Flask

framework. The reason we used the latter is because of the simplicity and speed with which we can write code, with satisfactory performance; we could allow to value productivity in this case, because the back end of this application is not its major constituent.

Furthermore, the software is best fit for datasets that are constituted of the images and descriptions of groups of images that go together. Hence, as an example, if a developer wants to recognize features of butterflies from varied species, they can group the species together, describe them uniformly and then attribute quotes from the description to each region they create. This has been done so that the user does not have to input the same words repeatedly. And, to make this task even quicker, our software automatically suggests relevant sections of descriptions that the user might want to use as labels, based on some straightforward text processing techniques and the inputs the users have previously selected. As for the annotation itself, the focus was mostly on making the tool simple and intuitive to use, and to allow basic functionalities such as undo/redo.

Finally, to further increase productivity, we have decided to allow multiple users to collaborate on different "image banks." What we have called "image banks" are nothing else but datasets of images with their descriptions. Then, using a simple multi-level permission system, users of different authority can manage the bank by allowing others to write, removing users, annotating images, just viewing them, or even deleting the bank itself. However, if one desires to just use the software on their own, they can also run it locally without any issue.

Working on this project as an efficient team required of us to be well organized. To try and do so, we applied a few strategies. First, we had a centralized group chat, where we could all post important messages about whether we should call, when, and why, or if someone encountered an important issue. We then used Microsoft Teams®to actually do the calls, and sometimes share more specific content on it; it also allowed us to produce this report effectively. For the tasks we had to accomplish, we simply grouped them all in a Scrum board, consisting of the three usual columns: 'To do', 'In Progress', 'Done.' We had thought of doing several boards for each aspect of the project (at least front-end/back-end), but as we are only a small team and as everybody worked on both sides of the project, we did not deem it useful in the end. Finally, we also were cautious when using our VCS (Version Control System), Gitlab, to enable further parallel work: we used branches and merged them back whenever they were ready in to a branch called 'develop'.

Part 2

Design and Implementation

2.1 The stack and the UI

Although briefly mentioned, let us expand on why we chose the tools we worked with and explain the decisions we made with respect to design.

We chose to work using a web application, for a few reasons. As mentioned in the introduction, this makes the application very easily deployable across multiple platforms, since there are no dependencies on the operating system of the user; maybe except for some of them, the user can choose the browser they wish, as JavaScript APIs are quite well unified overall. Then, we decided to do a single-page web app as it is very convenient for the user to navigate through, as they do not need to reload the page every time they want to consult some content. This has however posed a small performance issue discussed later in the section concerning the annotation part of our software. Another big advantage of the web is that it is much easier to create applications that scale well to different screens, thanks to many modern design frameworks; they are then called 'responsive.'

Thus, many choices were available to us with respect to the libraries/frameworks we were going to use. First, we chose Vue.js for management of the front-end because it is a very well documented solution and is remarkably simple yet powerful. So is React, but due to the experience of the team members we sticked to Vue. Then, we needed to decide what design framework we were going to use. Here again, we chose one of the most widespread solutions out there which is Bootstrap (usable in Vue thanks to BootstrapVue). We did so, because Bootstrap's grid system is remarkable and simple: the page is divided into 12 columns (a clever choice, as it can be divided by 1, 2, 3, 4, 6 and 12) and the programmer just specifies the number of columns they want for each of their elements to occupy. It also contained many practical components, and the community and the documentation are exceptionally large — which is particularly important to us to be able to fix issues efficiently. Some claim that Bootstrap is convoluted and out-of-date, but due to the previous considerations and the experience the team members had, it is the solution we kept. We have thus also designed a responsive website, which should work on many screens, although smaller ones are not recommended as they could be quite unpractical; besides, our software was not rigorously

tested on mobile phones, although there is already partial support for it.

As for the server, we decided to use Flask as it is a very flexible option with many different extensions available for it, very easily deployable on servers, and it is also impressively easy to write a piece of software quickly with it. Moreover, a few team members had experience with Flask, and everyone knew how to program in Python, which consolidated our choice.

2.2 The database and the image banks

As we are manipulating user data, we had to decide of a way to store it. We decided to use 'standard' SQL databases, as everybody was at least familiar with relational databases. The question was then: which one should we choose? Postgres? MySQL? We opted for another practical option as it fits the purpose of this program being run on private servers: an embedded SQLite database. SQLite allows having a small database without running an extra piece of software. To use it properly with Flask, we installed the 'SQLAlchemy' ORM, which also allows to define and use models programmatically: that is, we wrote 0 lines of SQL, and only created Python classes to represent our models, and then used pre-defined functions to query, update and delete entries from our tables.

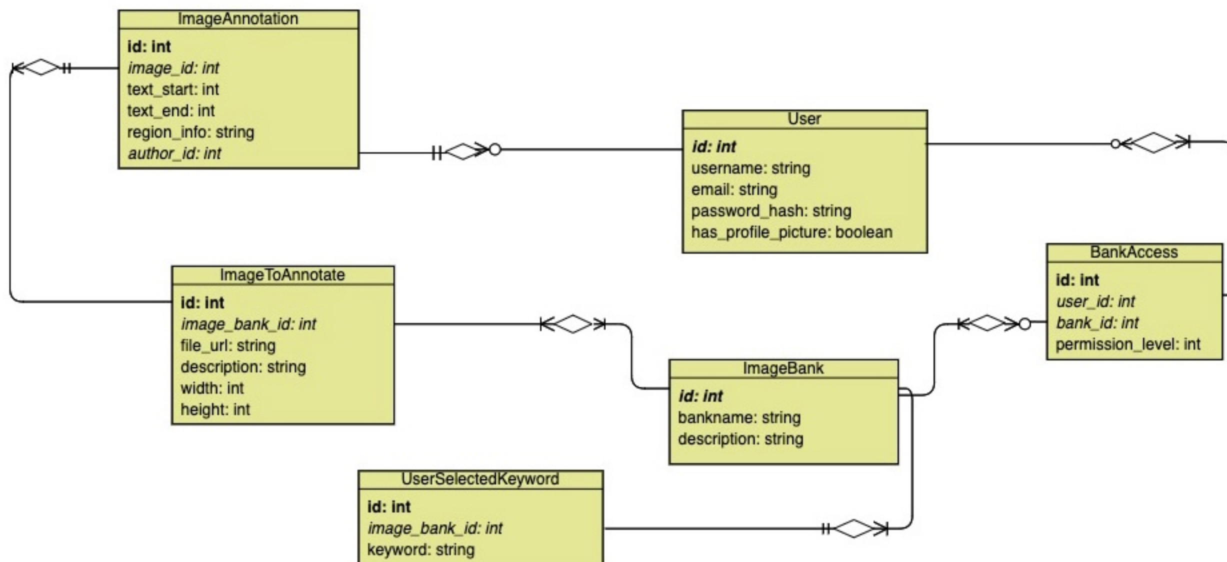

Figure 2.1: the representation of our relational database.

One quite practical feature of Butterfly Annotator is its ability to manage separate data sets, without having to change anything. The design is quite simple: each 'bank' represents a dataset. Each bank has users that can access it, as we will describe in a later section. Internally, each bank is simply a folder named after the bank itself, and containing three types of files:

- ‘description.txt’: a simple file containing a description for the bank,
- Textual descriptions of ‘groups’ in .txt files,
- PNG Images, grouped.

What is meant by ‘groups’ is that each image can belong to a group that categorizes it: it could be for instance its breed if the images represented dogs. To group images, it suffices to all prefix them by naming their files ‘<prefix>_nameofthefile.png’. (Note also that only PNG files are supported, so far.) Then, the corresponding description files should just be named ‘<prefix>.png’; all images of this group will have the same description; if no description can be found, the image will be skipped. As mentioned previously, this allows the user to easily reuse the same descriptions to be able to annotate faster, as they can then re-use the same description bits which the text-processing part will learn as describe further, and as grouped images most certainly require the same textual annotations.

Initially, the only way to upload a bank was to upload in the file system which would yield the rights by default to a single super-admin user called ‘admin’ whose password is editable in ‘password.txt’. Although the previous option is still available, now if one creates a ZIP archive with the previously mentioned structure, then it is possible to drag and drop it on the view displaying the list of banks.

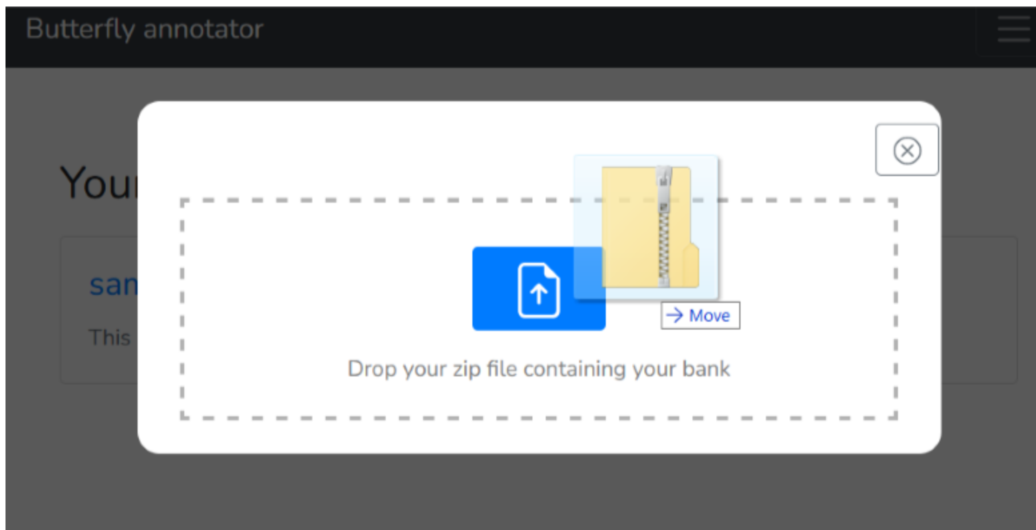

Figure 2.2: the drag-and-drop system.

2.3 Drawing

The key feature of Butterfly Annotator is to annotate pictures with the corresponding description bits. The textual description part is described in the next section, but this one assumes that the user has already selected quotes that they want to attribute to regions on the image (or uses the suggested ones).

The workflow is the following: users need to first create a polygonal area, by clicking on the image to place the points of the shape they are drawing, and then a tooltip window pops up when they close the figure (by clicking on the first point of the polygon), where they can associate the area with one of the previously chosen quotes. Of course, the users are offered a few practical commands, which are:

- Pressing 'Escape' while drawing a shape stops the shape,
- Clicking the last drawn point in a shape removes it,
- Clicking while holding 'Shift' close to a polygon removes it,
- Clicking while holding 'Shift' close to a polygon removes it,
- Clicking and holding close enough to a polygon's point drags it,
- 'Ctrl + Z' and 'Ctrl + Y' allow to undo/redo adding or deleting a polygon, or moving a point.

All these are described in the help section, opened by clicking 'Help?' at the bottom right corner of the screen.

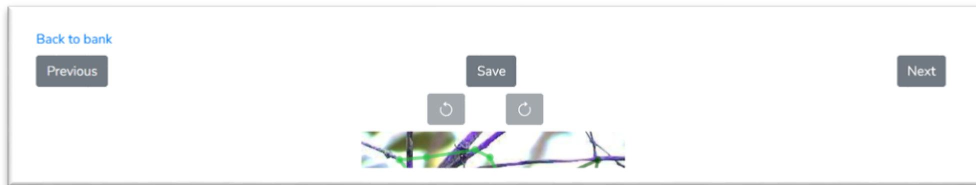

Figure 2.3: the annotation tool itself. Note the 'Next' and 'Previous' buttons allow to navigate between the images easily. You can also notice the Undo/Redo buttons.

From the design point of view, we separated the keywords selection and the region selection on the image, to encourage users to focus on one thing at a time. They would also usually first select keywords and then associate them to regions; so, this avoids going back and forth between the two areas of the screen. One only needs to go through the description text once and selects keywords along the path, and then they can concentrate upon drawing polygon from the image. The pop-up window for combining area and description bit is also designed to show up right beside the mouse to avoid long distance mouse movements.

As for implementation, we utilized P5.js for area selection and management. It is a simple graphics library that is particularly easy to use that allows to draw lines, ellipses, and images on what they call 'canvas'. One of the issues it posed, however, is that it seemed that if the user chose to switch to another image, the resources of the P5 canvas for the previous one would not be freed — probably because of the single-page nature of our application, and is it not exactly made to work with Vue in the first place. To fix this, we simply make the user reload the page when switching canvases.

Also, to make the user distinguish the polygons we created a small function, which, depending on the index of the polygon in the list of polygons, associates it to always the same color: it is like a mathematical sequence.

For the coordinates to be exact, we simply create a P5 canvas of the size of the image, and we keep track of the mouse position when the user clicks on the image and store all the clicked coordinates to form a polygon. Successfully created polygons are stored into an array. An index is kept recording the current polygon for undo/redo functionality. Finally, the 'Save' button initiates the serialization of the data and sends the collection of all the annotations to the server. If the user has edited an annotation or created one, they are recorded as its author, which can be seen by simply clicking the outline of a region that has already been annotated.

Also, implementing the tooltip was not an easy task, as it has to open up on the mouse position of the user when they have finished their shape. To do so, we created a 'fake'/invisible 'div' (an generic HTML tag) that follows the mouse; then, we created the tooltip itself by using another library, called Tippy, that enables easily aesthetic and complete tooltips creation.

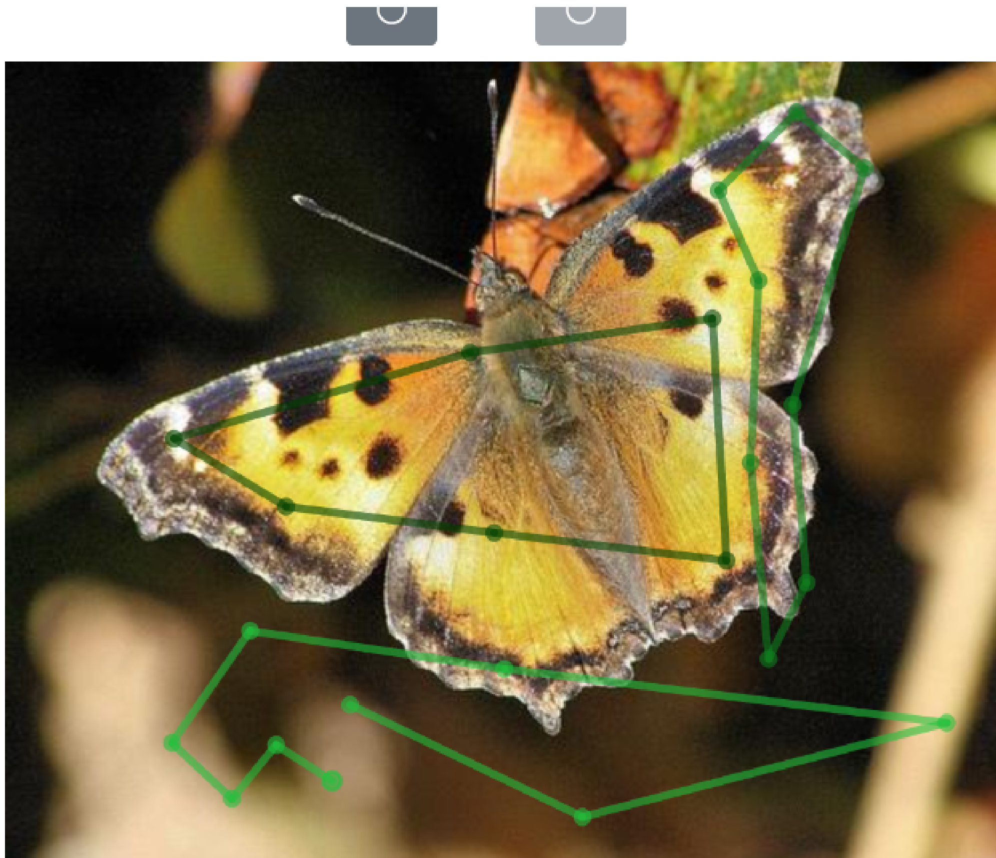

Figure 2.4: a small demonstration of what the regions look like when annotating.

One of the main shortcomings of this tool is that the size of the canvas is fixed; therefore,

the user cannot really zoom in to/out of the image, unless using their browser's zoom feature — which can be an issue for smaller and larger images, respectively.

No bugs that were caused exclusively by this part of the software were detected when stress-testing the tool.

2.4 The description selection

Under the image annotation part of the annotating view, the user can find the description of the current image. They can select keywords and create a mapping between the keywords and the polygons they drew on the image. We provided our users with an effortless way to select text bits, where they can highlight the quotes they want, and, with a click of a button ('Add bit'), the words are ready to be mapped to regions of the picture.

We would also like to provide our users with a functionality that suggests keywords to highlight in the description section automatically. This way, the users would not even have to go through the trouble of highlighting and adding words on their own, for every image.

Our supervisor, Josiah Wang, provided us with three lists of butterfly-related descriptive words, categorized as adjectives, colors, and patterns. The approach we chose is to iterate over all words, and, whenever we encounter an adjective, we assume it is the beginning of an eligible sequence for a suggestion; we thus keep the beginning index of this word, as the start of our suggestion. We continue reading until we reach a noun, whose final character's index in the description will mark the end of this suggestion. If we never encounter an adjective, we stop at the first period or semi-column. We then keep all pairs of indices in a list and pass them to the front-end to process.

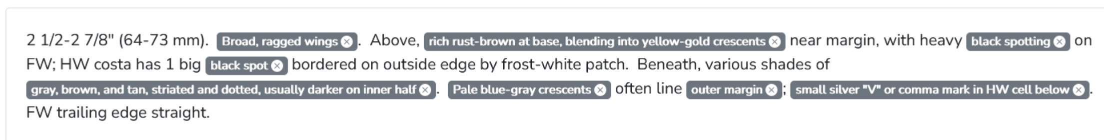

Figure 2.5: an example of the results of our algorithm on a description of a butterfly.

This simplistic process works well, because it relies on the structure the English language imposes on epithets: they come *before* the noun they qualify, usually. This heuristic also works well because we assume that the descriptions are somewhat 'fit' for annotation labels: indeed, we suppose that we will see quotes best described by 'many adjectives followed by a noun', then sentences, for instance, such as 'The car is red.'

To further enhance this functionality, we wanted our system to remember what our users have selected, so that future suggestions will be more accurate, targeting this user. Initially, we tried to update the adjectives list and the nouns list, by processing what users have selected from the front-end. This does not work because we do not know if the first word is an adjective or noun. For example, "wings with dots" can be a keyword that user selects, and we cannot know if the first word is an adjective. So, to make as much enhancement as

we can, we must narrow what our system learns from the user. We introduced a new list named user-selected keywords that stores keywords, that users previously selected and saved. In the future suggestion process, the user-selected list will be traversed first to match the description before mapping the pairs according to the instructor-given lists, and the contents between start and end index of each keyword extracted from the user-selected list will not be considered again in the mapping. This is in the aim of making the priority of user selecting words over auto generating pairs. This can help make the keywords suggestion more precise and save time selecting words manually.

2.5 Undo/redo

A key functionality present in many software relating to images is that of undoing actions and potentially redoing them. It is key, because mis-clicking can occur so simply that it should be easy enough for a user to cancel their mistake. First, let us note that before having to undo an action, a user has already at least two options to cancel their mistakes: they can press the 'Escape' button to stop drawing the current polygon, or they can click on the last point they have added to the shape to remove it. Although it does not cover all possibilities, it comes in quite handy to cancel actions on the fly.

In more sophisticated and complex systems that do actual image processing, such as Adobe Photoshop, the Commander Pattern is usually used with storage/caching of previous states partially. Here, each action is simple: we either add or delete polygons or points, or move points back and forth. Therefore, we chose to implement this system in a simple yet efficient fashion.

We implement this feature by creating a structure resembling doubly linked list with a pointer to artificial nodes for a head and a tail, and a pointer to the last node whose action was done, call them 'head', 'tail' and 'curr' respectively; we will call this structure 'history'. Each action that a user does is represented by a node in the list; the node itself contains two pointers (to point to the previous and next actions) and two anonymous functions. The latter are used for undoing and redoing an action; what we have indeed noticed is that as all actions are simple, it suffices to change the state back to what it was before in a few lines of code. Therefore, when a node is first inserted, its 'redo' action is executed, and it is appended at the end of the chain.

Initially, the 'head' node has no previous element — which will never change — and no next element for implementation reasons on when we insert the first element; the 'tail' node has no next element — and never will have one — and has as a previous element 'head'; finally, the 'curr' pointer points to 'head'. Thus, whenever an action in history is inserted, we take 'curr' and make its next pointer point to the new node, and we also set the previous pointer of 'tail' to point to the new node as well; finally, we set 'curr' to the new node.

Let us show briefly why this works well. In the case that no action has been undone, then the next pointer of 'curr' is 'tail' and the previous pointer of 'tail' is 'curr'. Then, we simply append the element to the end. If actions were undone, then setting as if it were the end of the linked list, losing the pointers to the previously held pointers by 'curr', makes it

the last done action and discards the previous 'branch' of the history — which is what we want.

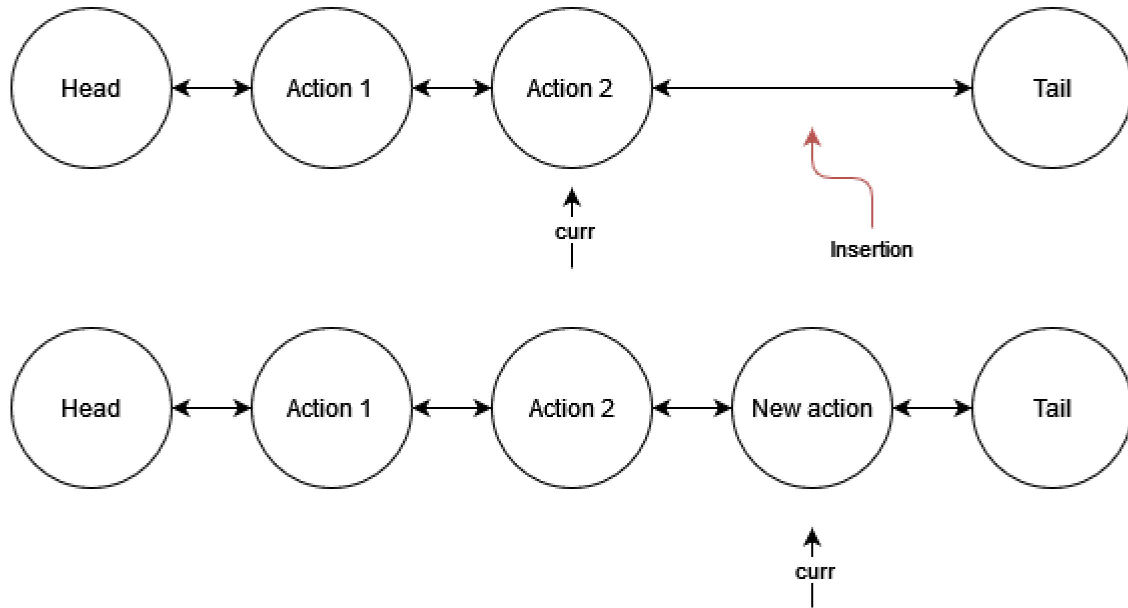

Figure 2.6: inserting when the last action is the most recent.

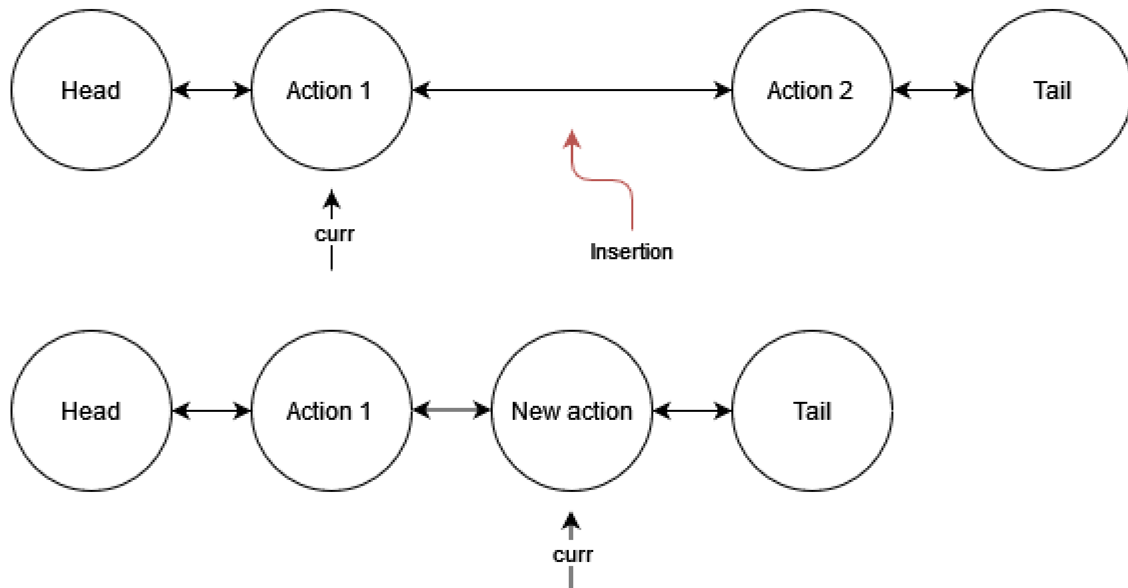

Figure 2.7: inserting when the last action is not the most recent.

Note that we have these 'head' and 'tail' nodes so that we know when there are no further actions to redo, or no previous actions to undo. We could have done this by identifying 'null' elements, but this way is much cleaner, as we do not have to disambiguate whether 'curr' points to a node whose action was taken or not: it has always been taken. Otherwise,

especially for the first action in the history, we would need to know if the action has been taken or not, since if 'curr' points to the first element, then we need to determine if the action has been undone or not. (If it has, then setting the pointer 'logically' to the element the previous pointer points to, would make the 'curr' pointer 'null' and could then lose the entire history.)

We have then just implemented this in our JavaScript code, where pointers are simple references to the objects. For the aesthetic side, we have two undo/redo buttons, surrounding the 'Save' button — disabled when nothing can be undone and/or redone, respectively. And users can of course enjoy applying their favorite key combinations that are "Ctrl + Z" and "Ctrl + Y".

Except for some rare display bugs, this part of the software seems to work correctly on its own. However, due to poor and quick conception on the annotation tool side, some obscure combinations of saving new annotations an undoing/redoing them can cause small side effects, such as duplicate regions. One of the reasons for these bugs is that JavaScript is statically typed; indeed, it makes it difficult to get a hold on which field is part of a certain object or if types match. Therefore, we have shortly considered switching to TypeScript, but it was too late, as we had already an important code basis. This issue can however be easily fixed by simply reloading the page and deleting one of the extra polygons.

2.6 Simple multi-user permission system

Once the user has successfully logged in or created an account, they now need to access banks to start working. As a side note, we will not discuss much of the logging/registering system, because it is standard; note, however, that users can edit their profile picture, which is useful when navigating through the list of images — indeed, they can see who the last user to have annotated each image is.

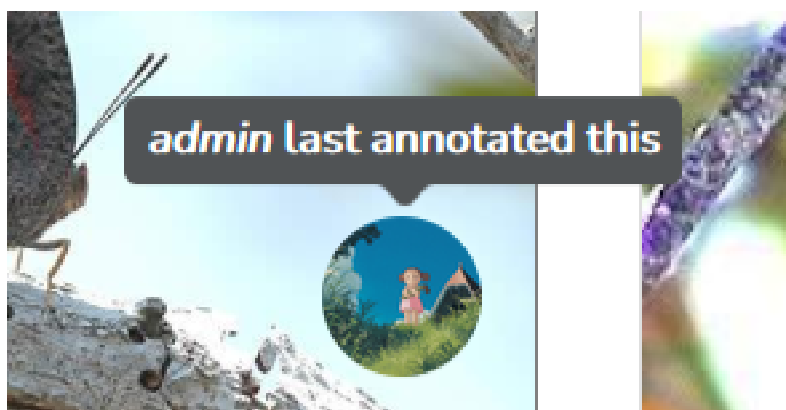

Figure 2.8: 'They can see who the last user to have annotated each image is.'

What we wanted to do is to create a flexible and complete system in which annotators can effectively work together. To do so, we have thought that a hierarchical structure among

the annotators would help to allow potentially large teams to annotate together, without too much struggle.

To manage accesses to banks we have represented them in our database as the 'BankAccess' model. Each such entry refers to a user and to the bank to which they are given access to, and it also contains the 'permission level' with which might want to access. The permission level is an integer which represents the authority the user has on the bank: the higher, the more they have power. We have identified a few distinct categories that might be useful: 'Viewer/Visitor' (so that users can just consult the bank and its annotations), 'Editor' (so that users can also edit annotations), 'Moderator' (so that users can moderate the team, i.e., add or remove people), 'Admin' (for the leaders of teams, who can also delete banks). Note that each level encompasses the permissions of the previous one.

Finally, note that this is represented by a tab on the view where the user can see the list of images, named 'Accesses'.

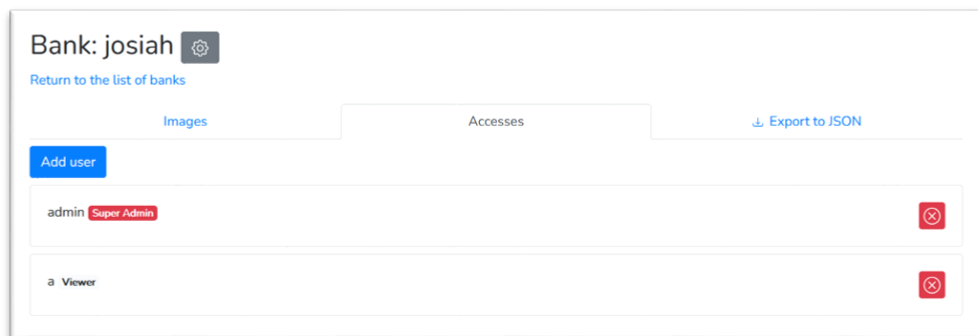

Figure 2.9: the 'Accesses' tab; notice also the 'gear' button next to the name of the bank, which allows to access settings (currently just deleting the bank).

Part 3

Evaluation

3.1 Comparison with existing solutions

We would like to compare this butterfly annotator with the existing commercial online image annotator "PlainSight" to find out what are the strengths and shortages of our project.

Differences in main functions:

PlainSight supports uploading files in multiple formats, including single image, via CSV, Amazon S3 and via Google Cloud. While our product only accepts zip file in a strict format as butterfly annotator is designed for specific bank images, which includes many images and description in text.

PlainSight only supports doing one label at a time while our software allows users to upload a whole description and map label area to different text.

The way of selecting items of PlainSight is flexible as it offers rectangle, polygon, etc. There is even an AI called 'SmartPoly' to polish the curve after one draw a rectangle upon the item. Our software can only do polygon selection.

Butterfly annotator has a multi-user permission system. It allows users to work on the same image bank with permission. PlainSight system is implemented like GitHub. There is a graphical dashboard indicating 'Overall Assests', 'Recent Activity', 'Label Usage', etc. It even comes with a version control system.

Our shortages

PlainSight provides many more buttons to help annotate, including 'Rotate Image', 'Resize to Fit', 'Delete label' to manipulate image/label, and 'Skip' to skip current image for now. It also allows users to change the appearance of image and pen. There is also a graphical progress bar beside the canvas to manage the whole annotation process. These are quite practical features our software does not implement.

Our strengths

Because of the more precise and narrow scope of our software, we have some tools that are quite adapted to the tasks at the hand, namely the description system, which can enable very quite labelling — as the user does not have to write the label at all.

3.1.1 Conclusion

Although it is of course a more powerful tool as it is developed by a larger team that has been running for quite a while now, we believe that the software can be convoluted a bit on some sides, despite its undeniable strengths. However, this is not an issue we have been facing, because we wanted to make the tool as intuitive and easy as possible, so that users can start working as quickly as possible.

3.2 Unit tests

We add unit tests in our front-end part (Vue) to test our webpage components' features and make sure we have high quality code. We use "Vue Test Utils" (VTU) and "@testing-library" to simplify testing Vue components. These tests are used for checking our Vue components are working properly and meet the functionality we want to show. They do not include any implementation details, so any refactors of our components do not break our tests. These can also improve the maintainability of our code, making it easier to fix bugs when updating the code at some point in the future.

3.3 How successful was the project overall & does this meet the needs of prospective users?

Our goal in this project was to provide our users with the easiest and effective annotation tool. We assumed our users would want to see a clear and simple annotation tool that is powerful yet convenient. We made our interfaces as simple as possible, and every feature we provided is easy to use. We did well in practicality and convenience in our project, as all the functionalities enabled by our software are accessible and enable quicker annotating.

Enabling the upload of zip archives was an important addition. At the beginning, all the images need to be uploaded separately, and their textual descriptions need to be copied and pasted manually, which takes a lot of time if the user want to upload lots of them. So, we tried to improve this feature by supporting the upload a combination of images and texts at the same time, which met the needs of Josiah Wang — our project's supervisor. We then further took the feature to detecting folders in file systems, and now dragging and dropping makes the process of creating an image bank very straightforward.

Providing automatic keywords selection for the image's description is a key feature for

annotators to be efficient. Indeed, it saves them a lot of time, because they do not need to type the labels of everything they annotate, every time. This is particularly true when we started learning from the user input, as the same labels would get suggested for the same type of images.

Overall, we believe the project was quite successful, except maybe for the first iteration, when we had to determine how we were going to work efficiently together and get used to that workflow. We followed the initial plan from the beginning during every iteration and completed all the required functions at the end. We achieved our original goal which is to make annotators' life easier.

3.4 How could this project be taken forward?

The most practical feature we could implement is to propose suitable candidate image regions that might be mentioned in descriptions automatically. This will be hard to achieve but it will save user's annotation time and make their work much easier. Another improvement could be reducing the errors in automatic keyword selection and try to cover more potential keywords. We have made a lot of progress to do in this feature, but still need to improve and NLP could be a viable technique to use. In addition, we have a quite small coverage of unit tests on our back end, and we should write more of them for further versions, as the number of features increases.

We could also implement more features for large teams, with respect to the permission system, and why not add web hooks for other applications to integrate well with ours. One practical component we could implement is to allow users to partition their banks, so that they can know which user has to annotate which image. Another desirable collaborative tool is the management of simultaneous editing. This could be simply implemented using Web sockets to broadcast events, although there would be quite a few challenges with respect to synchronization.

Finally, since we have made a single-page web application, we could actually try to deploy it as a software to install for users, using frameworks such as Electron, which enables making software that can be installed on one's computer.

Part 4

Ethical Considerations

As we view our software, we see three main ethical considerations.

The first of them is the data security. Of course, the data, once uploaded to our servers, should be safely stored and not accessible to any intruder, especially if the stored images and descriptions are sensitive information or even just owned exclusively by the user(s). However, the goal of this software was firstly to be functional and quite a few simple breaches are still exploitable in our product as it currently is. This is because we chose to focus on functionality rather than security, for time considerations, but also because we considered initially that users will often self-host our software and run it on their own servers which would accessible only to people of trust in the first place, if not just run locally on their computer. Moreover, quite a few vulnerabilities could be simply fixed, should the software be deployed on larger scales.

Then, there is data privacy. Indeed, because of the focus on features and the assumption that most users will run this program on their private servers, as things are, we simply store users' files locally on the server. If the software was ever to be continued, files cannot be boldly stored on the server, as it poses not only security issues, but also if the servers are owned, say, by a company, then this company can access the data without any trouble. To palliate this issue, files could be for instance encrypted using key pairs of some sort, with which only the user owns the private key. This way, once the file is uploaded and is verified to be a valid image, it would be encrypted for storage, and decrypted whenever the user accesses them. However, this creates other challenges with respect to multi-user systems (as the same key should be shared) and implementation, as we would need to access the private key from the browser too — which then poses the problem of how to store them properly.

Finally, one final consideration we have thought of is that of "unethical" text-processing suggestions or suggestions of other kinds. This could clearly be the case if the software's features are taken further. Indeed, if for instance a region suggestion feature was to be added, that would suggest some areas of the picture and some label to be combined for an annotation, some samples could face discrimination. It has been quite actively discussed recently when talking about ethics in AI, namely with Twitter having a piece of software choosing which part of an image should be displayed when it is not fully displayed; it was

shown that faces from ethnic minorities would not be recognized by the system. Although the software is not at this stage yet, it is an issue to foresee, since, as previously mentioned, ethics in automation are highly discussed.