Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.Y. 2022/2023

Graduation session October 2023

Web UI code generation

A transformer-based model applied to real-world screenshots

Supervisors:

Luigi De Russis

Tommaso Calò

Candidate:

Giuseppe Salvi

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

In today's digital age, the prominence of online platforms has heightened the significance of websites in establishing a strong online presence for both individuals and businesses alike. The virtual face of an entity, whether it's a personal blog or a corporate site, plays an important role in creating the first impression and ensuring sustained engagement with the audience. As a result of the high importance of websites, there is a growing requirement to develop complex, attractive, and sleek websites to gain a competitive advantage.

A crucial step of website creation involves conceptualizing the User Interfaces (UI) design through various stages, from drafting to prototyping. This iterative process engages end-users, stakeholders, and developers, enabling them to deliberate upon the website's proposed layout, composition, and interactivity.

Three primary artifacts dominate the web design landscape: hand-drawn sketches, wireframes, and mock-up designs [1, 2, 3]. The sketch serves as a preliminary, often rudimentary, representation of the intended UI design, enabling designers to swiftly consolidate and visualize their ideas. A wireframe, acting as a visual prototype, pinpoints the positioning of UI elements and content on the site. Lacking styling, graphics, or colors, it resembles a website's blueprint. The mock-up design stands as a more refined and visually enriched version of a wireframe. It encompasses styling, graphics, colors, typography, and other intricate visual specifics. After securing approval for a wireframe or mock-up design from end-users or stakeholders, a web developer proceeds with the actual website creation. Given that design and implementation typically fall under separate teams, the journey from concept to completion is not just time-consuming but can also be expensive. Such professionals invest immense effort in iterating, designing, and developing a site to meet clients' expectations.

Considering these challenges, the concept of automatic website generation has arisen. Automatic web generation refers to the use of software and technologies to automatically produce websites without the need for manual coding. This method integrates design and implementation phases, minimizing the need for back-and-forth adjustments between separate teams. By doing so, it ensures a more direct translation of design intent into the final product. Consequently, the reduced iterations lead to quicker delivery times, cost savings, and fewer chances for miscommunication or errors that arise from repeated handoffs between design and development teams.

Furthermore, the integration of automation allows for real-time adjustments based on immediate feedback loops. In traditional development scenarios, alterations post-deployment often require a revisit of the entire design-development-deployment cycle. With automatic generation, modifications can be made on the fly. Moreover, automatic website generation can help democratize web creation. Those without a technical background or coding skills can still venture into designing and deploying professional-level websites. This bridges the gap between developers and those without technical expertise and fosters a more inclusive digital landscape where creativity isn't bounded by technical constraints.

Within the predominant modalities for automatic website generation, the mock-up-driven approach, as the name implies, derives its functionality from mock-up designs and wireframes. Directly converting detailed mock-up designs or wireframes into functional GUI code eliminates the conventional, manual transition from design to coding. The fidelity of the resulting websites can be considerably high, given the detailed nature of mock-ups, thereby ensuring a closer alignment with the designer's original intent. Similar to the mock-up-driven approach, which utilizes the most refined artifacts, the sketch-driven conversion method uses the most preliminary stage of design representation: hand-drawn sketches.

The approach is particularly beneficial for novices; it offers an opportunity for those unfamiliar with web development processes to transform their basic sketches into functional websites.

Sketches serve as a natural form of human-AI interaction because they harness the inherent human ability to visualize and express ideas through simple drawings, irrespective of technical expertise. This universal method of representation ensures intuitiveness and ease of use for a wide range of users, bridging the gap between imagination and digital creation.

Moreover, for designers, sketch-driven conversion allows them to rapidly test their interactive prototypes, making it easier to iterate and refine their ideas at a faster pace, as well as gain feedback.

Two modalities of mockup-driven and sketch-driven automatic website generation are possible: the heuristic-based approach and the end-to-end approach. The heuristic-based approach processes sketches and mockups by leveraging a set of

predefined rules and patterns. In this framework, algorithms make determinations based on known patterns and guidelines. For instance, in the context of automatic web UI creation, a rectangular shape in a mockup might be recognized as a button, and a series of parallel lines might be interpreted as text fields. The algorithm then generates the necessary code based on these identified patterns.

Yet, this method has its limitations. Heuristic-based systems can be inflexible, and their effectiveness heavily relies on the quality and capabilities of the rules they are based on. If a design includes a novel element or a unique layout, the heuristic model might misinterpret it or fail to recognize it altogether. Moreover, it's a challenge to constantly update and maintain the ruleset as design trends evolve and as the complexity of designs increases. This means that while heuristic-based methods are generally efficient for more standardized and common designs, they may fall when faced with more intricate or innovative mockups.

On the other hand, the end-to-end approach represents a shift from relying on hard-coded rules. It adopts machine learning, specifically deep learning models, to handle the entire process of converting a mockup to a functional web UI. Instead of operating on a set of fixed rules, these models are trained on vast amounts of data, comprising various mockups and their corresponding web UI outputs. The more data they are exposed to, the better they become at making accurate predictions.

The advantages of the end-to-end method are several. First and foremost, it can handle a broader array of designs, including those that might fail for heuristic systems. Given adequate training data, it can continually adapt and improve, keeping pace with evolving design trends. Furthermore, it can discern and learn subtle patterns and nuances in designs that might not be explicitly defined in heuristic rules.

To illustrate, consider an unconventional mockup where buttons are represented by ellipses instead of the typical rectangles. While a heuristic system might struggle to identify these as buttons due to its rule-based nature, an adequately trained end-to-end model could recognize them based on its exposure to diverse design patterns.

Building on the innovations offered by deep learning architectures, end-to-end approaches have emerged as a powerful tool for turning mock-ups and wireframes directly into functional code. The encoder-decoder framework, a common structure in these methodologies, traditionally leverages convolutional neural networks (CNNs) to parse image features, converting visual representations into intermediate language constructs, which are subsequently decoded to yield the desired code. Pix2code [4] was the first contribution introducing an end-to-end approach for the task; it is capable of translating web user interface screenshots and transcribing them into domain-specific language (DSL) representations, which can then be compiled into specific HTML code.

While the advantages of these deep learning models are multiple, there remain

limitations that cannot be ignored. One of the more important issues revolves around the datasets these models are trained on. Many of the currently available UI/code datasets lack diversity, often being overly simplistic and not adequately representative of the complexities encountered in real-world scenarios. This simplicity limits the true applicability of these models in practical settings. The models, having been trained on such datasets, do not generalize well when exposed to more complex, real-world designs, thereby hampering their effectiveness.

Moreover, a significant portion of these models remains rooted in somewhat outdated machine learning architectures. For instance, while many of the described models employ Long Short Term Memory (LSTM) structures for decoding the visual representations, the potential of transformer-based architectures remains largely unexplored in this context.

Attention mechanisms, central to transformer models, can be especially advantageous for multimodal tasks. Due to the attention mechanism, they have improved capabilities of capturing relationships between visual components and their corresponding code representations, which can result in a more accurate conversion from design to code. By integrating these newer models, it's conceivable that the accuracy and applicability of mockup-to-code and sketch-to-code conversions could see substantial improvements.

1.2 Contributions and thesis outline

Our contributions are threefold and can be concisely summarized as follows:

- **Architectural shift:** At the architectural level, the previous dependence on Recurrent Neural Networks (RNNs), especially LSTM-based models, provided a foundational contribution to the domain. However, for the specific task, there is a need for improved performance and scalability to suit real-world applications. To accomplish this we introduce a multimodal transformer architecture that, when trialed over existing website datasets, outperforms the traditional LSTM-based models, aligning with the broader AI trends and hinting at the potential of such architectures (i.e., scalability and expressivity) in the future.
- **Dataset Enhancement:** We introduced a new real websites dataset and a specialized scraping pipeline to curate and clean scraped code. By eliminating non-essential tags and scripts for visual appearance, our approach ensures that the resultant dataset is less noisy. This, in turn, optimizes the transition from design mockups to functional website code, making the process more efficient.
- **Innovation in Practical Application:** Moving beyond the technical advancements, starting from generating a synthetic dataset of HTML Bootstrap

websites that mirrors the diversity of web designs, we leverage it to adapt our model to enable direct sketch-to-code translations. By facilitating the transition from rudimentary sketches to functional code, our system serves as a powerful tool to bring advancements to end-user website development and Human Computer Interaction (HCI) research.

Through our work, we aspire to advance and enhance the research in the realm of design-to-code transitions, especially in real-world settings. We believe that the methodologies we introduce can foster a more accessible and efficient creation of web interfaces.

The thesis is structured as follows:

- Chapter 2 focuses on the literature study of automatic code generation techniques
- Chapter 3 presents an overview of the tool for the extraction of website code and screenshots
- Chapter 4 shows the experiments conducted for the dataset creation and additional components introduced to guarantee results quality
- Chapter 5 focuses on the Pix2Struct model presentation and modification together with experimental results
- Chapter 6 offers a concluding discussion regarding the results, as well as the future prospects of both the tool and the model

The code used for the generation of the dataset and the notebooks used to run the experiments are available in this GitHub repository:

- `https://github.com/giuseppesalvi/webUI2code`.

Chapter 2

Literature review

2.1 Related works

Automatic website generation has emerged as a solution to ease the challenges associated with web design and development. Automating the process makes it possible to quickly and efficiently create websites without wasting time on manual coding.

There are three primary methodologies underpinning automatic website generation: Example-based automatic website generation, Artificial Intelligence-driven website generation, and Mock-up-driven automatic website generation.

Example-based automatic website generation allows users to create websites by referencing and adapting features from existing, professionally designed sites. By referencing real-world designs, it ensures quick customization and aesthetic appeal without the need for in-depth technical knowledge. Artificial Intelligence-driven website generation employs AI algorithms to design and build personalized websites based on user preferences, requiring minimal user intervention. Mock-up-driven automatic website generation transforms visual mock-ups or sketches of websites into working digital prototypes, often using heuristic techniques or deep learning models. It ensures that the final product closely aligns with the initial vision.

Each method has its nuances. Mock-up-driven generation streamlines the transformation of a visual idea into a working prototype, but it requires a clear mock-up or further programming and designing effort in the case the translation starts from wireframes or sketches. The example-based approach provides more freedom to those less technically inclined, allowing for customization based on real-world examples, but it might not be as tailored or unique. On the other hand, while AI-driven tools are powerful and user-friendly, they might not always offer the depth of customization that professionals need, and the results might vary based on the sophistication of the AI algorithms. [5, 4, 26, 37, 43, 44, 45, 46, 47, 6,

7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42]

2.1.1 Example-based automatic website generation

Many novice designers and developers, often with minimal web expertise, increasingly turn to pre-designed website templates. These templates help them establish aesthetically pleasing sites without delving into the complexities of GUI code [42]. Though such templates offer customization options, including theme colors, font adjustments, and image uploads, their scope remains limited. Thus, some users might find them insufficient for their specific needs.

Addressing this issue, researchers devised a method allowing those with little technical know-how to easily construct custom websites inspired by real-world, professionally designed sites. This system empowers beginners to explore these design exemplars, discern layout structures, select appealing elements and themes, and amalgamate these into their own designs through automated code generation.

Myers et al. [48] presented WebCrystal, a game-changing tool facilitating the extraction and replication of desirable HTML and CSS attributes from existing websites. The tool guides users with pre-set queries about adjusting HTML/CSS features. These questions come with textual descriptions, helping users choose the right attributes. WebCrystal then generates the suitable HTML/CSS code, which can be integrated directly or tweaked further. However, it has two main limitations: it's compatible only with HTML and CSS, neglecting client-side scripts like JavaScript, and occasionally, the extracted code might not perform identically across different sites.

Hashimoto et al. [49] introduced a system allowing UI designers to probe for site designs mirroring their sketched layouts. Using a crawler, it amasses web pages into a database. When users sketch a desired layout, the system offers sites with similar designs. This aids novices in selecting and understanding HTML/CSS design structures. Still, due to database constraints and matching algorithm limitations, it doesn't always locate the perfect design match.

Swire [50], geared towards mobile UI/UX design, lets designers explore Android UI designs resonating with their sketches or screen captures. Unlike the method by Hashimoto et al. [49], which relied on heuristic analysis, Swire uses a deep learning approach. However, Swire struggles with unique UI widgets or varied colors.

Xiaofei [51] developed a tool that identifies Android apps resembling hand-drawn GUI sketches. It translates sketches into an intermediary language, applies deep learning to generate GUI frameworks, and then finds analogous apps from a database. While promising, it doesn't consistently provide perfect matches due to search strategy limitations.

Behrang et al. [52] introduced GUIFetch, offering Java source code for Android

apps closely matching users' GUI sketches. It involves two main stages: analyzing to detect potential Android apps and computing similarity scores between the sketched GUI and app GUIs. This tool aids in the tedious UI development phase, but currently, it can't detect or compare images.

2.1.2 Artificial Intelligence-driven website generation

The contemporary field of web development is significantly enriched by the advancements in artificial intelligence. Modern AI-driven website builders collect user preferences through a series of predefined questions. Following this, they autonomously design and construct personalized websites based on the user's preferences, theme, and content, obviating the need for manual coding. Consequently, even those without a background in web design or tech can effortlessly establish their online presence using these platforms.

A few pioneering commercial platforms dabbling in this artificial design intelligence are highlighted below: The Grid [53] is an AI web design tool that guides users through selections of color schemes, web components, fonts, and layout patterns. After inputting content, The Grid crafts the website, and users can recalibrate their initial style choices for further refinements. One drawback is its minimal design editing features.

Bookmark [54] is a cloud-based AI website platform that allows users to simply and quickly build responsive one-page websites that, through a question-intensive first phase, ensure to meet the layouts and styles specified by the users. On top of this, it offers an expansive library of e-commerce and industry-specific templates.

Firedrop [55] specializes in building websites for small enterprises. By chatting with a virtual assistant, users can communicate their design preferences, which Firedrop then translates into the final design.

Wix ADI [56], has established itself as a leading contemporary AI website platform, offering affordability and customizability. It excels in producing websites by integrating optimal design layouts and components.

In 2018, Leia [57] emerged as an AI website platform enabling the building of mobile-responsive sites initiated by simple voice commands or keywords.

Zyro [58], introduced in 2019, specializes in crafting SEO-optimized, responsive websites for small businesses with an array of template options and a user-friendly drag-and-drop interface. Additionally, it offers a complimentary logo-making tool, although its template range is somewhat restricted and basic.

While these AI website platforms signify an important shift in web development, a comprehensive exploration of their underlying mechanisms and research is not publicly accessible.

2.1.3 Mock-up-driven automatic website generation

A prevalent strategy to transform mock-up designs of websites into working prototypes harnesses *heuristic techniques*. The heuristic-based approach to website generation focuses on the use of domain-specific rules to guide the process. Such techniques typically extract web elements, discern their semantic relations, choose the fitting tags for these elements, structure the web elements hierarchically, and subsequently produce the source code.

The method proposed by Huang et al. [43] identifies vertical and horizontal separators on the mock-up based on color differentiation. The resultant sections contain distinct web elements for which tags are generated. The tag generation uses the Random Forest method [59] for basic elements and a bottom-up approach for more complex elements. The heuristically proofed tags then inform the final website's HTML and CSS structure.

Shifting our focus to mobile app development, Nguyen et al. [44] introduced REMAUI, a method that automates UI source code generation for mobile apps from mock-ups. REMAUI's six-step process begins with the Tesseract Optimal Character Recognition (OCR) engine, which extracts text from screen captures. To rectify OCR's occasional misclassifications, domain-specific heuristics are employed. Alongside, computer vision methods identify UI boundaries to establish a UI hierarchy. REMAUI then refines this hierarchy, constructing a UI suitable for mobile apps. Upon testing, REMAUI averaged a 9-second runtime but faced challenges with certain OCR limitations and prototyping multi-page applications.

P2A [45], in contrast, addresses REMAUI's limitation by prototyping animated mobile UIs from screen captures. Like REMAUI, P2A employs computer vision and OCR to detect UI widgets, but with the added capability of enabling users to add custom animations and transitions. After integrating these enhancements, P2A produces an executable with necessary asset files. However, while heuristic methods can be accurate and efficient, they come with the limitation of the inherent imperfections of heuristic rules, which might not account for every scenario or outlier.

Transitioning from heuristic, a distinctly different approach gaining traction in the domain of website generation is the use of *end-to-end methods*. These methods utilize deep learning models to transform website mock-ups or sketches directly into operational Graphical User Interface (GUI) code. They harness deep learning classifiers for converting visual layouts to code via an encoder-decoder setup. Notably, this strategy doesn't use preliminary image processing or heuristics but relies purely on the inherent capabilities of neural networks to interpret and translate visual designs. It is within this area that our contribution stands out, introducing innovative solutions to the existing challenges.

Pioneering this field, Beltramelli [4] introduced the "Pix2Code" model, leveraging

a dataset of UI snapshots from iOS, Android, and websites; the method works by producing intermediate Domain Specific Language (DSL) code. This model includes one CNN and two LSTM networks. First, a CNN-driven vision model encodes UI captures into a fixed-length vector, and an LSTM parallelly encodes the DSL context into an intermediate representation. These vectors combine and are decoded using an LSTM, eventually classifying DSL tokens through a SoftMax layer.

Several modifications of the architecture of "pix2code" [4] then emerged: Zhu et al. [46] introduced a model emphasizing UI components' hierarchical layout in the code. A CNN-based vision model extracts visual details from UI components, feeding a hierarchical LSTM decoder. With attention mechanisms, this model demonstrated a more accurate GUI code generation. Liu et al. [28] substituted LSTM with Bidirectional LSTM (BLSTM) (see Fig. 8), improving the accuracy results on pix2code's dataset.

Han et al. [6] aimed to create webpages with Cascading Style Sheets (CSS) styling details; their model utilized object detection methods and attention mechanisms to determine CSS contents.

Kumar [7] developed SketchCode, converting hand-drawn wireframes to HTML; leveraging the pix2code framework, it incorporated Gated Recurrent Units (GRUs) for encoding and decoding.

Yong Xu et al. [8] crafted image2emmet, detecting GUI elements in web images, converting them to HTML-CSS; the tool integrated a Faster RCNN [33] and an LSTM, focusing on individual GUI elements instead of entire websites.

Chen C. et al. [9] transitioned web mock-ups to mobile design code using a generative tool with an RNN encoder and decoder and tested on 1208 real-world Android screen captures.

Building on the work of prior models, our research identifies and addresses the limitations often seen in RNN-based architectures, especially when considering their training dynamics and scalability. While RNNs have been foundational in the earlier stages of automatic website generation, the true power of attention mechanisms, central to transformer architecture, remained untested. We thus introduced a multimodal transformer architecture, aligning with the recent trends seen in the broader AI community. Our empirical data clearly underscores its superior performance over existing datasets. Beyond performance enhancements, this approach sets the stage for enhanced scalability, offering the promise of more sophisticated models in the future of automatic website generation research.

The power of the end-to-end approach comes from the ability of these models to learn from vast amounts of data and generalize to new, unseen data. End-to-end methods can sometimes produce more fluid and adaptive results due to their learning nature. However, they might require significant amounts of labeled data for training. Many current end-to-end solutions for automatic website generation

are benchmarked against Beltramelli's "pix2code" dataset. Such a dataset, while beneficial for initiating research, might not sufficiently capture the complexities of real-world web designs. As a consequence, models trained solely on these datasets could be confined in their abilities and may not generalize well when faced with more intricate and diverse designs outside their training scope.

Moreover, more complex proposed datasets, such as the one proposed by Chen [9], or RICO [41], focus primarily on mobile UI hierarchies, not on compilable HTML code. This highlights a significant gap in the available resources: a complex dataset of websites' code and mockups remains absent. The recently proposed WebUI dataset [60] does take a step in this direction by comprising scraped HTML code of the webpage. However, it brings along unnecessary tags and scripts that do not contribute directly to the reconstruction of the mockup. This excess of data can introduce noise and complexities, making the task of code generation from mockups more challenging than it needs to be.

In response to this identified challenge, we introduce a scraping pipeline specifically designed to clean the scraped code. By removing unnecessary tags and scripts, our processed dataset becomes more streamlined and better suited for code generation from mockups. This approach ensures that the models are not overwhelmed or misled by irrelevant code, making the translation from design mockup to functional website code more efficient.

Additionally, in recognizing the value of human-computer interaction (HCI) and the evolving landscape of website design, it's important to ground our research in real-world applications and to ensure it remains user-centric. More than just performance metrics, research in automatic website generation should aim to enhance collaboration between designers, developers, and end-users.

Towards this, we present an HTML bootstrap synthetic dataset that, unlike other datasets, offers a wide array of components and layouts, targeting a diverse range of design elements to mirror different design scenarios.

Given the synthetic nature of our dataset, we provide precise component localization. Utilizing this, along with the "synz" dataset [61]—comprising sketched web components by designers—we transform our dataset into a sketch-HTML dataset that we use to train a sketch-to-code system. Unlike previous proposals, our system is more than a technical showcase. We present it as a foundational tool for subsequent explorations in end-user website development, HCI, and broader computer science research. By simplifying the transition from a sketch to functional code, sketch-to-code systems would facilitate a more intuitive design process, lowering the barrier of entry for novice developers and catalyzing collaborative efforts between designers and developers.

In conclusion, our research extends beyond the scope of traditional automatic website generation. By bringing technological advancement and showcasing their practical applications, we hope to inspire and contribute meaningfully to the

interplay between human creativity and computational capabilities in the field of website design and development.

Chapter 3

Website code and screenshot extraction tool

At their core, websites are built from a blend of codes and resources that enable them to operate and present their design in an aesthetically pleasing and user-friendly way. The transformation from code to visual is orchestrated by three fundamental building blocks: HTML, CSS, and JavaScript.

HTML (HyperText Markup Language) gives structure to the web content, defining elements like paragraphs, headings, and areas where other media, like images and videos, will be displayed. HTML creates a tree of elements, allowing browsers to understand and display the intended structure of the pages.

CSS (Cascading Style Sheets) enriches websites' appearance by providing style, color, and layout. It defines rules on how the various building blocks of the website should be displayed visually. It can be used to ensure that websites maintain a consistent look and feel across different platforms and screen sizes.

JavaScript enables interactivity, allowing the website's reaction to user inputs and a dynamic update of the displayed content. Modern JavaScript frameworks can be used to build websites, and their appearance is profoundly influenced by Javascript code. Moreover, to embed additional resources like images, videos, or fonts, websites often include various files and links.

In our context, since the final goal is the creation of a dataset of static representations of website pages through their screenshots, alongside their code, we decided to collect only HTML and CSS files, avoiding JavaScript. The screenshots were collected in this setting, and some strategies were used to identify websites that rely massively on JavaScript code through frameworks. To limit the amount of information needed to represent each website, we decided to avoid downloading all the resources associated with the websites and replace images with a static default one. This is particularly relevant for downstream machine learning tasks, where a

network is asked to predict website code and cannot retrieve the original website resources.

In the output of our tool, a website is represented by three fundamental files: an HTML file, a CSS file, and a PNG image of the screenshot.

3.1 System overview

This tool is designed to accomplish two primary objectives: extracting the code and taking a screenshot of a website. In addition, this tool is primarily intended for the generation of a dataset of websites, which will be used in machine learning applications.

For this reason, a key feature of this tool is to simplify and minimize the code and the screenshots. This is to ensure that all websites produce comparable results, thus minimizing unnecessary variability in their final appearance. However, this tool is highly flexible and can be customized by disabling specific features as necessary for alternative purposes.

The tool consists of three primary components, one for retrieving HTML code, another for fetching CSS code, and a third for capturing a screenshot of the website. Moreover, several utility functions are available to process the final output, including some for extracting statistics, some for relocating and rearranging the results, and a classifier for analyzing the screenshots and distinguishing good and bad results.

The diagram in figure 3.1 shows the step-by-step process of obtaining website code and screenshot.

1. Download of HTML file, which is then sanitized and cleansed.

2. The HTML file goes through a Web Framework detector that filters out files containing specific Frameworks.

3. CSS URLs are extracted from the HTML file, and CSS files are then downloaded, minimized, and merged.

4. A detector is used to exclude files that have zero CSS classes.

5. Processed HTML and CSS files are used to extract website screenshots.

6. The screenshots are labeled based on their quality by a classifier

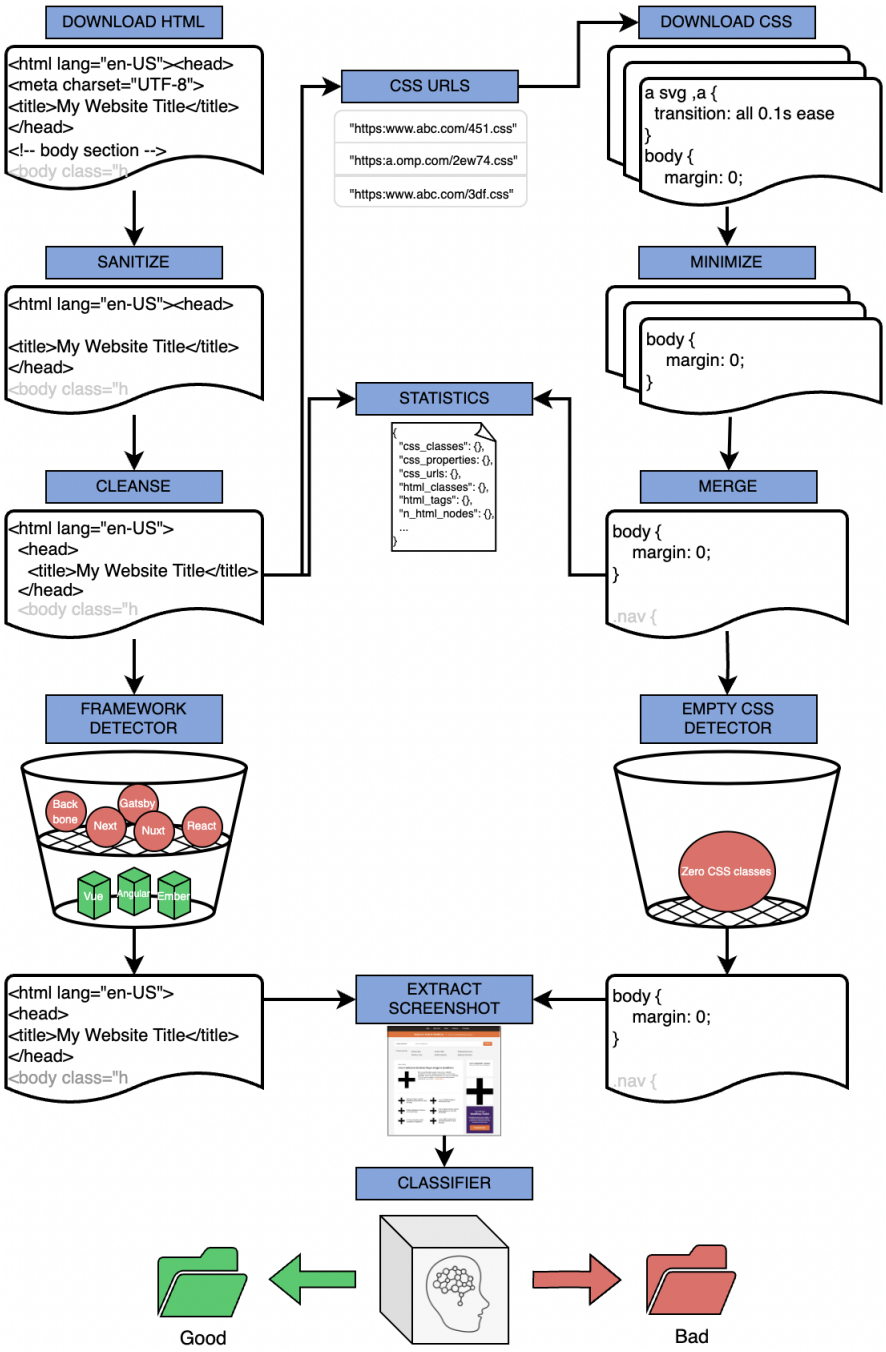

Figure 3.1: Diagram showcasing the website code and screenshot extraction pipeline.

3.2 Obtaining HTML code

The process of obtaining the HTML code of a website involves three distinct stages. Firstly, retrieving the HTML code rendered by the browser, secondly sanitizing the code, and finally, cleansing and formatting the sanitized code appropriately. With the term "sanitize", we refer to the process of removing unnecessary code lines, and also fixing code syntactical errors, like tags not closed or in the wrong position. "Cleansing" and "formatting" include the removal of comments, multiple white spaces or tabs, and adjusting the structure and layout of the code, making it uniform and more readable. This involves fixing the indentation, adopting a consistent use of quotes (single or double quotes), and aligning tag attributes. Additionally, an HTML parser is used to extract statistics such as the number of HTML nodes and the number of different HTML tags and classes.

3.2.1 Retrieve website's HTML code

For the first step, Selenium [62] is used, an open-source automated testing tool commonly used for web application testing. Selenium enables the automation of the process of interacting with a website and retrieving its HTML code. The chosen browser is Google Chrome, with a 1280x1024 window size.

3.2.2 Code sanitizing

To sanitize the code, the sanitize-html tool [63] is used. It is built on top of htmlparser2 [64] and effectively removes undesirable HTML code by eliminating tags specified in a deny list. It also corrects poorly closed tags and allows tag and attribute substitution.

Tags that do not affect website structure, are associated with external resources, or impact only the dynamic behavior of the websites are excluded, like <script>, <meta>, <noscript>, <svg>, <path>, and <iframe>.

Additionally, attribute substitutions are performed for tags including <img>, <href>, <picture>, <a>, <source>, <link>, , and <figure>. Specifically, "data-src" and "data-lazy-src" attributes are replaced with "src", and "data-srcset" and "data-lazy-srcset" are substituted with "srcset". All links to images are also replaced with a link to the default image within the project.

These HTML attributes are normally replaced asynchronously and are used to speed up website rendering and enhance user experience. Since in our scenario this is not needed, and images are substituted with a default one, we can substitute them during this phase. The HTML tags remain unchanged, while the attribute name is replaced according to the substitutions previously listed. The attribute value is left unaltered, except for the link to the default image change.

This enables each website to have a default image that can be used in place of the original images. This resolves the issue of resource downloads for each website. It will also provide a common appearance for images, facilitating their recognition in subsequent machine-learning tasks.

Another transformation used is the replacement of all <ol> tags with <ul> tags to minimize tag variability when there are no apparent structural differences.

3.2.3 Code cleansing and formatting

For the last step, a tool called clean-html is used. It cleans up HTML code, by removing comments, random line breaks, and mixed tabs. It also formats and indents code correctly.

We can see the result of the process after each step in Figure 3.2, where from left to right we have displayed the first 50 rows of the unprocessed raw HTML, sanitized HTML, and cleansed HTML (final result) respectively.

The first processing step is responsible for a major reduction in the line number through filters and transformations. In the final step, empty lines are removed. Each non-empty line can potentially generate one or more lines due to formatting, because it distributes one HTML tag per line, with a few exceptions.

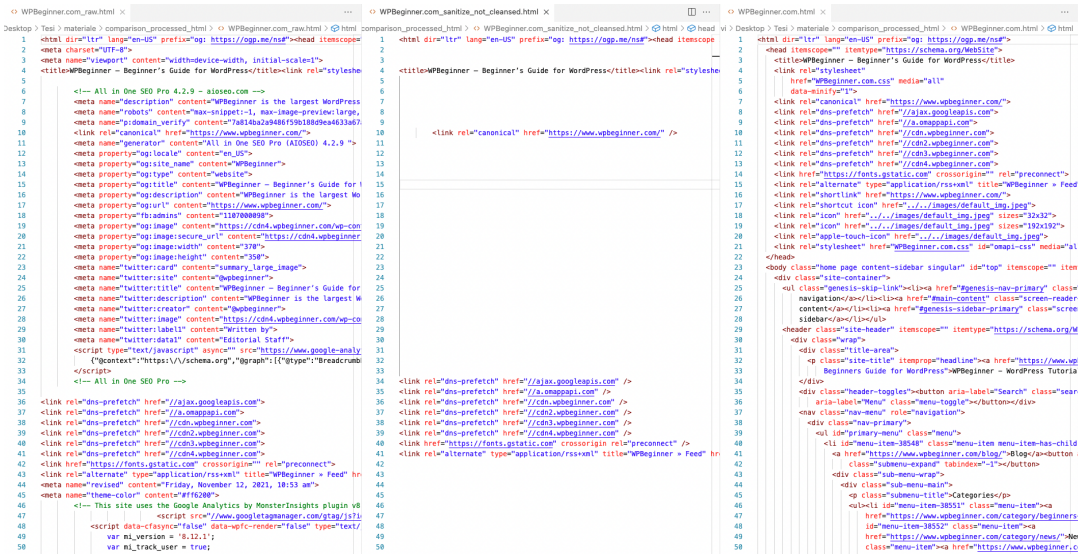

Figure 3.2: Comparison between unprocessed, sanitized and cleansed HTML code

3.3 Obtaining CSS code

The methodology for obtaining CSS code starts with getting all the CSS related to the website. Each of those is then processed individually, and cleansed and minimized according to some rules. In the end, the CSS files are merged, and their references are updated.

3.3.1 Get the CSS files related to the HTML file

The first step consists of searching for CSS file references in the HTML file obtained in the previous phase. Each file is then downloaded and processed.

3.3.2 CSS file processing

To process each CSS file, a custom-made parser was used. This parser is built on top of tinycss2 [65], which is a low-level CSS parser and generator, capable of processing CSS strings and returning CSS tokens and objects.

This allows for identifying all CSS components and recognizing CSS patterns such as qualified rules or at-rules. Each rule, based on the category is decomposed into different parts and recursively analyzed.

3.3.3 CSS file minimization

To minimize CSS code, the general idea is to remove all code that does not impact the website's appearance.

In fact, it is common practice to put all the style rules for all pages of a website inside one or more common CSS files, avoiding code duplication. However, in our scenario, we are interested only in the rules that affect the page rendered by the previously gathered HTML file. This means that, in many cases, CSS files can be reduced by a lot.

An even bigger reduction is possible when references to big CSS files from frameworks or libraries are present. This is because, usually, only a small portion of their classes are used. Some examples of those frameworks are Bootstrap [66], Tailwind CSS[67], and Bulma[68].

The strategy is to exclude the rules specified for tags or classes, which are not used in the HTML file. For this reason, a complete list of all the tags and classes used in the HTML file is extracted.

Moreover, CSS properties that have a small impact on the appearance of the resulting website screenshot are excluded too. These properties include those related to the website's dynamic behavior, and style properties that do not bring

structural changes. In addition, browser-specific CSS properties that are valid for other browsers but not for the one used in the experiment are excluded too.

Table 3.1 shows the lists of excluded properties, divided by type. Only a smaller portion of the Mozilla Firefox and Internet Explorer properties is shown for readability. The full list can be viewed in the code repository.

Experimental results show that the aforementioned measures result in an average size reduction of the number of lines in the output CSS file by a factor of 10.

type	properties
dynamic	transition, transition-timing-function, transition-delay, transition-duration, transition-property, animation-delay, animation, animation-direction, animation-duration, animation-fill-mode, animation-iteration-count, animation-name, animation-play-state, animation-timing-function
various	font-style, text-transform, letter-spacing, word-spacing, line-height, text-shadow, box-shadow, background-image, background-repeat, background-position, hyphens, border-radius, border-style, border-color, order-width, -webkit-font-smoothing
Mozilla Firefox	-moz-appearance, -moz-border-right-colors, -moz-binding, -moz-border-bottom-colors, -moz-box-align, -moz-border-left-colors, -moz-box-flex, -moz-border-top-colors, -moz-box-direction, -moz-box-shadow, -moz-box-ordinal-group, -moz-box-orient, ...
Internet Explorer	-ms-accelerator, -ms-behavior, -ms-block-progression, -ms-content-zooming, -ms-filter, -ms-flex, -ms-flex-align, -ms-flex-direction, -ms-flex-item-align, -ms-flex-line-pack, -ms-flex-order, -ms-flex-pack, -ms-flex-wrap, -ms-grid-column, ...

Table 3.1: List of excluded CSS properties divided by type.

3.3.4 Merge of CSS files

To simplify matters, a single CSS file is created by combining all the processed CSS files. Any references to CSS files in the HTML code are updated to point to this specific local file.

3.4 Screenshot extraction

To capture website screenshots Selenium [62] is used, with the same setup as when obtaining HTML code. Two possibilities exist: one connects the website URL and captures the screenshot, while the other (the one used in our experiments) loads the local HTML file and captures the screenshot, producing a website image representative of the processed HTML and CSS files.

Another useful feature is added to close the "accept cookies" pop-ups, which are common on many websites and usually occupy a significant portion of the resulting screenshot. This is particularly important in the first scenario, as in the second one, numerous pop-ups are eliminated due to removing during the sanitizing process of the <script> tag that typically contains them. This functionality simply attempts to locate buttons with common words to dismiss the popups, such as "I Accept", "Ok", and other variants and clicks on them.

Figures 3.4 and 3.5 show the results of the screenshot obtained for the website `WPBeginner.com`, using the two possibilities.

Figure 3.3: Image of black cross used as default for images substitutions.

We can notice how the images from the original website are replaced with the default image, the black cross in Figure 3.3. Some small details are missing in the processed version as a result of sanitization, but the overall structure is the same.

3.5 Collection of statistics

Various statistics are extracted for each website with the purpose of monitoring certain metrics that hold potential significance for the development of subsequent machine learning models or other relevant tasks. Statistics are saved in JSON files, one per website.

The dimensions of the recorded screenshot, including its width and height, are preserved alongside the count of lines present within the CSS and HTML files.

The number of nodes, CSS URLs, distinct CSS classes, and tags is extracted from the HTML files. Moreover, CSS files provide information about CSS classes and properties, as well as those excluded during minimization.

Statistics are collected also for the "raw" HTML and CSS files, those without the sanitizing and minimization process, to evaluate the impact of such procedures.

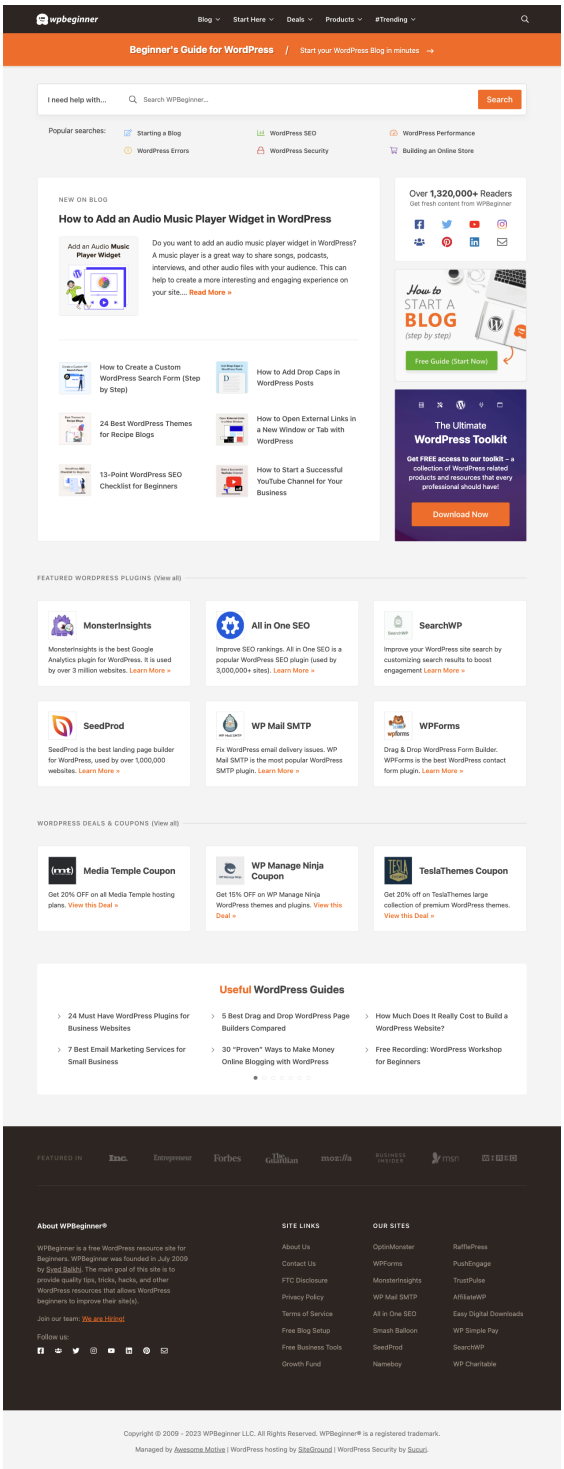

Figure 3.4: Screenshot of website before any processing.

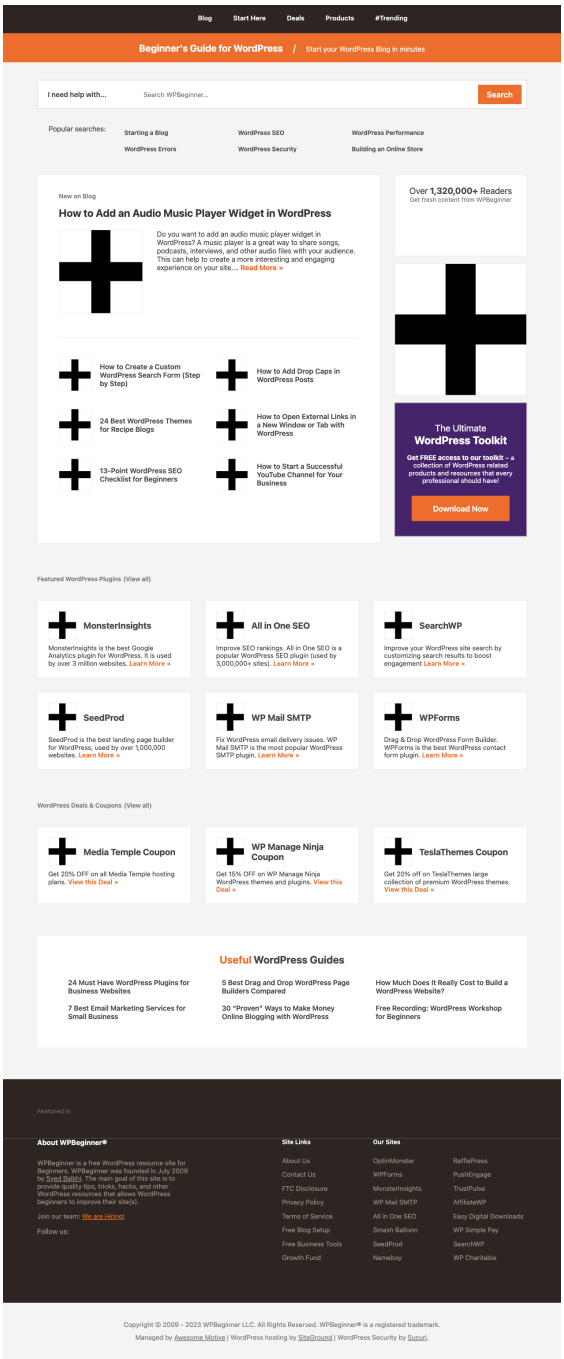

Figure 3.5: Screenshot of website after processing.

Chapter 4

Dataset creation

4.1 First experiment on blog websites

4.1.1 Experiment setup and evaluation methodology

To validate the process and evaluate the script's behavior, an initial experiment was performed on a limited number of websites. A set of blog websites was identified as suitable for this purpose, given their relative simplicity and standard appearance. Specifically, a list of 51 popular blog websites was obtained from the website `https://passionwp.com/most-popular-blogs/`.

Subsequently, each resulting website screenshot was reviewed and compared to the original website's appearance, without processing or minimization. Based on this comparison, a comprehensive list of observations was recorded, considering factors such as the degree of similarity between the processed screenshot and the original website, the identification of website frameworks, and the nature of the differences between the two versions.

The differences that are typical effects of HTML processing, such as image substitution, are not considered part of the abnormal differences.

A grade was given to each website from 0 to 5:

- 5 to websites almost identical to the original, and with minor differences
- 4 to websites similar to the original, with slight differences, or with differences in small parts of the website (ex: a list is different in a part of the footer)
- 3 to websites with a structure comparable to the original, but with some differences
- 2 to websites with large portions of the screenshot that do not reflect the original website, or with major differences

- 1 to empty websites, websites without styles, and websites completely different from the original ones
- 0 to websites with errors, that did not produce a final screenshot.

4.1.2 Framework detector

Based on the analysis of the first experiment, it was observed that some critical results that received a grade of 1 exhibited the presence of a web framework. As a result, additional system functionality was introduced to detect web frameworks. This is achieved by examining certain keywords and attributes in the website's HTML code.

The web frameworks that are considered include React[69], Gatsby[70], Next[71], Nuxt[72], Backbone[73], Vue[74], Angular[75], and Ember[76]. Table 4.1 shows the keywords searched for each framework.

By comparing the previously assigned grades of websites with the detected frameworks, it was found that only some of these frameworks consistently produced poor results, while three of them (Vue, Angular, Ember) did not. Therefore, if a framework from the remaining five (React, Gatsby, Next, Nuxt, Backbone) is present, the website is marked as "excluded".

4.1.3 Exclusion of websites without CSS

Another similar pattern was recognized among websites with a resulting CSS file with no CSS classes. For this reason, these are marked as "excluded" too.

4.1.4 Results

Upon removing the "excluded" websites (grade -1), only a few websites had bad results (grades 1, 2). Overall, 62.75% of websites had good results (grades 3, 4, 5).

After calculating the statistics on the good results, we can see some interesting trends, like the average reduction of lines of CSS code by over 90%, and a reduction of HTML lines by more than 35%.

4.2 Second and third experiment on a different list of websites

4.2.1 Majestic million list and second experiment

A second experiment was conducted on a portion of a larger list of websites, which could be used later in the final experiments at scale.

Framework	Keywords
React	data-reactid=".*?"
	React.createElement'
	ReactDOM.render'
Gatsby	gatsby-
	_gatsby
	GATSBY_.*_POST
Next	_app.js
	_document.js
	_error.js
	_documentSetup
	_appContent
	___NEXT_DATA___
Nuxt	nuxt-
	___NUXT___.js
	fetch___.js
	nuxt.js
Backbone	backbone−
	backbone.js
	backbone.min.js
Vue	vue−
	Vue.js
	Vue.min.js
Angular	ng-
	angular.js
	angular.min.js
Ember	ember-
	ember.js
	ember.min.js

Table 4.1: Keywords used to detect the presence of the different web frameworks.

The list used was Majestic Million [77], a list of a million website domains with the most referred subnets. The initial 100 websites were analyzed in this experiment.

The results of this experiment are worse than the first one, as was expected by introducing all kinds of websites, some more complicated than blogs. In particular, there is a significant increase in the number of websites with errors from 1 to 10, and websites with very low grades (white pages, websites without CSS).

The reason could be that, since these websites are more popular and drive

Grades	Total
0	1
-1	14
1	0
2	4
3	7
4	9
5	16
total	51

Table 4.2: Websites grades obtained during first experiment.

Averages	Raw	Processed
css classes	1788.03	143.39
css classes skipped	0	1596.25
css properties	220.31	78.42
css properties skipped	0	17.83
css urls	8.14	8.11
html classes	238.64	234.75
html tags	35.94	26.86
n html nodes	860.75	699.58
n lines css	20240.5	2255
n lines html	1542.25	996.08

Table 4.3: Websites statistics obtained during first experiment.

more traffic, they have additional measures to prevent web scraping. In addition, they are more sophisticated and complex overall. This is suggested by the error messages, that, in some cases mention the impossibility of making screenshots or the detection of automatic tools.

4.2.2 Third experiment on .blog websites from Majestic million list

At this point, the idea was to test the tool on another portion of the Majestic Million list. This was the first 100 websites with the .blog top-level domain. This was done to extract from the same list a sublist of easier websites, more similar to the ones used in the first experiment.

The outcomes demonstrate a marked improvement compared to the second experiment and are more in line with the first. The percentage of good website screenshots (grade 3, 4, 5) is slightly higher (72% versus 64.29%), but also in this

Grades	Total
0	10
-1	35
1	29
2	6
3	4
4	9
5	7
total	100

Table 4.4: Websites grades obbtained during second experiment.

case, the number of bad results, not excluded by the system's filters is not negligible (12%).

Grades	Total
0	7
-1	9
1	9
2	3
3	8
4	29
5	35
total	100

Table 4.5: Websites grades obtained during third experiment.

4.3 Introduction of the screenshot classifier and fourth experiment

Initial experiments indicated that the system performed better on simpler websites. However, the difficulty in obtaining large lists of simple websites led us to examine the problem from a different perspective.

Since most of the poor results are easily recognizable by a human and present common patterns, such as blank white pages or unstructured pages lacking CSS, the idea was to train a convolutional neural network to classify the results as either "good" or "bad", and filter out the second ones, similarly to the websites excluded during the previous phases by the framework detector and the detection of websites with zero CSS classes.

4.3.1 Dataset

A dataset is composed of previous experiments' results, which have been manually classified as "good" or "bad" and some of them have been removed since they are less easily identifiable than others. It contains 219 images, of which 112 are "good" and 107 are "bad". The dataset is almost balanced, with the first class containing approximately 51.1%.

75% of the dataset is used for training and validation, while 25% is for testing. The training-to-validation split is also 75:25.

Some samples from the dataset are shown in picture 4.1

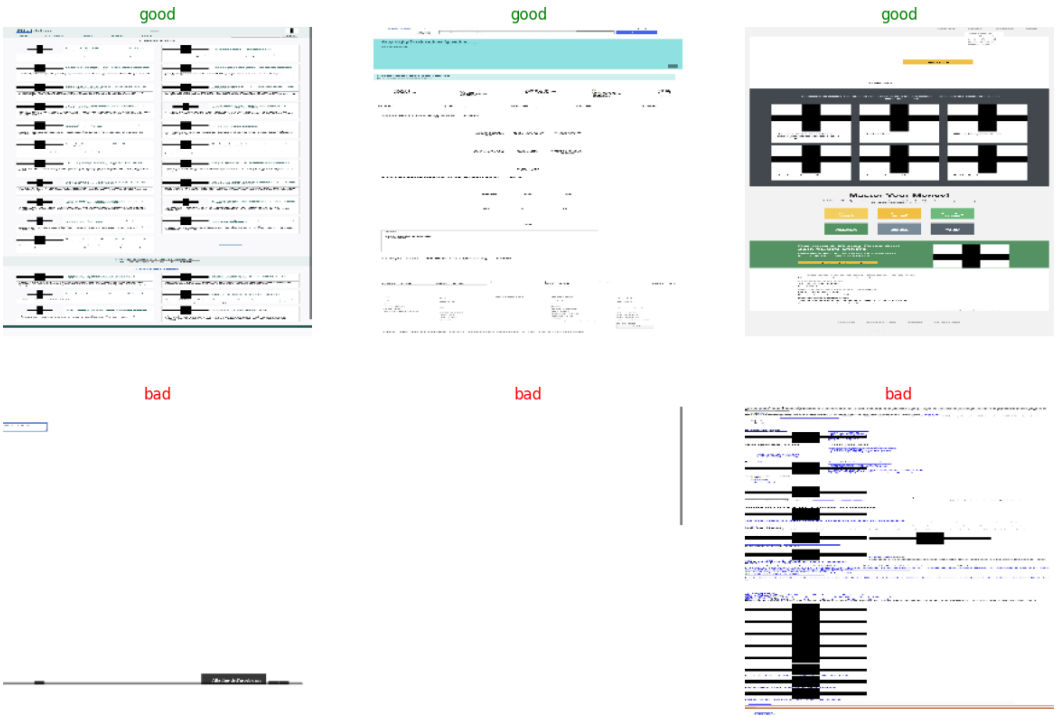

Figure 4.1: Samples from the dataset of the screenshot classifier.

4.3.2 Model

The model used is based on the ResNet50 architecture, a widely adopted neural network for image classification.

It is pre-trained on the ImageNet dataset, which contains millions of labeled images across thousands of classes. By leveraging these pre-trained features, the network is able to learn from a small amount of data and achieves high classification accuracy on new images.

The top layer of the ResNet50 model, responsible for the final classification task, is removed. New layers are added on top to fine-tune it for our specific classification task. Additionally, the model includes a dropout layer to reduce overfitting, a random-cropping layer for data augmentation, and layers to resize and scale the images.

The model is trained to classify images into "good" and "bad" using binary cross-entropy loss function and the Adam optimizer.

4.3.3 Training and testing

Several metrics were considered during the training of the model, namely loss, accuracy, precision, recall, and AUC. An "early stopping" strategy was used to avoid overfitting, monitoring the validation loss with a patience value set to 10 epochs. The model was trained for 30 epochs and reached a training accuracy of 83.74% and a validation accuracy of 87.70%.

As a comparison, the model without pre-training on ImageNet reached a training accuracy of 52.03%, and a validation accuracy of 51.22%, always predicting the second class.

This shows the inability of the model to learn from the small data at its disposal. It also shows the impact of transfer learning in a scenario with a scarcity of training data.

During testing, the model reached an accuracy of 81.82%, a precision of 80.00%, a recall of 85.71%, and an AUC of 85.19%. Table 4.6 and Figure 4.2 show the classifier results during training, validation, and testing and the confusion matrix on the test set.

	Training	Validation	Testing
Loss	0.332	0.354	0.703
Accuracy	0.837	0.878	0.818
Precision	0.812	0.864	0.800
Recall	0.889	0.905	0.857
AUC	0.935	0.946	0.852

Table 4.6: Screenshot classifier performance metrics.

4.3.4 Fourth experiment

A new experiment was performed to see the results after the screenshot classifier introduction. The list of websites analyzed comes from the second one hundred websites at the top of the Majestic Million [77] list.

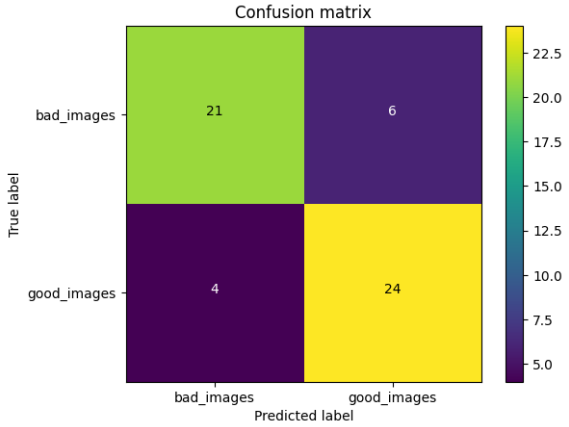

Figure 4.2: Confusion matrix illustrating the performance of the screenshot classifier on the test set by displaying the true and predicted classifications.

By merging the previous grades 3:5 into the class "good", and the grades 1:2 into the class "bad", it is possible to compare the results with the previous experiments.

	Blogs (1)	MM 1-100 (2)	MM.blog (3)	MM 101-200 (4)
Errors	1	10	7	20
Excluded	14	35	9	26
Bad images	4	35	12	25
Good images	32	20	72	29
TOT	51	100	100	100

Table 4.7: Table comparing results across different experiments.

Table 4.7 presents a comparison of the four experiments' outcomes.

The columns of the table correspondingly exhibit the results of the initial experiment performed on blog websites, the second experiment conducted on the first 100 websites listed in the Majestic Million [77] ranking, the third executed on 100 websites enlisted in the Majestic Million ranking with .blog domain, and, finally, the results obtained from the current experiment.

The results of this experiment are similar to the human-classified websites on the top one hundred websites of the Majestic Million list. Specifically, the proportion of websites retained (not excluded and without errors) was 54% (compared to 55% of human evaluation), with a higher proportion being classified as "good", i.e. 53.70% (versus 36.36% of human evaluation).

Again, the number of websites with errors is high, and the motivations are the same ones mentioned in the previous example.

Overall, the final result of this early experiment on a small list is that almost

30% of websites analyzed produce results classified as "good".

4.4 Final experiment on scale

A final experiment was conducted on a larger list, containing the top 100000 websites from the Majestic Million list. It was performed on the Politecnico di Torino Big Data Cluster [78], on a BigDataLab Education environment, with 30 GB of RAM reserved.

It lasted for about 3 weeks, and the 100000 websites were divided into 10 batches of 10000 each. The training of the screenshot classifier took approximately 10 minutes, while the main script ran for around 50 hours for each batch. The other minor scripts consumed a negligible amount of time, while the classification of the non-excluded websites took about 45 minutes for each batch of websites.

The experiment generated about 100GB of files, with the final dataset containing files of websites classified as good taking up 54GB of space.

The results showed similar numbers to the previous experiment in terms of the percentage of the "included" website. The percentage of errors increased from 20% to 29.74%, while the percentage of excluded websites dropped from 26% to 16.46%, and these two experiments somehow balanced the total number of not included websites at around 46%. From the included websites the percentage of them classified as "good" increased from 52.70% to 63.36%.

	Total
Errors	29736
Excluded	16459
Bad images	19716
Good images	34089
TOT	100000

Table 4.8: Results for the final experiment.

4.4.1 Statistics

The number of CSS files found in a random website from the analyzed list reproduces a decreasing exponential function, as shown in Figure 4.3. The average is around 7 files per website, and only less than 10% of the websites have more than 15 CSS files.

The average number of nodes in the processed HTML files is 1061.61%, with an average reduction of 11.34% during cleansing and sanitizing.

Averages	Raw	Processed
css classes	1965.54	139.92
css classes skipped	0	1775
css properties	172.55	67.34
css properties skipped	0	16.56
css urls	7.09	7.07
html classes	224.31	220.69
html tags	34.28	27.44
n html nodes	1197.39	1061.61
n lines css	23037.31	2264.85
n lines HTML	1794.84	1478.43

Table 4.9: Statistics on the extracted codes from the final experiment.

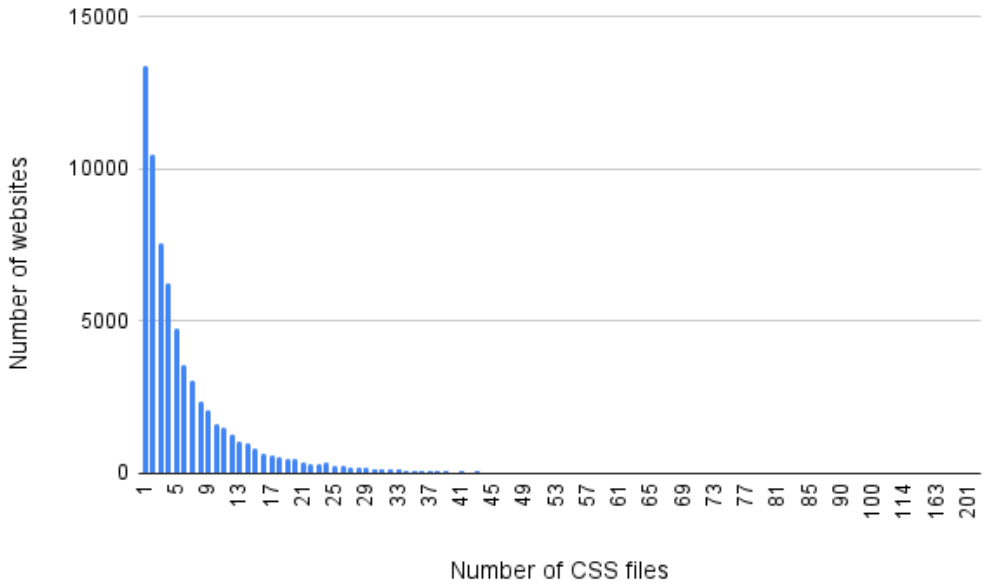

Figure 4.3: Distribution illustrating the number of CSS files associated with various websites.

The reduction of the number of lines in the CSS files before and after processing is around an order of 10, and it is quite consistent from small files to large files, as shown in Figure 4.5. The average length before processing is 23037.31 lines, and after processing is 2264.85 lines.

The impact of processing on the number of lines for HTML files is lower, with an average reduction of 17.63%, as shown in Figure 4.6 In particular, when the

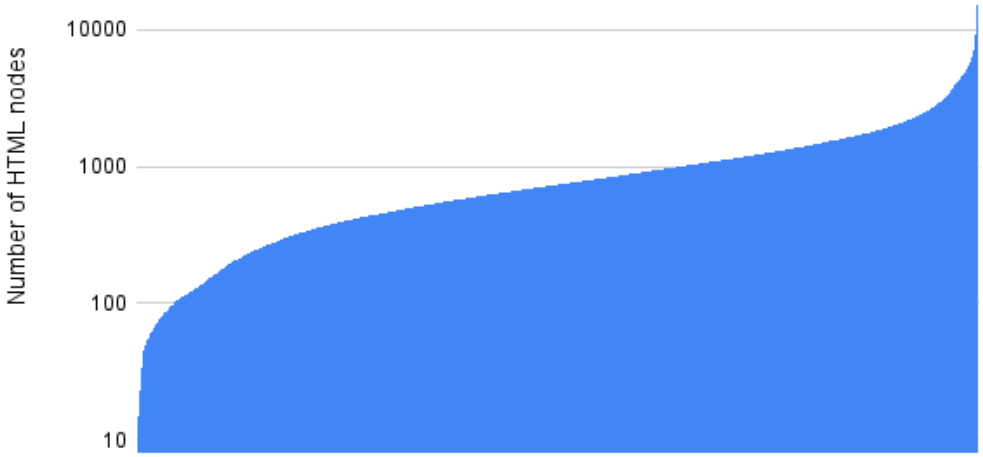

Figure 4.4: Distribution illustrating the number of HTML nodes associated with various websites.

number of lines is low, the number of lines in the processed file can be higher than the number of lines in the raw file. The reason for this behavior could be the addition of correct indentation and formatting of such files, where each line of the original file can generate one or more.

4.4.2 Errors

The experiment showed a significant percentage of websites that generated errors, about 29%. Further analysis was done to understand the causes of these errors and their nature.

Table 4.10 shows all the most common errors encountered during the experiment, with a percentage of occurrence greater than 0.5%.

The majority are related to connection issues or SSL certificates. They occur at the beginning of the experiment, during the connection to the target website to retrieve HTML code. These issues are often caused by firewalls or networking rules that prevent automatic tools from connecting.

The error "List index out of range" occurs during the CSS extraction phase, and is usually caused by using incorrect CSS syntax or invalid characters.

Two errors occurred during screenshot extraction. The first error with the message "Element click intercepted" is raised during the click on Cookies pop-ups,

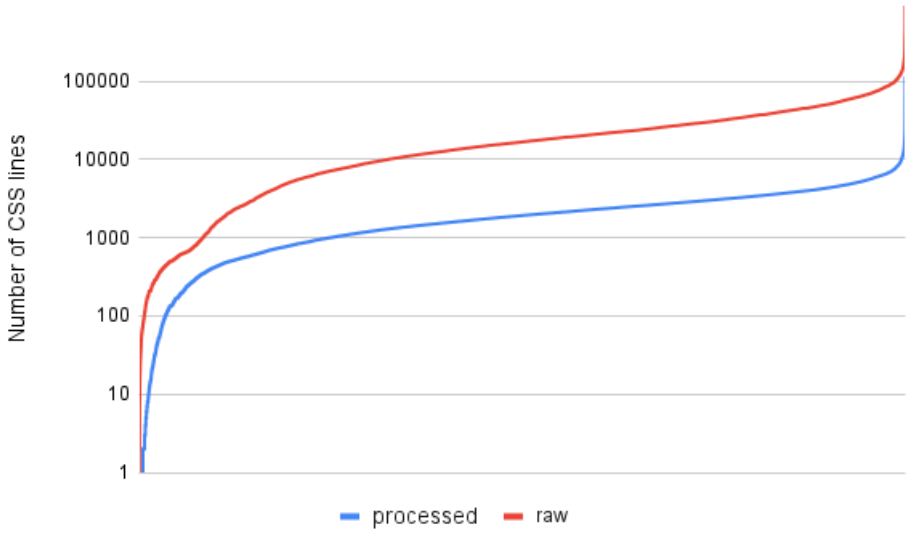

Figure 4.5: Comparative distributions (on a logarithmic scale) illustrating the number of lines in CSS files before and after processing, emphasizing the efficiency and impact of the processing step.

but it is not a blocking error, so the process continues after handling the error.

The second error is "Unable to capture screenshot", which can be due to various issues such as browser incompatibility, network issues, insufficient permissions, or timing issues.

Overall, many of the previous errors are inevitable mainly due to the nature of the experiment setup and the target websites that populate the target list. Some of these websites may be inaccessible to the public, while others may have sophisticated security defenses to avoid suspicious traffic. However, some other errors could be investigated more accurately and handled, like those related to CSS files.

Percentage	Type	Message
16.69%	ConnectTimeoutError	Connection timed out
13.29%	SSLError	<hostname> doesn't match <allowed hostnames>
13.17%	NewConnectionError	No address associated with hostname
8.14%		List index out of range
7.38%	NewConnectionError	Connection refused
5.29%		Read timed out.
5.05%		Timed out receiving message from renderer
3.92%	SSLError	Self signed certificate
3.80%	SSLError	Certificate has expired
3.79%	SSLError	Unable to get local issuer certificate
3.53%		Element click intercepted
2.90%	NewConnectionError	Temporary failure in name resolution
2.67%	UnknownError	Unable to capture screenshot
2.49%	NewConnectionError	Name or service not known
1.59%	ConnectionResetError	Connection reset by peer
0.98%	OSError	Connection aborted
0.78%	SSLError	Wrong version number
0.69%	SSLError	Alert internal error
0.66%	NewConnectionError	No route to host
0.65%		Remote end closed connection without response
0.50%	SSLError	Alert handshake failure

Table 4.10: Most common errors encountered during scraping of websites.

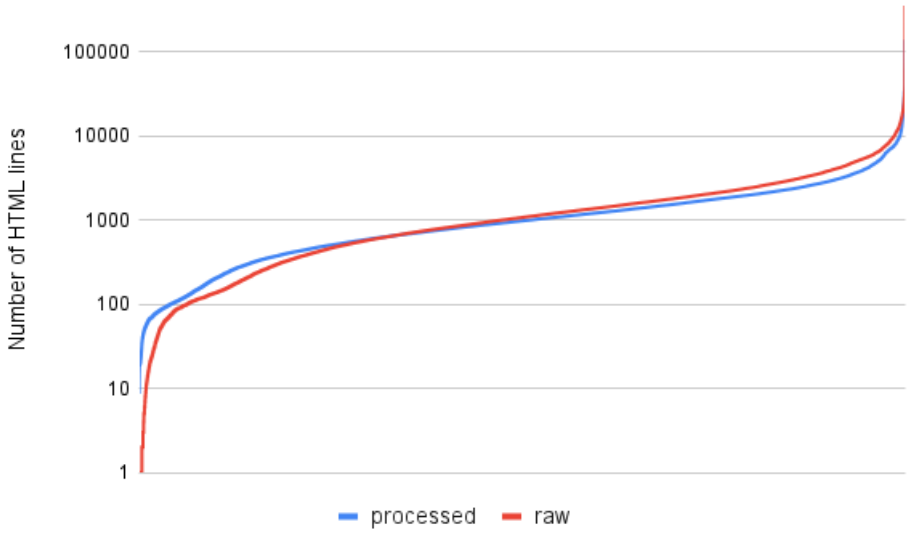

Figure 4.6: Comparative distributions (on a logarithmic scale) illustrating the number of lines in HTML files before and after processing, emphasizing the efficiency and impact of the processing step.

Chapter 5

Model for website code generation

5.1 Introduction

The recent months have marked a significant turning point in the field of generative textual models, with the initial recognition of success attributed to models like GPT[79] and BARD[80]. These transformer-based models laid the foundation for understanding context and generating coherent texts, proving effective in multiple tasks.

Progress has also been remarkable for specialized task-specific transformer models, including code interpretation and generation. As an example, GitHub Copilot[81], based on OpenAI's Codex model [82], can understand existing code and provide suggestions on new code lines. CodeGen[83], an autoregressive model for program synthesis, can automatically generate code based on descriptive prompts.

By integrating a visual component to extract features from images, multi-modal image-to-text models can be developed, thereby unlocking possibilities for various tasks. These range from image captioning and classification to visual question answering, among others.

Long Short-Term Memory (LSTM) models, a type of Recurrent Neural Network (RNN), process texts step by step, using the output from one step as the input for the next. This sequential processing enables them to retain a memory of previous inputs in the sequence. In contrast, transformer-based models feature an architecture that allows them to process inputs in parallel rather than sequentially. While they lack a built-in memory of previous inputs, they utilize a self-attention mechanism that enables them to weigh the importance of different words or tokens in the input data. This trait makes them more scalable and capable of efficiently handling longer sequences. Considering the nature of website codes, involving

extensive text and long-range dependencies between elements, we decided to apply transformer-based models to the context of website code generation.

All the successful models previously mentioned share one common characteristic: they possess a colossal internal architecture, containing hundreds of millions, if not billions, of trainable parameters. Given time and cost constraints, fully training a model of this type is not feasible in our setting. One viable approach is to fine-tune an existing model for a new downstream task.

In the context of machine learning and natural language processing, fine-tuning refers to the process of taking a pre-trained model and further training it to adapt to a specific task. The pre-trained model already learned from a larger, general-purpose dataset, and possesses foundational knowledge and patterns. By fine-tuning, the model adapts its pre-learned patterns to the peculiarities and specifics of the targeted task, thereby enabling enhanced performance with relatively minimal computational cost compared to training a model from scratch. It is important to avoid major architectural modifications to preserve the foundational knowledge of the model, obtained during the initial training. The loss of this general-base information can undermine the model's ability to apply it effectively to new specific tasks, obtaining poor results.

These considerations led us to choose Pix2Struct as a baseline model, which was fine-tuned for website code generation tasks.

5.2 Model: Pix2Struct

Introduced by Google Research in October 2022, Pix2Struct [84] is a pre-trained image-to-text model designed for understanding visually situated textual information. Ready to be fine-tuned for tasks involving text in images, this model has achieved state-of-the-art performance on various tasks across multiple domains, including documents, illustrations, interfaces, and natural images. Unlike its predecessors in visual language understanding tasks, Pix2Struct does not rely on Optical Character Recognition (OCR) systems but starts directly from the input image pixels. Furthermore, it is more general-purpose, not depending on task-specific metadata or utilizing other inputs or tools.

5.2.1 Model overview

Two variants of the Pix2Struct model are available: the Pix2Struct-Base, which has 282 million parameters, and the Pix2Struct-Large, with 1.3 billion parameters. This work utilizes the first version.

The model was pre-trained on 80 million screenshot web pages from the C4 Corpus[85], a large dataset created by researchers at Google, which includes a diverse range of internet text. Its pre-training process involved predicting the text

in randomly masked parts of the websites. The provided text is a simplified HTML version of the code, retaining only visible elements like texts and images filenames, or alternative texts. Any other structural information, such as element tags, style, titles, and URLs, is omitted.

The context in which this model was pre-trained appears particularly relevant to our task. The goal is to fine-tune the model to determine whether it can predict more complex code structures with syntactical accuracy. Moreover, it needs to predict the whole website code skeleton from scratch, starting only from its screenshot.

Pix2Struct is based on Vision Transformer (ViT)[86], a model that represented a notable shift in the field of computer vision by adopting techniques from the world of natural language processing. Unlike traditional convolutional neural networks (CNNs) which have been predominant in image processing tasks, the ViT leverages transformer architectures, which were originally designed for text data.

In ViT, each image is divided into fixed-size, non-overlapping patches, which are flattened and linearly transformed into vectors using a fully connected layer. A key aspect is the addition of positional embedding to the linear ones, compensating for the spatial information lost during flattening. The enriched vector is then fed into a standard transformer architecture [87].

In Pix2Struct, there is a modification to how the image is processed and the patches are extracted, which makes the model more robust to images of different aspect ratios. Rather than scaling the images to a predefined resolution before extracting the patches, the image is scaled up or down to allow for the extraction of the maximum number of patches while still maintaining the aspect ratio. Using a 2-dimensional positional embedding, the number of patches per row and column can vary, adapting to the original image shape. Figure 5.1 shows the comparison between the patches extraction process in the two architectures.

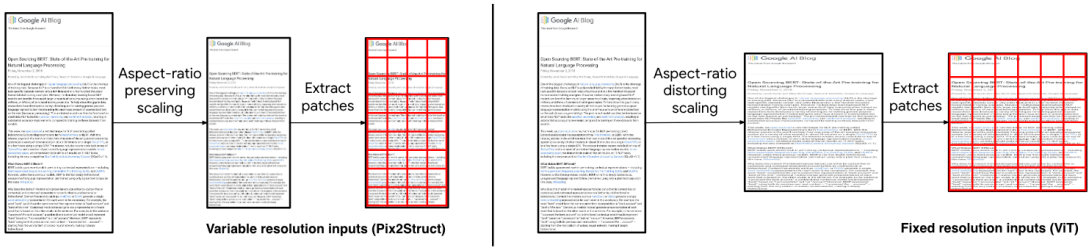

Figure 5.1: Comparison between variable resolution inputs (Pix2Struct) and fixed resolution inputs (ViT).
Source: `https://arxiv.org/abs/2210.03347`

The version of Pix2Struct used in this work is designed for conditional generation and is available from the Hugging Face Transformers library [88]. It has a language

modeling head, which can be used for sequence generation tasks. This encoder-decoder architecture consists of two main components: a vision model responsible for understanding the image and encoding its features, and a textual model that handles the translation of features extracted from the image into coherent textual output.

After the previously mentioned embedding, the Pix2Struct vision encoder introduces a dropout layer for regularization. Dropout is a technique that involves randomly disabling a fraction of the input units at each update during training time, which forces the network to learn more robust features and prevents overfitting. The encoder is then composed of 12 identical layers, each with multi-head self-attention and multi-layer perceptron (MLP) sublayers. Each layer presents also a residual connection, followed by layer normalization.

Self-attention is a key mechanism in transformer-based architectures. It enables the model to encode each input element while considering other parts in the sequence, capturing dependencies among them. Specifically in visual transformers, this mechanism allows the model to integrate information from different image patches when encoding each one, thereby understanding spatial and contextual relationships within the image.

The textual decoder of the model consists of 12 identical layers, each comprising self-attention, cross-attention, and MLP sub-layers. The self-attention mechanism operates similarly to its functionality in the encoder. However, it now incorporates a mask to ensure the network attends only to preceding tokens when processing a new one, thereby preventing it from accessing future tokens it is tasked to predict. On the other hand, cross-attention facilitates the decoder's focus on pertinent sections of the encoded image during textual generation, enhancing its capability to generate coherent and contextually relevant text. Moreover, each sub-layer presents layer normalization and residual connections, alongside a dropout mechanism.

Figure 5.2 displays the high-level schema of Pix2Struct model.

5.2.2 Comparison with other models

As mentioned in this chapter's introduction, the availability of a pre-trained open-source model becomes a necessity in our setting. The Hugging Face transformers library [88] provides numerous implementations of transformer-based models, along with their pre-trained weights. The unified, straightforward interface to these models allows easy access to them, ensuring reproducibility and consistency. Several textual and multi-modal models are available, and there is the possibility to combine independent visual and textual components to create multi-modal encoder-decoder architectures.

Several models were considered for their unique characteristics. For example, CodeGen [83] is a family of large language models, with configurations up to 16.1B

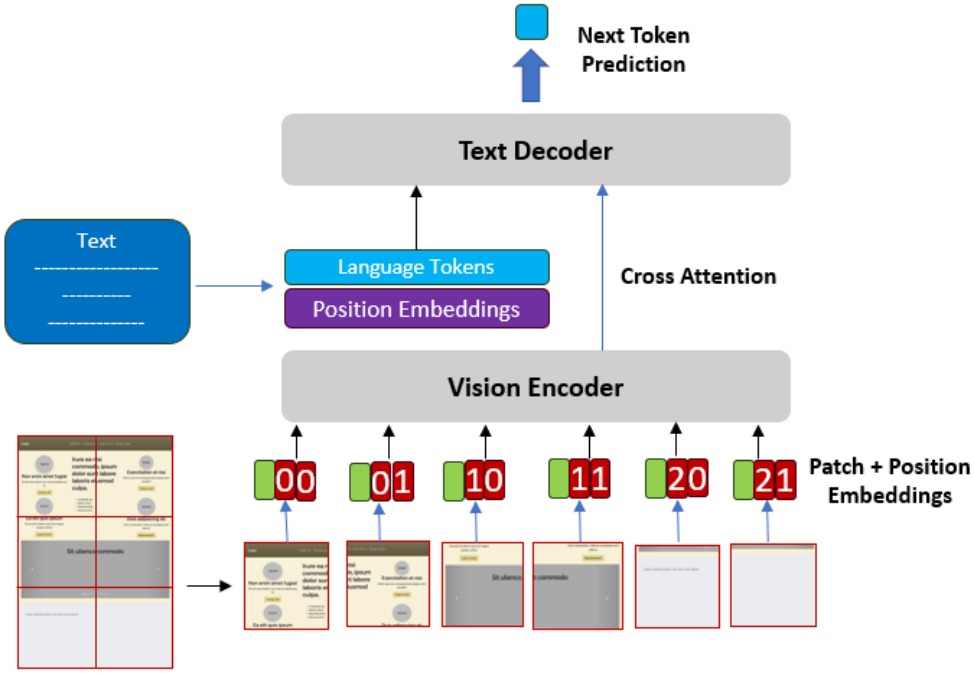

Figure 5.2: Pix2Struct model schema: an image is processed through a Vision Encoder, after dividing it into patches with positional embeddings. Together with text embeddings they are passed through a Text Decoder, with a Cross Attention mechanism linking visual and textual data, leading to the prediction of the next token in the sequence.

parameters, trained on a corpus of natural language and programming language data. Its performance on code synthesis tasks can have an advantage in a context like Web code generation, compared to other general-purpose models. Exhibiting competitive capabilities in code synthesis tasks, particularly in zero-shot Python code generation as assessed on HumanEval, CodeGen may present advantages in our Web code generation context compared to other general-purpose models. The model notably excels in multi-turn program synthesis, where a single program is divided into multiple prompts, each specifying subproblems, thereby augmenting both the efficiency and precision of the program synthesis process. Nevertheless, this diverges from our multimodal context, where the input is a website screenshot, casting uncertainty on the model's suitability for our task.

Longformer [89] presents an element that would suit our limited-resources scenario. It is a textual model designed to handle long text sequences, which are problematic for traditional transformer models, due to their quadratic computational complexity with respect to input length. Unlike standard transformers, which compute attention over all pairs of input tokens, Longformer utilizes a sliding

window attention mechanism that computes attention only over a limited number of neighbor tokens. This reduces the computational complexity to linear with the sequence length. It combines this sliding-window attention mechanism with global attention for the tokens that require looking at all the other tokens.

Regarding the visual component, the benefits of using a model that preserves various image aspect ratios seem crucial, especially when dealing with website screenshots. This makes choosing Pix2Struct, over alternatives like ViT, a clear decision. Additionally, the convenience of using the Pix2Struct model for Conditional Generation as a unified solution, without needing to couple it with a separate textual component, affirmed our choice. The model's pre-training is notably relevant to our context, and its established proficiency in predicting diverse structural texts contributes to its preference over other models, like those mentioned before.

5.2.3 Addressing model challenges

The primary challenges stem from limited available resources, including GPU RAM, time, and costs. All experiments were run on Google Colab [90], with limited access to Tesla T4, V100, and A100 GPUs. Some initial experimental tests were executed to identify the most suitable GPU between the three. In a trade-off between performance and costs V100 was revealed to be the best for sampling purposes, with similar performance as the A100 but with almost a third of the costs. A100, however, was clearly superior during training, not only for its speed-up but also for allowing longer sequences and multiple data in parallel.

The batch size could be increased to a maximum of 4 before exceeding the memory capabilities. Since this value is relatively low compared to the usual ones used in this kind of experiment, and because batch size is important for stabilizing the learning process and exploiting computational resources, a gradient accumulation strategy was employed. Gradient accumulation allows the model to be trained with an effective batch size that is larger than what the hardware could handle per single forward and backward pass, without requiring additional computational resources. When a batch of samples is passed through the network and the loss and gradients are calculated, the model parameters are not immediately updated; instead, the gradients are stored. The values accumulate over several "mini-batches", and once the desired batch size is reached, they are used to update the model parameters. This strategy simulates having bigger batch sizes, by delaying the model updates. In our setting, a value of 8 was used, doubling the batch size that is physically allowed, and updating the model parameters every two passes.

With 40 GB of available RAM, much higher than 16GB of V100, the A100 GPU is capable of handling sequences with a maximum length of 1024 tokens. This proved to be sufficient for simpler datasets but was not enough for datasets with

HTML codes, especially the more complex ones.

For this reason, a sliding window approach was utilized. The fundamental concept involves dividing a larger sentence into chunks of a fixed maximum size, in this case, 1024. By incorporating an overlap between these chunks, the model can comprehend the context from the preceding segment and continue predictions for subsequent words. This necessitated a modification in the data passed to the model. Specifically, the model's textual decoder now needs to receive the last tokens of the previous chunk to maintain context and understand at which point in the generation it is situated. A modification to the decoder's attention mask was required to ensure it could perceive these tokens.

Figure 5.3 illustrates an example of the sliding window mechanism, with a sentence length of 4096, a chunk length of 1024, and an overlap of 256. Each chunk is composed of 256 tokens from the previous one and 768 new tokens. An exception is the first chunk, which has 1024 new tokens. This configuration was used in some of the experiments and resulted in a maximum of 5 windows per sample.

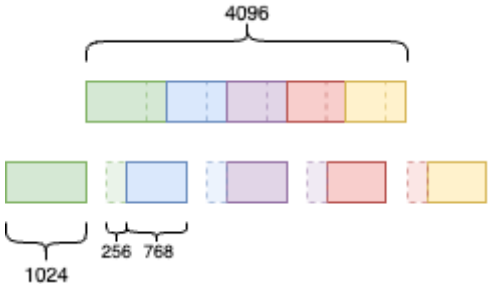

Figure 5.3: Illustration of the sliding window mechanism applied to a sentence with a length of 4096, using chunks of 1024 and a context of 256, yielding 5 distinct windows.

The term "maximum" is used because an optimization strategy was introduce to avoid processing empty chunks for shorter samples. During the construction of the training dataset, the textual component of each sample undergoes processing, and the tokens are extracted. Instead of padding each sentence to the maximum length and loading them directly, while managing the sliding window logic inside the training loop, the sentence is immediately divided into chunks. This approach is feasible because the context length and overlap are already known. Empty chunks are discarded, and the data loader is now populated with the chunks, not with whole samples. Each entry also has an annotation indicating its position because the first chunks need to be treated differently, as they don't have overlapping parts at the beginning. Additionally, each chunk has a reference to the index of the image associated with the sample. The index is utilized to prevent excess memory consumption, ensuring the image is stored in memory only once.

This strategy is not possible during evaluation, as the sample must be fully reconstructed to save the final prediction and calculate the metrics.

The model begins by generating text with a maximum length equal to the chunk size. The last portion of the prediction is used as an overlapping context for the next generation iteration. This process continues until all iterations are completed, going through all the possible sliding windows. After each generation step, the overlapping context is truncated from the predicted sentence, and the predicted chunk is concatenated to the preceding ones. In the case of the first iteration, since context is not present, it is not removed, and the prediction is saved as is. An optimization is performed here by marking in a mask which sentences in the batch are already finished before all iterations are completed. When all the data in the batch are marked as finished, or when the maximum number of iterations is reached, the generation loop ends, and the predictions are ready to be evaluated.

5.3 Metrics

Several metrics are collected to measure the model's ability to predict website code starting from a screenshot. Most of the textual ones are calculated starting from the unprocessed version of the answer and its prediction. In other cases, like Structural BLEU, or HTML tree edit distance, it is required to have fully compiling code and the texts go through post-processing steps to guarantee it. During post-processing correction, the encountered errors are saved so that they can be analyzed later. The codes are then rendered by a browser and the screenshots are collected, to calculate the image similarity metric.

5.3.1 BLEU

BLEU[91], which stands for Bilingual Evaluation Understudy is a metric for evaluating machine-translated text. It provides a score that measures how many words and sentences in a machine prediction match with a reference one, taking into account both precision and recall. Since its introduction, it has been widely used by the machine translation community but has also been employed in various other natural language processing tasks, including those of language models for text prediction.

BLEU provides a single score, making it easy to understand and communicate results. It doesn't require human evaluators, making it a fast and cheap automatic metric. It considers overlapping n-grams (sequences of n words) of varying lengths in the generated and reference text. For language modeling tasks, this means BLEU can help evaluate how well the model predicts both individual words and longer sequences or structures in the language.

One limitation of this metric in the context of code prediction, is that code is highly syntactic. A missing or misplaced character can render code non-functional. BLEU, which primarily measures n-gram overlaps, might not emphasize these crucial syntactic distinctions sufficiently. Moreover, multiple code snippets can correctly perform the same function but look very different syntactically. In this scenario, they would obtain a very low value, despite both being a valid solution for the problem.

Additionally, all the words inside the text have the same impact on the BLEU score. This aspect differs from code programs, where structural keywords and constructs have a much more important role than textual elements. For this reason, I introduced an additional metric called "structural BLEU". This metric calculates the BLEU score on a modified version of the prediction and answer by removing non-structural elements such as texts, titles, and button texts.

The implementation of the BLEU score from the Natural Language Toolkit (NLTK) [92] is used, with "Smoothing method 4" as the smoothing function. Since shorter translations may have inflated precision values due to having smaller denominators, this function is used to give them proportionally smoothed counts.

5.3.2 Edit Distance

Edit distance is the Levenshtein edit distance between two strings or texts, in our case prediction and answer. This distance is the number of characters that need to be substituted, inserted, or deleted, to transform the first string in the second one.

For example, transforming "rain" to "shine" requires three steps, consisting of two substitutions and one insertion: "rain" -> "sain" -> "shin" -> "shine". These operations can be done in different orders, but at least three steps are needed. This metric is implemented using NLTK [92] version, setting the cost of all three operations to one.

As an additional metric, the normalized version of edit distance is also used, dividing it by the maximum number of characters between the answer and the prediction.

5.3.3 HTML Tree Edit Distance

For the same motivation that led me to introduce the "structural BLEU" metric, I decided to introduce a new distance metric, more focused on structural elements. In particular, the HTML code is represented with a tree, whose nodes are its constitute tags. Then, the tree edit distance between the answer HTML tree and the prediction tree is calculated using the Zhang-Shasha algorithm, whose revision was presented by Paassen[93].

Zhang-Shasha, introduced in 1989, is an efficient dynamic programming algorithm computing the tree edit distance. Tree edit distance is the minimum number of node deletions, insertions, and replacements that are necessary to transform one tree into another. Beautiful Soup parser is used to extract the nodes to recursively create the HTML tree and Tim Henderson's Python implementation of Zhang-Shasha algorithm[94] to calculate the final metric.

Additionally, the normalized version of the tree edit distance is also used, after dividing it by the highest number of nodes in the two trees.

5.3.4 Structural Similarity Index

As a metric to measure visual similarity between the resulting website screenshots, the Structural Similarity Index (SSIM)[95] is used. SSIM evaluates the structural similarity between two images. It takes luminance, contrast, and structure into account and outputs a score between -1 and 1, with 1 indicating two identical images. Compared to other methods like Mean Squared Error (MSE), which estimates absolute pixel-level errors between two images, the concept of structural information focuses on the strong inter-dependencies between pixels, particularly when they are in close proximity. These dependencies convey crucial information about the structure of objects within a visual scene. Scikit-image[96] implementation of structural similarity index is used, which outputs also an image with the full SSIM map and an image of its gradient.

The biggest drawback of this metric is that it requires the two images to have the same size, so a resizing is needed. After this resizing the texts and the components of the image can be moved by some pixels, and this would be recognized as a change in the SSIM index calculation. This behavior is shown in figure 5.4, which corresponds to a final SSIM index of 0.88.

5.4 Datasets

5.4.1 Pix2Code Dataset

Introduced by Beltramelli in 2017[4], the Pix2Code dataset is a relatively small and simple collection containing screenshots of various web and mobile applications. For each image, there is a corresponding Domain Specific Language (DSL) code, which represents the structure and elements of the UI, such as buttons, text fields, and images. This intermediate representation is used to facilitate the translation from image to actual usable code, such as HTML or Android XML. A sample of the DSL code and its corresponding HTML code is illustrated in Figure 5.5.

The dataset consists of three parts, for Android, iOS, and Web user interfaces respectively. The latter is the one taken into consideration for this scope. The web

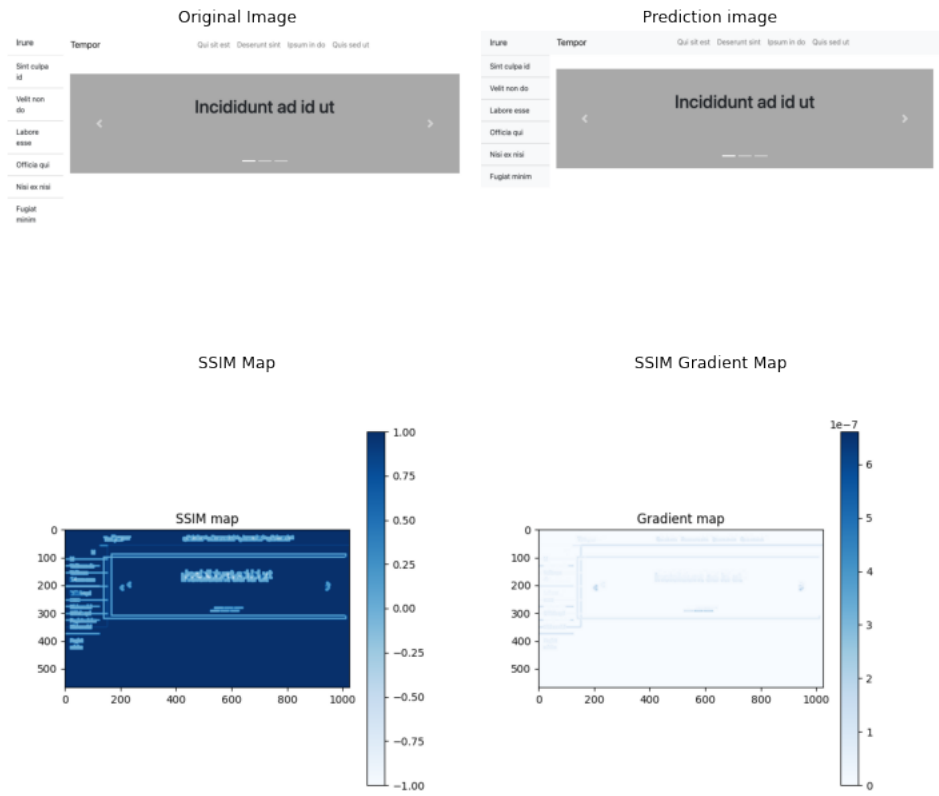

Figure 5.4: Example of the resulting Structural Similarity Index map and gradient images.

portion of the dataset consists of 1742 samples, identified as the original 1750 used in the Pix2Code[4] experiments after the removal of erroneous samples, performed later on by the author.

The availability of only 12 distinct structural elements limits the size and simplicity of the dataset. Also, the sample's internal variability is restricted, as the website codes consist only of the mentioned elements and curly brackets to specify their internal hierarchy. No other texts or elements without structural impact are present.

The elements "small-title", "text", and "quadruple" are used the most, accounting for almost 50% of all elements in the entire dataset codes, as shown by Figure 5.7. Each of the first two is used on average more than 6 times per website, which is significant since the average number of elements per website is only 32. The smallest website code comprises just 8 elements, while the largest has 56. The number of lines in the sample DSL codes ranges from a minimum of 8 to a maximum of 45,

```
header {
btn-inactive, btn-active, btn-inactive, btn-inactive, btn-inactive
}
row {
quadruple {
small-title, text, btn-orange
}
quadruple {
small-title, text, btn-red
}
quadruple {
small-title, text, btn-green
}
quadruple {
small-title, text, btn-orange
}
}
row {
single {
small-title, text, btn-green
}
}
```

```
<div class="header clearfix">
  <nav>
    <ul class="nav nav-pills pull-left">
      <li><a href="#">Tb Blrurys</a></li>
      <li class="active"><a href="#">Jkm Mtpkvu</a></li>
      <li><a href="#">Yxuvjyh Hg</a></li>
      <li><a href="#">Mueq Qvnkm</a></li>
      <li><a href="#">Rr Rwealkl</a></li>
    </ul>
  </nav>
</div>
<div class="row">
  <div class="col-lg-3">
    <h4>Jmpen</h4>
    <p>ffnj jvazv bnbxktwbqqktxzjnikfs axq qhv yoyz itaajgryrw</p>
    <a class="btn btn-warning" href="#" role="button">Rwivbjy Ya</a>
  </div>
  <div class="col-lg-3">
    <h4>Cggek</h4>
    <p>xdqejabv nrlnwiua mmjutjyaafe hqnqlqwokkkf ezy zcku ldc</p>
    <a class="btn btn-danger" href="#" role="button">Phoo Oetdk</a>
  </div>
  <div class="col-lg-3">
    <h4>Szrwh</h4>
    <p>kggnoatkvkk ez mxrvwlyty qlkikk pnbikxj zy zblgirnslpfs</p>
    <a class="btn btn-success" href="#" role="button">Yxbute Euj</a>
  </div>
  <div class="col-lg-3">
    <h4>Xqhhx</h4>
    <p>mks oonwdkpvifkfrzxbjjqpssqui i kobg hshc axinzxrmuzwtqs</p>
    <a class="btn btn-warning" href="#" role="button">Qrckl Lhfk</a>
  </div>
</div>
<div class="row">
  <div class="col-lg-12">
    <h4>Atbgk</h4>
    <p>nqojy veodjgedv ygqjie rcd bnk ivqrhjyahioeo scstzivgyy</p>
    <a class="btn btn-success" href="#" role="button">Tdxasff Fa</a>
  </div>
</div>
```

Figure 5.5: Comparison of DSL code and the corresponding HTML code for a Pix2Code sample.

with a median of 30 lines.

Using a compiler, it is possible to retrieve the HTML code corresponding to the simplified DSL code. This result can be rendered by a browser to obtain the same user interface represented in the screenshot.

For this reason, another version of the original dataset was used, with the samples consisting of HTML code and screenshots. Since the compiler that generates the HTML code picks random characters to form the words to put in titles, buttons, and paragraphs and this process is not deterministic, the resulting websites have a different text compared to the original sample screenshots. To solve this issue, new screenshots were taken from the newly generated HTML codes.

Additionally, a new version of the dataset was created, by substituting the random texts with random words taken from the Lorem Ipsum placeholder text, limiting the number of different words used and recognized. This was revealed to be a significant factor in the capability of the model to recognize not only the words but also the overall interface structure.

These two versions differ only in the internal texts, and the number of different tags is limited to 17, as shown in figure 5.8. The average number of lines goes to 64.28 and 58.11 for the original HTML and HTML Lorem Ipsum versions respectively.

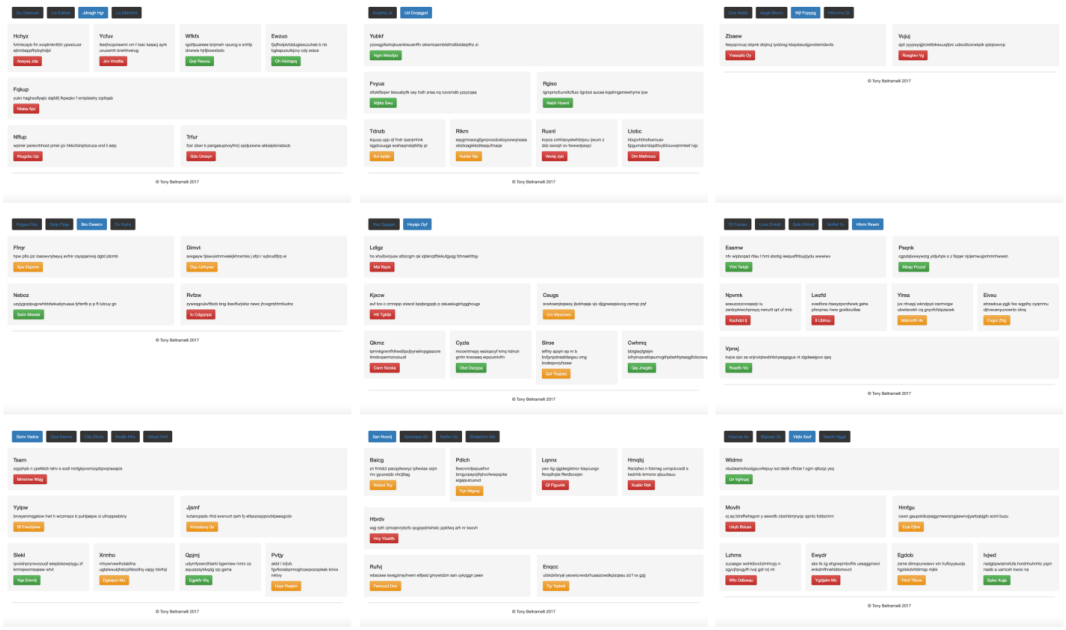

Figure 5.6: Nine representative screenshots taken from the Pix2Code Dataset.

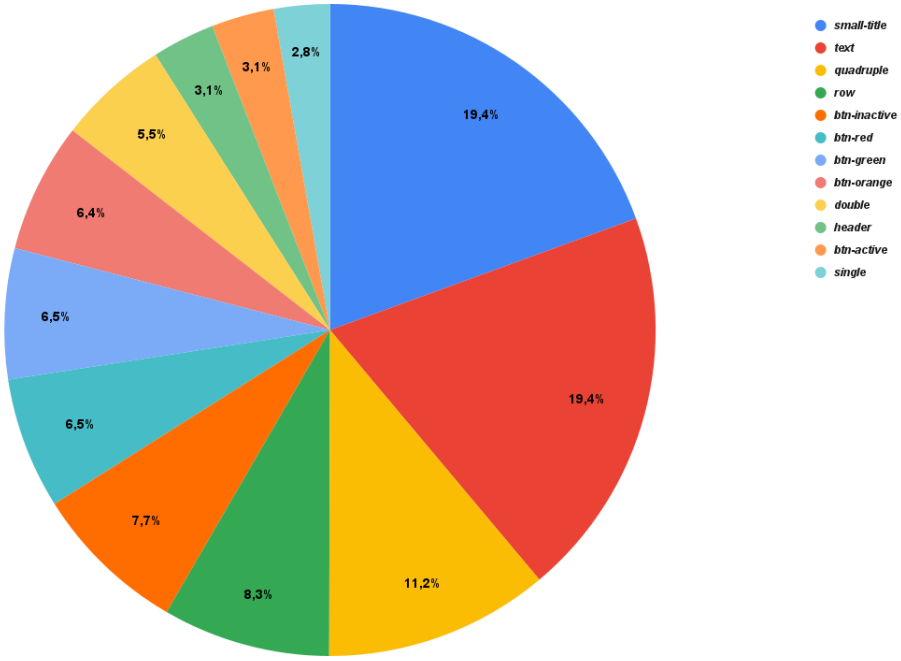

Figure 5.7: Pie chart visualizing the frequency distribution of different elements used in the Pix2Code DSL codes.

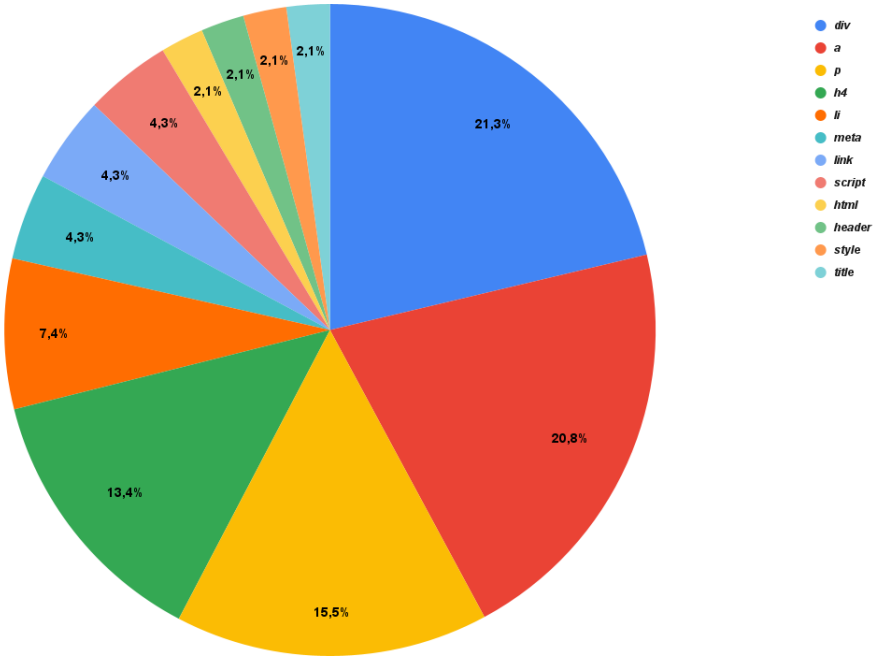

Figure 5.8: Pie chart visualizing the frequency distribution of different elements used in the Pix2Code HTML codes.

5.4.2 Synthetic Bootstrap Dataset

As an intermediate step between simple datasets and a real-world dataset, a more complex synthetic dataset was created. This was achieved using an open-source software for the creation of synthetic web-based UIs, called WebGenerator [97]. It is a script-based tool capable of generating single-page websites built on Bootstrap [66] framework. It can also generate colored websites, using random palettes and changing accordingly the CSS file referenced in the HTML code. The chosen colors are annotated also at the beginning of the HTML files, making it possible to have every information in a single file and generate the CSS file with the correct colors in a second moment, before taking the screenshot. Allowing processing in batches, it is possible to easily build a big dataset using this tool, obtaining at the end a set of HTML codes and the corresponding website screenshots, extracted using Selenium [62].

It is possible to generate websites with various visual elements, like cards, placehoders, tables, navigation , carousels, forms, and others. To populate the elements' textual components, random Lorem Ipsum sentences are generated and used. The probability of obtaining different kinds of websites can be controlled with a set of parameters. Table 5.1 shows those parameters and the values that were used for the dataset generation. In particular, layouts refer to four different

websites structures, with different positioning and extension of elements, such as lateral navigation bar. In our case, the probabilities of these layouts were set to be equal to 0.25 for each one. The choice was done empirically, by looking at the resulting websites during early experiments and trying to mimic the aspect of most common websites.

Name	Description	Value
with_sidebar_p	Probability that the sidebar is present	0.7
with_header_p	Probability that the Header is present	0.5
with_navbar_p	Probability that the Navbar is present	0.8
with_footer_p	Probability that the Footer is present	0.6
layouts_p	Probabilities for each layout.	0.25 each
boxed_body_p	Probability that the page's Body is boxed inside a container	0.0
big_header_p	Probability of having a big header (> 50% of the screen height)	0.0
sidebar_first_p	Probability of the Sidebar being at the left side of the Body	0.8
navbar_first_p	Probability of the Navbar being above the header	1.0
bg_color_classes_p	Probabilities for the combination of CSS Bootstrap's background color classes.	Default

Table 5.1: Generation probabilities of website characteristics for WebGenerator tool

The result is a dataset composed of 50,000 samples, each one consisting of a screenshot of the website and its HTML code. Figure 5.9 displays the screenshots of some samples from the generated dataset, while Figure 5.10 illustrates the HTML code related to the first sample of the previous image.

Note the <meta> element at the beginning with the annotation of the color palette used for the website.

During the experiments, only a smaller portion of this dataset was used with 10000 samples, due to resources limitations. Additionally, a "mini" version with only 1000 samples was used in one experiment for a comparison of the results.

Compared to the Pix2Code Dataset in the HTML version, the number of distinct HTML elements almost doubles, from 17 to 32. Figure 5.11 shows that and <a> are still the most used in the dataset with around 41% of the whole number of tags being these two. The maximum number of lines is 264, while the smallest website has 28 lines and the average is 104.

A new variant of this dataset was created having sketches of website components

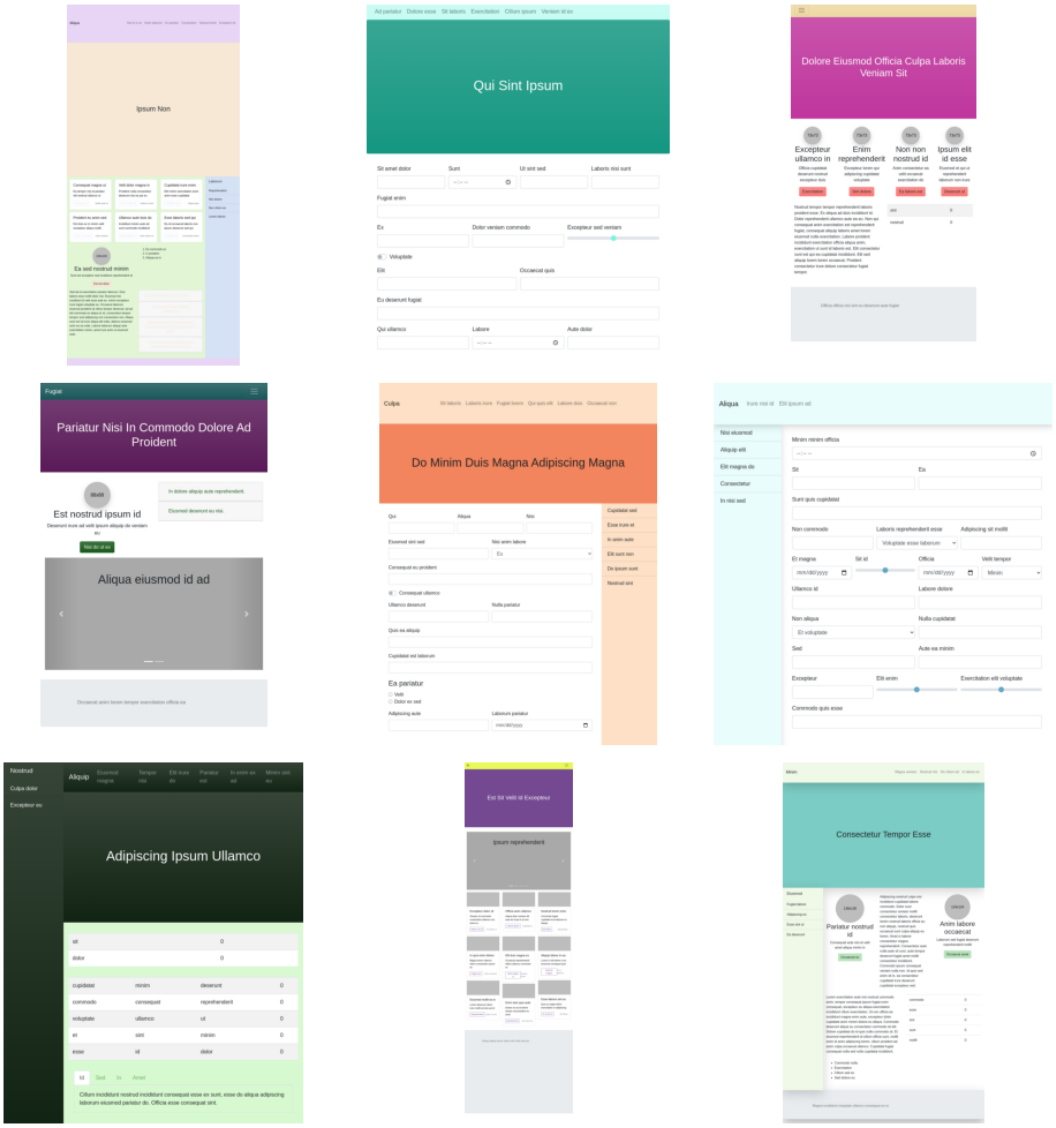

Figure 5.9: Nine representative screenshots taken from the Synthetic Bootstrap Dataset.

instead of the normal Bootstrap elements. This was done by identifying the area that each real element occupies and substituting it with a sketched version of it. The creation of this dataset started from scratch because it required black-and-white websites.

The result is a dataset of almost 10000 websites (9789), whose HTML codes are created in the same way and with the same parameters as the original Synthetic Bootstrap dataset. Figure 5.12 shows some samples of this dataset.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Dominate</title>
    <meta content="{&amp;quot;primary&amp;quot;: &amp;quot;rgb(203, 146, 26)&amp;quot;, &amp;quot;secondary&amp;quot;: &amp;quot;rgb(151, 59, 28)&amp;quo
    <meta content="width=device-width, initial-scale=1.0" name="viewport">
    <meta content="Web Generator" name="author">
    <meta content="1" name="wg-layout">
    <link href="../css/custom-bootstrap.css" rel="stylesheet">
    <link href="../css/wg-extras.css" rel="stylesheet">
    <script src="../js/jquery-3.2.1.slim.min.js" type="text/javascript"></script>
    <script src="../js/bootstrap.min.js" type="text/javascript"></script>
    <!--Layout: 1-->
  </head>
  <body class="">
    <div class="d-flex " id="full-wrapper">
      <div class="w-100">
        <div class="container-fluid py-3" id="page-content">
          <div class="row my-3 wg-detect" data-wg-type="mix">
            <div class="col">
              <table class="bg-white table-striped table" data-wg-type="table">
                <tr>
                  <td>officia</td>
                  <td>lorem</td>
                  <td>0</td>
                </tr>
                <tr>
                  <td>nostrud</td>
                  <td>ex</td>
                  <td>0</td>
                </tr>
              </table>
            </div>
            <div class="col">
              <div class="text-center wg-detect" data-wg-type="featured_item">
                <span class="text-black rounded-circle mx-auto  placeholder" style="width: 102px;height: 102px;">
                  <span>102x102</span>
                </span>
                <h2>Consectetur laborum</h2>
                <p>Ipsum reprehenderit veniam id sit ea enim sint est</p>
                <a class="btn btn-primary" href="#">Ipsum sed et</a>
              </div>
            </div>
          </div>
        </div>
        <footer class=" shadow jumbotron mb-0" data-wg-type="footer" style="height: 22vh;">
          <div class="container">
            <span class="text-muted">Culpa ipsum in dolore proident irure eu et ut id</span>
          </div>
        </footer>
      </div>
    </div>
  </body>
</html>
```

Figure 5.10: Code of a sample taken from Synthetic Bootstrap Dataset

5.4.3 WebUI2Code Dataset

"WebUI2Code" refers to the dataset created in chapter 4, using the tool for scraping website codes and screenshots from the web. Compared to the previous synthetic datasets, this dataset contains much more complex and diverse data, directly extracted from real-world websites. This dataset includes 34089 samples, each comprising an HTML file, a related CSS file, and a screenshot. Additional files,

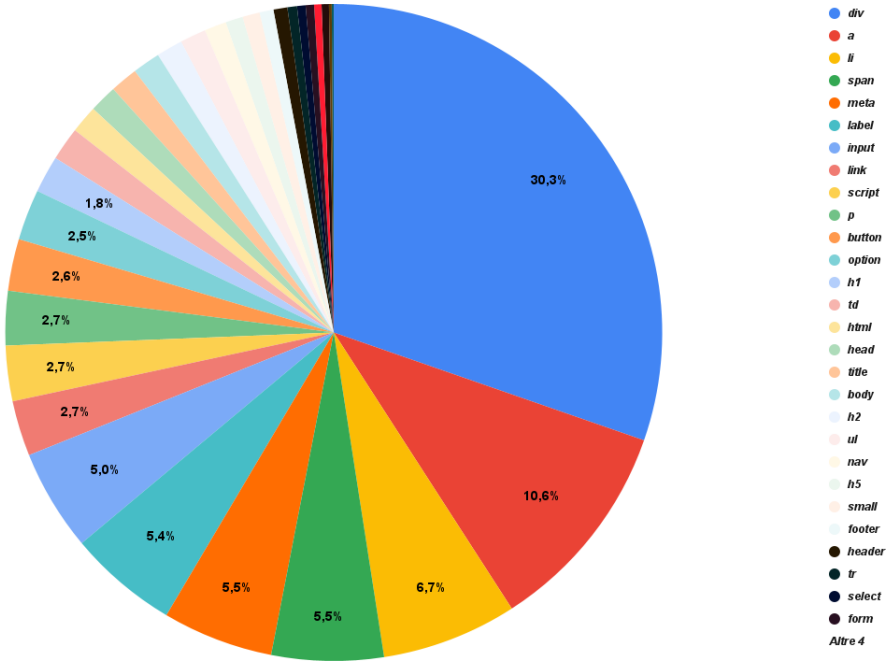

Figure 5.11: Pie chart visualizing the frequency distribution of different elements used in the Synthetic Bootstrap HTML codes.

	Pix2Code	P. HTML	P. HTML LI	SynthB.	SketchSB.
count	1742	1742	1742	50000	9789
mean	27.07	67.07	61.00	104.00	104.00
median	30.00	64.28	58.11	109.03	108.19
std	6.51	9.09	7.81	35.72	35.38
min	8	36	34	28	32
max	45	104	82	264	258

Table 5.2: Statistical analysis of the line count across various versions of the Pix2Code and Synthetic Bootstrap Datasets.

that are not essential for the following experiments are present, such as a JSON file containing information and statistics from the scraping process, and the raw HTML and CSS files.

An initial analysis of the dataset revealed its variety and diversity exceed what is suitable for our experimental scenario, particularly given our limited resources. GPU memory requirements, time, and cost consumption scale with the length of the processed texts. Even after pre-processing, it became evident that managing the entire dataset in our setting was not feasible.

The average number of lines in the entire dataset is 2327.52, more than 20 times

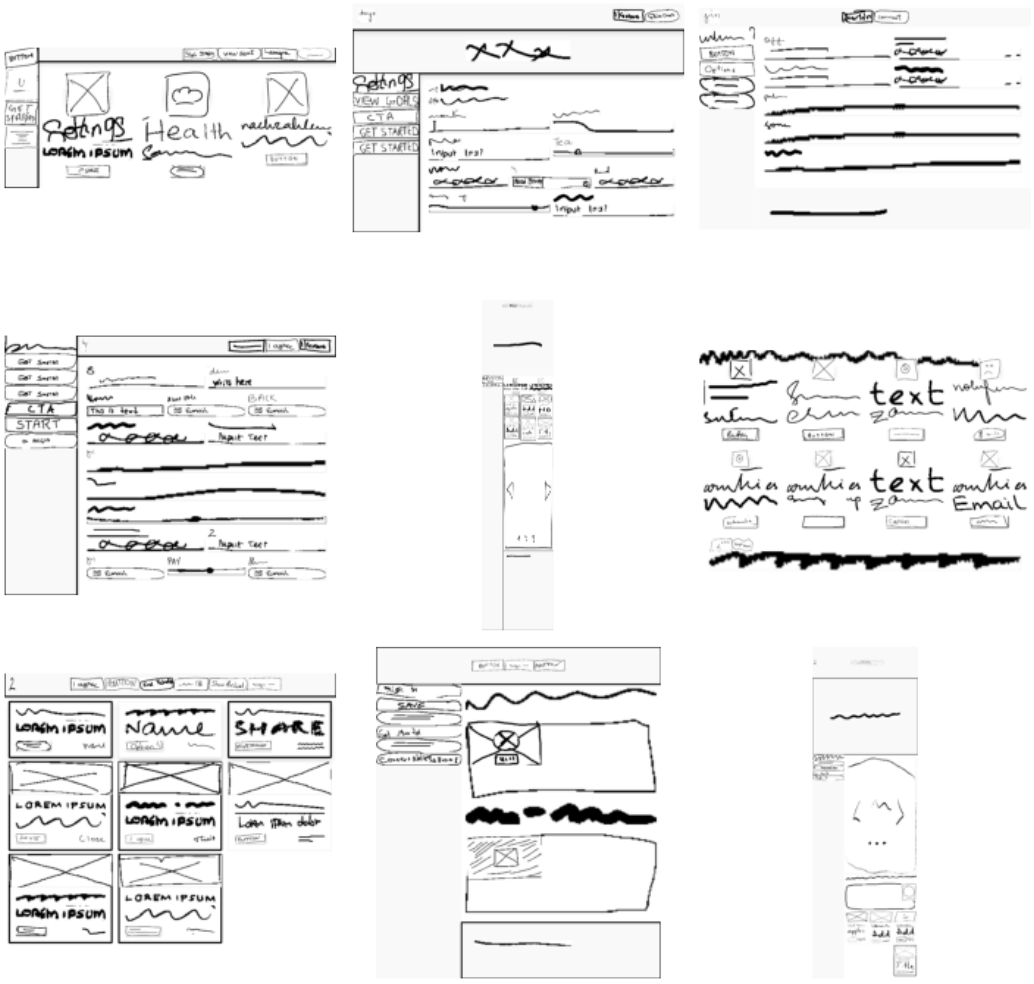

Figure 5.12: Nine representative screenshots taken from the Sketch Synthetic Bootstrap Dataset.

larger than the largest of the previous ones, which is 104.00, as shown in Figure 5.2.

Table 5.3 displays the number of samples from the dataset that remain below-specified token thresholds. This is determined by tokenizing the pre-processed code samples with the model's processor and recording the resultant token counts. The Pix2Struct model utilizes the T5 tokenizer [98], which converts input texts into tokens, the smallest units processed by the model. These tokens can represent words, subwords, or characters, depending on the training data and tokenization strategy. The first table column reports different values as thresholds, starting from 4096, which is the value used in the experiments on other datasets, and continuing with its multiples. The second column provides information on how many samples

finished under that threshold. The third column aggregates this value with those of the smaller thresholds, indicating the total number of samples available if that threshold is chosen. The last column provides the number of windows necessary to process files with that number of tokens, assuming a maximum sentence length of 1024 and an overlapping context of 256, as these values resulted in the maximum capability of our system.

Thresholds	**# samples**	**# aggr.**	**# windows**
< 4096	2442	2442	5
< 8192	1906	4348	11
< 12288	2170	6518	16
< 16384	2355	8873	21
< 20480	2466	11339	27
< 24576	2486	13825	32
< 28672	2275	16100	37
< 32768	2091	18191	43
< 36864	1896	20087	48
< 40960	1629	21716	53
All	12373	34089	

Table 5.3: Breakdown of websites from WebUI2Code Dataset by token thresholds, with cumulative aggregation and required window counts for each threshold.

Two thresholds were chosen for our experiments: 4096 and 16384. The first one matches the threshold used for other datasets; however, the number of samples is limited to 2442. The second threshold is a compromise between the number of samples and the number of sliding windows, with 8873 and 21, respectively. Larger thresholds were not analyzed because they are not usable in our setting. Different versions of the dataset were created, each identified by the threshold number.

The dataset showcases a notable diversity with 5354 distinct HTML tags, a stark contrast to other synthetic datasets, which peak at 32 tags. Figure 5.13 presents a pie chart, showcasing the most frequently used tags along with their respective percentages. To streamline model training, it was imperative to curtail this number. Consequently, through a detailed methodology outlined in the subsequent pre-processing section, the total was reduced to 130 varied HTML tags.

The first version of the dataset that was analyzed includes the samples under 4096 tokens, with a starting number of 2442 samples. An initial study on the screenshots' dimensions and on the number of CSS and HTML lines was performed. In particular, the width and height of images are important because they significantly impact image size, which can create problems during the opening and processing of the images. Moreover, when textual files consist of a limited number of lines is often

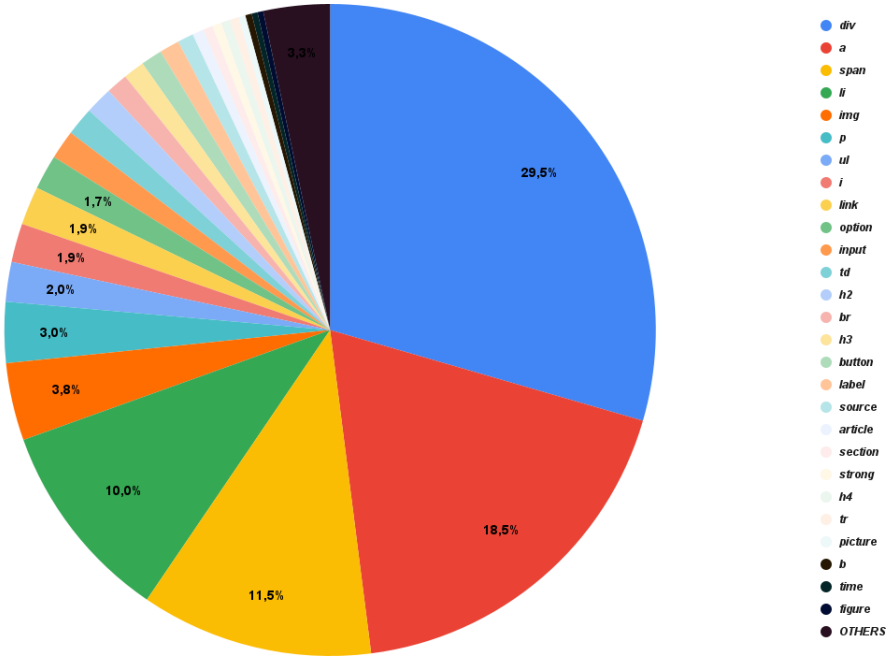

Figure 5.13: Pie chart visualizing the frequency distribution of different elements used in the WebUI2Code HTML codes.

a symptom of having encountered problems during content extraction. This was discovered when analyzing CSS files with very few lines, which usually contained the word "error". Small HTML files, on the other hand, can be considered outliers in this dataset, since they have almost no information inside them, given the long nature of HTML language and the formatting performed after code extraction.

Figure 5.15 illustrates the distribution of widths, heights, and aspect ratios of images within the WebUI2Code-4096 dataset version. Most images showcase a width ranging between 1100 and 1500 pixels, with the narrowest being less than 300 pixels wide, and the widest reaching up to more than 5000 pixels. The height distribution is even more varied, including outliers extending up to almost 40000 pixels. The majority of the files exhibit an aspect ratio (width/height) between 0.4 and 1.4. Figure 5.14 presents a scatter plot of images widths and heights. Each point in the plot symbolizes a sample in the dataset. From this analysis and through the manual evaluation of the quality of screenshots with varying heights, sizes, and aspect ratios, it was decided to establish certain thresholds and exclude all the samples falling outside of them. The maximum allowed width was set to 2,000, the minimum width to 250, the maximum height to 8,000, and the minimum height to 250, all while maintaining a minimum aspect ratio of 0.10 and a maximum of 2. These limits do not dramatically reduce the intrinsic diversity of the dataset but aim to remove the outliers and problematic images.

Figure 5.16 shows the distribution of the number of lines in the samples' HTML and CSS files. The scatter plot in Figure 5.17 highlights that files with a large number of CSS lines typically do not have a high number of HTML lines, and vice versa. This is expected since all the files populating this dataset are constrained by the number of tokens being lower than 4096. These tokens are extracted from the text obtained through concatenating the HTML and CSS files. Several files are characterized by a very low number of CSS lines, while this is true for a lower number of HTML files. For the reason previously expressed, a threshold was set to 10 as the minimum number of CSS lines, as well as the minimum number of HTML lines, and files that do not meet those limits are excluded.

The number of samples that did not meet one of the images' constraints was 69. 55 of the samples did not meet the minimum number of CSS or HTML lines, with 52 failing due to CSS and 3 due to HTML. Considering some files were excluded for violating both rules, the final number of samples excluded was 120, leaving 2322 samples in the 'cleaned' version of the dataset. Figures 5.18 and 5.19 display the distributions of image sizes and text line counts, respectively, in the final version of the dataset after the exclusion procedure.

For brevity, plots of the other portions of the dataset are not presented, however here are some general considerations:

- The distribution of widths in the datasets with longer websites is shifted to the right compared to the simpler dataset. The majority of the websites' screenshots have still a maximum width of 2000 pixels, with outliers that reach up to 14,000.
- In terms of screenshot heights, a bigger number of websites do not respect the threshold, having a bigger height due to the bigger number of texts inside the website. The outliers' maximum height scales almost regularly, passing from 40000 to 60000, to 80000 to 100000, in the 4096, 8192, 12288 and 16384 versions, respectively.
- Compared to the first dataset, the other versions have an increase in the percentage of websites with small aspect ratios, between 0 and 0.5. This is correlated with the bigger heights of the websites, caused by the higher number of words and elements.
- The number of CSS and HTML lines increases coherently with the number of tokens in the files in the dataset. While in the bigger datasets, the number of files with few HTML lines is reduced, there is always a significant amount of websites with few CSS lines.

In version WebUI2Code-8192, out of the initial 1906 websites, 243 fail to meet the size threshold, 53 do not satisfy the number of lines constraint, and 26 violate both conditions. In total the exclusions are 270, leaving 1636 websites.

Version WebUI2Code-12288 begins with 2170 websites, excluding 407 for not meeting the size threshold and 60 for the number of lines threshold, with a total of 428 exclusions, since some of them were excluded twice, and a final count of 1742 websites.

WebUI2Code-16384 starts with 2355 samples, excluding 604 for size and 48 for the number of lines, totaling 624 exclusions with 24 websites excluded twice, and resulting in a final dataset of 1731 websites.

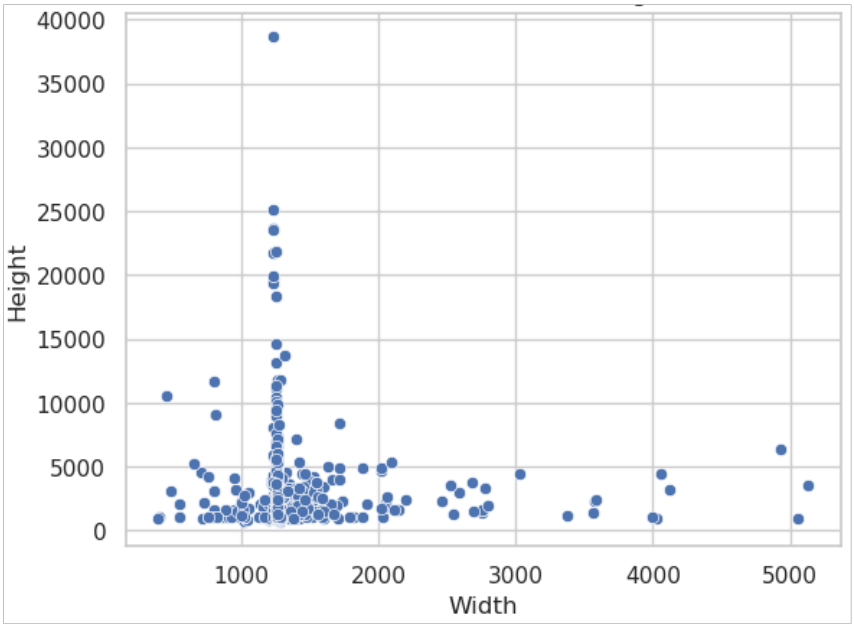

Figure 5.14: WebUI2Code-4096: Scatter plot illustrating the distribution of image dimensions, comparing widths to heights, prior to data cleaning.

Figures 5.20, 5.21, 5.22, and 5.23 display samples from the four portions of the dataset, with thresholds of 4096, 8192, 11228, and 16384, respectively. These images illustrate how an increase in tokens correlates with a rise in the amount of text present on the website screenshot, leading to overall more complex websites.

5.4.4 Rico Dataset

To expand our collection of real-world interface datasets, we have included mobile UI datasets, the first example is Rico. Rico is a dataset composed of more than 66 thousand Android user interfaces, mined from over 9000 free mobile applications. The dataset consists of several components, each offering a unique lens through which to explore UI design and interaction:

- UI Screenshots and View Hierarchies: involves original UI screenshots alongside their respective view hierarchies.
- UI Metadata: encompasses metadata related to the UIs, such as app name, alongside user interaction traces, providing a glimpse into user navigation and interaction.
- UI Layout Vectors: provide 64-dimensional vector representations of each UI screen. These encode layouts by discerning the distribution and arrangement of text and images.
- Interaction Traces: categorizes interaction traces by app, with each app potentially having multiple traces. Each trace is a sequence of UIs, captured as both screenshots and view hierarchies.
- Animations: offers GIFs illustrating screen animations in response to user interactions, visually showcasing UI dynamics.
- Play Store Metadata: contains Google PlayStore metadata linked to each app, enhancing contextual understanding.
- UI Screenshots and Hierarchies with Semantic Annotations: contains UI screenshots and hierarchies with semantic annotations, indicating the meanings and usage of screen elements.

The part of the dataset utilized in our experiment is the first one. The view hierarchy is described in a structured JSON format, representing a recursive hierarchy of UI components. All elements in the UI can be accessed by traversing the hierarchy, starting at the root node. Each element presents the "class" property, which specifies the component class name, and the "children" property, which offers a recursive representation of the sub-components. Additional information is provided for each node, such as bounds, ancestors, and properties of the element, including whether it is clickable, visible, or focused.

In our experiments, we extracted only the class names from the hierarchy and organized them in a structured way by separating them with curly brackets when they included children. This procedure is described more in detail in the pre-processing section. This representation is similar to Pix2Code DSL codes and simplifies the texts. A comparison between the original layout view hierarchy code and the resulting simplified structural code can be seen in Figure 5.24

Figure 5.25 displays nine samples' screenshots from the Rico dataset.

Figure 5.26 illustrates the distribution of the number of classes per sample within the dataset, revealing that the majority of samples include fewer than 100 classes. A minority portion of samples encompass up to nearly 500 classes.

A comprehensive analysis across the entire dataset identified a total of 34142 distinct classes. The most utilized classes are depicted in Figure 5.27. In all the samples combined, there are almost one million and a half classes. "widget.LinearLayout" is the most used, averaging more than 20 repetitions per sample. The next three more common are "widget.AppCompatTextView", "widget.RelativeLayout", and "widget.TextView" with around 10 repetitions per sample.

To constrain the class quantity, we tried applying a threshold, dictating the minimum usage frequency required for a class to be retained and deemed "valid". Given the challenge of substituting "excluded" classes, the approach involved testing different thresholds, identifying valid classes that adhere to the threshold requirement, and subsequently omitting all samples that incorporate classes not present in the valid list.

Figure 5.28 shows how changing the threshold value impacts the number of classes and the final number of filtered samples. As a good compromise between the number of classes and the number of samples, a threshold equal to 2000 was chosen. This leads to a total of 131 classes and 12480 files.

5.4.5 UI2Code Dataset

UI2Code is a large dataset of Android UI screenshots and GUI skeletons comprising more than 185277 pairs, extracted from 6000 mobile applications. The collected GUI codes use 291 unique Android GUI components, including Android's native layouts and widgets and those from third-party libraries. The distribution of the components across the whole dataset is shown in Figure 5.29. The GUI structure is similar to the one adopted for the original Pix2Code dataset. It contains the names of the components in a structured hierarchy, separated by curly brackets. An example of GUI code is shown in Figure 5.30, while Figure 5.31 shows some samples screenshots. In the original version of the dataset, the screenshots are rotated by 90 degrees counter-clockwise. They have a width and height of 300 and 200 respectively.

5.5 Pre-processing and post-processing

This section outlines the comprehensive data processing pipeline, from initial preprocessing to final post-processing. The initial segment of the following sections describes all steps involved in transforming the original data from the dataset into a format suitable for model input. Subsequently, the post-processing steps are detailed, involving the reversion of the model's output back to the original text format, to extract the screenshots, and calculate all relevant metrics.

In particular, for HTML files, some metrics, and also the tool that extracts screenshots, require the code to be syntactically correct. To achieve this, and to

correct any minor errors that might prevent screenshot extraction, a tool called HTML-Tidy[99] is utilized. Tidy is a console application that corrects HTML and XML markup errors. Modern browsers are capable of achieving the same result and can still render a website even if it contains some errors. The command that was used returns all the types of errors encountered, and indicates their positions. These are saved in a JSON file, unique for each sample processed so that a record of all the sample errors is maintained.

5.5.1 Pix2Code Dataset processing

The pre-processing of the original Pix2Code dataset was very limited. The newline characters inside the DSL codes were replaced by spaces, and multiple spaces were truncated to have always one space separating the elements or the curly brackets.

For the other two versions of this dataset, namely the HTML version and the HTML Lorem Ipsum one, one more pre-processing step was performed in addition to the two mentioned before for the original dataset. This consists of the removal of three tags from all the HTML codes: <header>, <footer> and <script>. These tags are always in the same position and have the same content for all the samples, independent of the visual interface. This removal was done to make the texts shorter, limiting the number of tokens. They were added back in the same position during post-processing, before the extraction of the screenshots and the metrics calculation.

5.5.2 Synthetic Bootstrap Dataset processing

Four pre-processing operations were used both for the original version of this dataset and the sketch one. The first two were the same ones used for Pix2Code datasets: replacing newlines with spaces and substituting multiple spaces with a single space. The third one was the removal of HTML comments. These codes presented some of them because they originated in the generation script, like a comment that specifies which type of layout was generated. The last operation reduces the precision of decimal numbers present in the code. In particular, numbers with many decimal digits are used in the metadata element to specify the RGB values of the chosen color palette. To facilitate the model prediction of the colors, and given that such high precision was not necessary, the numbers were truncated to 0 decimal digits.

No particular post-processing was needed for the HTML files of these datasets beyond the standard steps that are common to all HTML files, as explained at the outset of this section. Prior to capturing screenshots of the predictions, it was crucial to place the folders containing all requisite CSS and JavaScript files from the original dataset along the correct path. Furthermore, replicating the screenshot extraction procedure used during the dataset generation was necessary to allow

for modifications to the CSS file using the palette derived from the HTML file metadata, ensuring that the screenshots remained faithful to the original.

5.5.3 WebUI2Code Dataset processing

Three main pre-processing steps were followed for all utilized variants of the WebUI2Code Dataset: one for the HTML file, one for the CSS file, and one for the combination of the two. The sample screenshot, did not require any processing.

For the HTML file processing, the same operations previously mentioned for the Synthetic Bootstrap dataset were also applied in this case. Specifically, these operations included the replacement of newlines with spaces, the substitution of multiple spaces with single ones, and the removal of HTML comments. The latter was still used, even though these codes should not have any comments because they were removed during the extraction of the website codes (see: Section 3.2). Some new operations were performed. All references to external websites, beginning with "http" or "https", and conforming to the URL format, were substituted with a default one: "https://example.com". Similarly, all references to local files, excluding the one linking to the CSS file, were replaced with "ref.placeholder". These modifications were implemented because the network lacks knowledge about this information and can only select random words. Furthermore, this information does not affect the appearance of the website's screenshot, and the usage of this common name can facilitate the prediction task.

The largest pre-processing procedure involved reducing the number of HTML tags. As previously mentioned in the dataset section, the original dataset includes more than 5000 tags. This is possible because HTML allows the definition of custom tags. They replace, by default, the generic inline tags , and are also commonly used to substitute tags by providing a CSS rule to change their display type to block. The idea is to use them as components, enabling easier style and behavior encapsulation without relying on generic tags and applying numerous different classes. Their use is not recommended for replacing HTML semantic tags because doing so can destroy the meaning of the markup code and have a negative impact on website indexing, as their behavior cannot be understood. With this premise, the approach was the following:

- Create a comprehensive list of all existing HTML tags.
- Identify all tags (including custom ones) referenced in the CSS files.
- Exclude from the original list all tags that are not present in either of the two lists.
- This resulting list is now considered the list of "valid" tags. When processing

the HTML file, all tags that are not in the valid list are replaced with a generic one, either or , depending on whether it's a block or inline tag.

The list of all HTML tags contains 131 tags, while the list of tags referenced at least 100 times in the CSS files comprises 334 tags. Those referenced fewer than 100 times were truncated because their number of references is very low, considering that the dataset has more than 30000 files. The final list of valid tags contains 129 tags, encompassing most of the tags from the first list and some from the second one. The <s> tag was removed and replaced with an equivalent tag because it is the special character used as the starting token in the tokenizer of our model. It was easier to replace the <s> HTML tag than to change the special tags of the model.

The last pre-processing done was removing any special character or non-ASCII character from the textual parts of the HTML codes. Again to limit unnecessary variability and especially to avoid expanding the dictionary of the tokenizer too much, with all the unrecognized words and subwords that these special characters bring.

CSS files underwent several of the previous pre-processing steps, which included the removal of comments (specifically, CSS comments), rounding of floating-point numbers, truncation of spaces, replacement of newlines, and finally, substitution of URLs with a placeholder.

The final step involved combining the two resulting text blocks by inserting the following string between them: " /* START CSS */ ". This special block is utilized in post-processing to divide the text back into two separate files. Since this follows the syntax of CSS comments, and all of those were removed during pre-processing, each file is guaranteed to have one of these special phrases.

Post-processing starts with the separation of HTML and CSS files. If the special string is not found, the sample is marked with an error in the recap JSON file, to keep track of it. Inside the HTML file, a reference to the name of the output CSS file is added. This is because the network cannot know the name of the file to reference by looking only at the website screenshot and choosing a random name.

5.5.4 Rico Dataset processing

Rico dataset codes are processed to extract only the class names of the components and organize them in a structured way, using curly brackets for separation. The "class" attribute of an element is extracted, and its full name is reduced by taking only the last two parts of the path, which indicate the element package and name respectively. Then, each of its children, if any, is processed in a similar way. The children are put between curly brackets, to maintain a clear hierarchy. No post-processing function is performed, and the predictions are compared with the

original texts with the application of the class extraction methodology. Both have the same format.

5.5.5 UI2Code Dataset processing

The UI2Code dataset necessitates minimal pre-processing, as the provided GUI texts are already appropriately formatted, with elements separated by single spaces and confined to a single line. Additionally, screenshots are rotated 90 degrees clockwise to restore their canonical vertical orientation.

No post-processing is required, given that the predictions retain the original text format.

5.6 Experiments

All the experiments were conducted using Google Colab[90]. The majority of the training sessions were performed on A100 GPUs equipped with 40GB of RAM. Most of the testing sessions were executed on V100 GPUs, optimizing both costs and time consumption without compromising the required performance. However, the initial experiment utilized a T4 GPU due to its lesser resource requirements.

Some training parameters were common for all the experiments, while others were chosen based on the dataset characteristics.

For example, the maximum number of patches for the images was set to 1024. With the ability of the model to resize the images to always extract as many patches as possible while still maintaining the correct aspect ratio, we decided to keep this parameter constant.

The number of epochs for training ranged from 10 to 15 epochs, based on the dataset size, and training speed to respect time and cost limitations. In the case of the biggest experiment performed on the WebUI2Code dataset in the 16384 version, the experiment was suspended after 5 epochs due to hardware limitations. Batch size was always set to 4 during training. Bigger values caused the experiment to crash, because of memory limits exceeding. Accumulation of gradient batches was used to simulate using a bigger batch size of 8, as explained in Section 5.2.3.

The training-testing dataset split was always around 90:10. A small portion of training is used for validation, monitoring metrics during training, and finding optimal hyper-parameters. This value changed depending on the experiment. In the most costly experiments, the validation set was restricted to a few samples, because the inference of samples required a lot of time and resources. Validation was almost always run every five epochs, to save resources.

The optimizer used was Adafactor [100], known for its effectiveness in training large-scale Transformer models with a reduced memory footprint compared to other popular optimizers such as Adam. Instead of maintaining a moving average

of parameter gradients and squared gradients for each parameter, Adafactor approximates the second-moment matrix (which represents the variance of gradients) using only a few parameters, thus conserving memory. Weight decay is applied to penalize the optimizer for having large weights, helping the model generalize better and preventing overfitting to the training data.

The optimizer is linked to a scheduler that dynamically adjusts the learning rate based on the current training step. The strategy employed is a cosine similarity schedule with a warm-up. The learning rate starts from zero and increases linearly to the starting value in the first period called warm-up; after warm-up, it decreases following the values of the cosine function to 0. This type of schedule has been shown to typically yield benefits in training and fine-tuning deep networks such as Transformers.

As another form of regularization, gradient clipping was used. Gradient clipping is a technique used to prevent gradients from becoming too large during neural network training, which can cause numerical instability and hinder model convergence. This problem is called "exploding gradients". By capping the gradients during back-propagation, gradient clipping ensures that the updates to the model parameters remain controlled and within a specified range, thereby stabilizing the training process and facilitating convergence to a good model.

The chosen loss function is Cross Entropy Loss, which quantifies the dissimilarity between the predicted probability distribution and the real distribution of subsequent tokens in a sequence during language model training. This loss function penalizes highly confident incorrect predictions, with the goal of enhancing the model's capacity to generate or predict the next token in a sequence by minimizing the gap between the predicted and actual distributions.

The moving average loss was employed to monitor loss behavior during training. Loss values may exhibit fluctuations due to various factors, including mini-batch noise, gradient noise, or other sources of variability. These fluctuations can complicate the assessment of the true trend in loss reduction. The calculation of a moving average helps to smooth out these fluctuations, making it easier to discern the overall trend.

Our implementation of the moving average loss utilizes the exponential moving average (EMA) method, which calculates a weighted average between the current loss and the previous moving average. The weight assigned to the current loss is determined by the parameter alpha, while the weight assigned to the previous moving average is 1 - alpha. In this implementation, the smoothing factor (alpha) was set to 0.1.

5.6.1 Experiments on Pix2Code Dataset

The initial experiment with the Pix2Code dataset was carried out using the original dataset, which comprised DSL codes and associated screenshots. The maximum sentence length was set at 250 tokens. This decision was made after preprocessing the DSL codes and observing that the lengthiest code contained 200 tokens.

The results highlighted the model's proficiency in handling this basic dataset, as evidenced by an average BLEU score of 98.9% on the validation set.

Table 5.4 offers an in-depth statistical breakdown of metrics derived from the test set used in the experiment. This includes the total number of samples, mean value, standard deviation (std), minimum observed value (min), the 25%, 50%, and 75% percentiles, and maximum recorded value (max).

	BLEU	**ED**	**N.ED**	**SSIM**
count	174	174	174	174
mean	0.983	4.437	0.016	0.942
std	0.028	6.522	0.028	0.026
min	0.873	0.000	0.000	0.875
25%	0.964	0.000	0.000	0.922
50%	1.000	0.000	0.000	0.949
75%	1.000	14.000	0.035	0.956
max	1.000	14.000	0.125	0.992

Table 5.4: Statistics on the metrics from the Pix2Code Dataset's test set.

BLEU Score is high across the whole dataset, with a minimum value of 87.3%. Edit distance is very low, with only an average of 4.4 characters per website that are incorrect.

Each row in Figure 5.32 juxtaposes the correct answer screenshot (left) with its prediction (right) for 4 test set samples. The colors and elements are largely accurate, with the exception of some additional buttons present in the predictions. The texts do not match because they are randomly generated during the compilation of the predicted DSL and HTML code, and were not a prediction target in this experiment.

Table 5.5 shows the results on the Pix2Code dataset of an improved variant of the original Pix2Code model, proposed by Angerer et al. [101]. It is a Pytorch implementation of a model based on the original one proposed by Beltramelli [4], with the substitution of the CNN model used for visual encoding. The original VGGNet model is replaced with another CNN, namely ResNet-152, pre-trained on the ImageNet dataset. The features of the input image, extracted by the CNN, and the sequence of GUI tokens pass then through an LSTM model, similarly to the original model architecture. The complete model architecture schema can be

seen in Figure 5.33.

The authors' proposed experiment settings were preserved to train the model, generate testing predictions, and calculate the desired metrics.

	BLEU	**ED**	**N.ED**	**SSIM**
count	348	348	348	348
mean	0.878	44.925	0.132	0.935
std	0.091	33.041	0.086	0.025
min	0.421	0.000	0.000	0.873
25%	0.870	25.000	0.075	0.913
50%	0.904	36.000	0.105	0.937
75%	0.925	54.000	0.159	0.951
max	1.000	236.000	0.567	0.992

Table 5.5: Statistics on the metrics for the LSTM-based model from the Pix2Code Dataset's test set.

The LSTM-based model attains an average BLEU score of 87.8%, though it encounters difficulties with some samples, dropping to a minimum value of 42.1%. In contrast, Pix2Struct outperforms this, achieving an average score of 98.3%, with its minimum value remaining above 87%. Figure 5.34 juxtaposes the BLEU score distributions of the two models. It is noteworthy that the full-scale minimum values in the two graphs are 0.88 and 0.4, respectively.

In evaluating normalized edit distance, the Pix2Struct model demonstrates a notable improvement over the Pix2Code Pytorch. The discrepancy, measured as the average percentage of incorrect characters between predicted and answer texts, reduces dramatically from 13.2% to 1.6%. Additionally, the similarity index shows a marginal enhancement in the Pix2Struct results, posting an average value of 0.94 against the 0.93 of Pix2Code Pytorch.

The second experiment, initiated from a checkpoint obtained after the first, utilized the HTML version of the dataset. Due to the presence of longer HTML code texts in this dataset, the maximum sentence length was increased to 1024. However, the model was unable to effectively predict the website codes, achieving a modest validation BLEU score of approximately 21%.

Figure 5.35 shows the comparison between the answer and the prediction for one sample in the validation set. The behavior highlighted in the comparison is common to all the samples in the validation set. While the model accurately predicts the first part of the code, it struggles to correctly decipher the text of buttons and paragraphs in the middle section. This issue becomes even more pronounced in the final part of the text, where the model repeatedly produces the same characters and fails to complete other structural components, leaving a partial and broken

code.

The cause of this phenomenon may be catastrophic forgetting (CF), where a machine learning model loses previously acquired knowledge while learning new data. Luo et al. [102] showed how CF has a significant impact on large language models, and in particular on encoder-decoder ones, such as the one used in these experiments. This experiment started from a checkpoint from the fine-tuning of the original version of the Pix2Code dataset, where button texts and paragraphs were not meaningful because the DSL codes only contained structural elements. So, the model can have lost the ability to recognize those texts, focusing only on elements forms, colors, and positions.

A new experiment was conducted on the same Pix2Code HTML dataset, starting from the original base-model weights. The model achieved a validation BLEU score of 84%, accurately predicting most of the code. However, it did miss some button colors and texts, specifically those with white text on colored backgrounds. Figure 5.36 shows this phenomenon.

The statistics of the metrics calculated on the test set for this experiment are shown in table 5.6. From left to right the metrics present are BLEU score, edit distance, normalized edit distance, structural BLEU score, HTML tree edit distance, normalized HTML tree edit distance, and structural similarity index. These now include Structural Bleu and Tree Edit distance, which can be calculated from HTML codes.

	BLEU	**ED**	**N.ED**	**S.BLEU**	**TED**	**N.TED**	**SSIM**
count	174.000	174.000	174.000	174.000	174.000	174.000	174.000
mean	0.847	176.345	0.124	0.955	1.770	0.033	0.970
std	0.021	44.626	0.018	0.022	1.456	0.027	0.012
min	0.781	36.000	0.081	0.879	0.000	0.000	0.912
25%	0.832	155.000	0.112	0.936	0.000	0.000	0.967
50%	0.847	184.500	0.125	0.955	2.000	0.036	0.971
75%	0.860	203.750	0.134	0.970	2.000	0.049	0.974
max	0.904	292.000	0.188	1.000	4.000	0.108	0.994

Table 5.6: Statistics on the metrics from the Pix2Code HTML Dataset's test set.

Another variant of the dataset was then analyzed, with the random-characters words being replaced with Lorem Ipsum words. Maximum sentence length is kept to 1024 tokens. Again, a first experiment was executed starting from the last checkpoint of the fine-tuning on the original Pix2Code dataset. This time the model didn't have the code-breaking prediction tendency, while still not being able to predict correctly most of the titles and button texts. However, it still reached 77% validation BLEU score, thanks to its capability to predict correctly almost all

the structural elements of the samples.

The last experiment was done on this version of the dataset starting from the original base-model weights. The model was able to reach the same level of correctness shown on the original Pix2Code dataset, with a validation BLEU score of 98%.

Table 5.7 shows the statistical analysis of the metrics calculated on the test set for this last experiment. The results were consistent with validation ones, obtaining an average BLEU Score of 97.4%.

	BLEU	**ED**	**N.ED**	**S.BLEU**	**TED**	**N.TED**	**SSIM**
count	175.000	175.000	175.000	175.000	175.000	175.000	175.000
mean	0.974	29.646	0.023	0.998	0.114	0.002	0.994
std	0.020	16.157	0.019	0.009	0.466	0.010	0.003
min	0.873	5.000	0.003	0.941	0.000	0.000	0.985
25%	0.970	16.500	0.013	1.000	0.000	0.000	0.992
50%	0.978	29.000	0.020	1.000	0.000	0.000	0.994
75%	0.986	39.000	0.027	1.000	0.000	0.000	0.997
max	0.996	91.000	0.116	1.000	2.000	0.057	1.000

Table 5.7: Statistics on the metrics from the Pix2Code HTML Lorem Ipsum Dataset's test set.

Figure 5.37 displays the comparison of answers and predictions screenshots for 4 samples of this dataset. The model is now correctly predicting the buttons' colors and texts, overcoming the previous limitations.

Overall, these experiments showed Pix2Struct model capability to predict structured code for web UIs, both for a simplified DSL language and more sophisticated HTML code. However, they also highlighted some critical issues of the model when it finds complex or unknown words. In particular, it is interesting the tendency of the model to not be able to finish the prediction of the structured code when it enters in a loop where it doesn't know how to exit from.

Figure 5.38 displays the distribution of maximum lengths across the test set samples used during the Pix2Code final experiments, highlighting a 5x increase from the samples of the original dataset to those of the HTML and HTML Lorem Ipsum versions. Maximum length here refers to the maximum between prediction and answer number of characters, which is used as the denominator for the calculation of normalized edit distance metric.

Figure 5.39 displays the distribution of normalized distance for test samples in the previously mentioned experiments. It highlights a discernible improvement in model performance on the HTML LI version of the dataset compared to the HTML experiment. However, it still does not attain the minimum number of incorrect

characters achieved on the original dataset, which features significantly shorter texts, as previously demonstrated.

Similar conclusion can be derived by looking at the distribution of BLEU score in Figure 5.40.

Experiments on both HTML and HTML LI versions slightly enhance the average Structural Similarity Index (SSIM), compared to experiments on the original version of Pix2Code. This metric, being the first non-textual one and not affected by the simplicity of the original dataset codes, is higher for the two more complex datasets.

5.6.2 Experiments on Synthetic Bootstrap Dataset

The first experiment was performed using the "mini" version of the dataset with 1000 samples. Maximum sentence length was set to 4096 tokens, using the sliding window system with chunks of 1024 tokens, and a context overlap of 256 tokens. After 20 epochs, the model achieved an average validation BLEU Score of 81%, peaking at 93% in certain samples but also underscoring a poorer performance in a substantial number of instances with a score dipping to 41%. When evaluated on the test set, which offers a larger and more representative data subset, the model's performance slightly improves. The overall average BLEU score rounds to 88%, showcasing impressive scores of 93.6% and 96.5% in the 50th and 75th percentiles, respectively. However, a notable decline to 85.7% is observed in the 25th percentile, highlighting disparities in model performance across different samples.

	BLEU	**ED**	**N.ED**	**S.BLEU**	**TED**	**N.TED**	**SSIM**
count	100	100	100	100	100	100	96
mean	0.878	774.120	0.134	0.903	11.010	0.126	0.741
std	0.132	963.414	0.133	0.086	12.848	0.147	0.091
min	0.480	41.000	0.014	0.579	0.000	0.000	0.399
25%	0.857	142.250	0.043	0.878	2.000	0.032	0.688
50%	0.936	332.500	0.081	0.939	5.000	0.069	0.753
75%	0.965	933.000	0.173	0.960	18.000	0.195	0.792
max	0.986	4033.000	0.530	0.986	68.000	0.971	1.000

Table 5.8: Statistics on the metrics from the Synthetic Bootstrap Mini Dataset's test set.

Similar numbers are obtained for the Structural BLEU score, with a couple of percentage points higher for the average value (90.2%) and the 25th percentile one (87.8%). The distributions of the BLEU Score and Structural BLEU score are very similar, presenting a considerable percentage of samples with lower performance.

This indicates that their fallback is related not only to textual elements but also to the structural ones, like HTML tags and attributes.

The mean of the Structural similarity index (SSIM) is 0.74, and its distribution approximates a normal distribution, extending from a lower extreme of 0.4 to an upper limit of 1.0, as shown in Figure 5.41. In this case, the SSIM was calculated only on 96 out of the 100 test set samples, since 4 of them presented some errors during the extraction of the screenshots.

Figure 5.42 displays the screenshots for answers and predictions of 3 samples of the test set. Most of the elements and the texts are correct, but the colors are completely wrong or missing, and some structural components are out of position.

These difficulties have been overcome with the second experiment on the dataset of 10000 samples. Note that the full dataset of 50000 samples was not used for computational limitations, as mentioned in the dataset section. The setup in this experiment was the same as the previous one, and the validation BLEU score was 92%, with a minimum of 76% and a maximum of 98%.

Table 5.9 shows the statistics of the metrics calculated on the test set for the experiment on the Synthetic Bootstrap Dataset. The mean test BLEU is very high and reflects the validation results.

	BLEU	**ED**	**N.ED**	**S.BLEU**	**TED**	**N.TED**	**SSIM**
count	1000	1000	1000	1000	1000	1000	1000
mean	0.929	443.890	0.081	0.951	3.946	0.049	0.783
std	0.079	668.592	0.090	0.052	6.486	0.075	0.110
min	0.359	23.000	0.005	0.524	0.000	0.000	0.293
25%	0.918	94.000	0.023	0.951	0.000	0.000	0.720
50%	0.962	170.500	0.042	0.969	1.000	0.018	0.790
75%	0.976	478.750	0.105	0.977	5.000	0.067	0.848
max	0.993	6099.000	0.605	0.994	55.000	0.592	1.000

Table 5.9: Statistics on the metrics from the Synthetic Bootstrap's test set.

Figure 5.43 shows the distribution of BLEU score across the samples of the test set, highlighting a low number of outliers with low score values.

In comparison to the prior experiment utilizing the mini dataset, the Structural Similarity Index (SSIM) measured in this experiment demonstrates a generally elevated mean value of 0.78. Although the distribution retains a shape analogous to the previous experiment, it is now shifted to the right and features a peak at 1.0, as shown in Figure 5.45. This suggests a surge in the proportion of samples achieving a flawless screenshot.

Figure 5.44 illustrates the screenshots of answers and predictions for three test set samples. Although the colors are not uniformly accurate across all samples, the

disposition of lighter and darker parts appears to be preserved, with all structural elements correctly positioned.

The average value of the HTML tree edit distance has reduced to nearly a third, now standing at 3.94 compared to the previous 11.01. Additionally, the edit distance, reflecting the number of differing characters between answer and prediction, has decreased from 774.12 to 443.89.

The correlation matrix between the metrics is depicted in Figure 5.46. Prediction length, answer length, and the maximum between the two are also present in the matrix. There are negative correlations between BLEU Score and textual lengths, with values ranging from 0.19 to 0.38. This suggests that the BLEU Score tends to decrease as the length of predictions and answers increases. BLEU score has a notably strong negative correlation with edit distance and its normalized counterpart, around -0.88 for both. This is as expected because as the edit distance increases, indicating greater dissimilarity, BLEU Scores tend to decrease. In this experiment, the newly introduced structural BLEU is confirmed to capture aspects of text similarity comparable to traditional BLEU. This aligns with our expectations, given that textual elements are present in the images, enabling the model to effectively predict them. Similarly, the HTML tree edit distance shows a strong correlation with edit distance (0.67). The similarity index generally shows a very low correlation with other metrics, indicating that is the only measure capable of monitoring some visual aspects of the data, and none of the textual ones can identify them.

The final experiment was conducted on the Sketch Dataset variant. The samples with more than 4096 tokens in the HTML code were filtered out, having a resulting total of 9775 samples. This experiment started from the last checkpoint of the previous one and lasted only 10 epochs. The model achieved a notable average validation BLEU score of 83%, a significant accomplishment given that texts were replaced by sketches and not visible in the interface. The BLEU score is inherently limited since the model selects all Lorem Ipsum words randomly. Thus, other metrics may more accurately represent the model's success on this specific dataset during testing on the test set. Table 5.10 shows the statistics for the metrics calculated on the test set of this dataset.

As expected, the Structural BLEU score is considerably higher than the BLEU score, a difference attributable to the texts not being visible in the sketches. It increases from an average value of 82.5% to 92.2%, approaching the result on the other dataset, which recorded a 92.9% BLEU score and a 95.1% Structural BLEU score.

The edit distance in this dataset has risen substantially from the previous one, escalating from an average of 443.9 to 1146.7 for consistent reasons. This signifies a jump from 8.1% to over 20% concerning the percentage of incorrect characters throughout the entire text. The HTML tree edit distance has also experienced

	BLEU	ED	N.ED	S.BLEU	TED	N.TED	SSIM
count	979	979	979	979	979	979	979
mean	0.825	1146.678	0.202	0.922	6.619	0.072	0.810
std	0.115	1066.018	0.109	0.071	11.721	0.094	0.097
min	0.333	126.000	0.041	0.516	0.000	0.000	0.342
25%	0.767	478.000	0.117	0.907	0.000	0.000	0.750
50%	0.870	763.000	0.170	0.943	2.000	0.037	0.820
75%	0.911	1428.000	0.268	0.965	8.000	0.105	0.877
max	0.959	6096.000	0.612	1.000	97.000	0.519	0.986

Table 5.10: Statistics on the metrics from the Sketch Synthetic Bootstrap Dataset's test set.

a rise, averaging 6.62 per website compared to 3.95 in the prior dataset. When normalized, an average of 7.16% of HTML tags are either improperly used or mispositioned, marking an increase from the 4.88% found in the preceding dataset. However, this increase is considerably lower when compared to the one in edit distance. The reason is the same as previously mentioned for BLEU and Structured BLEU. The presence of random texts, due to the non-visibility of texts in the sketches, has a significant impact on metrics that are not general and 'structural', but instead focus on entire texts.

The Structural Similarity Index (SSIM) is computed by comparing the rendered website screenshots between the predicted sample code and the answer. This screenshot was never seen by the model, which only saw its corresponding sketch version, but reflects the original and predicted HTML code.

The SSIM is higher in this experiment with a mean value of 81%, compared to the 78.3% calculated on the previous dataset. Its distribution is shown in Figure 5.47.

Figure 5.48 displays screenshots comparing answers and predictions for four test set samples. The first column contains the sketch versions provided to the model, the second column presents the original website screenshots, and the third column showcases the model's predicted website screenshots. The model is capable of recognizing all the website components from the sketch and providing the correct HTML code.

5.6.3 Experiments on WebUI2Code Dataset

Due to limitations related to hardware, costs, and time, most experiments were conducted using the WebUI2Code-4096 version, which aligns with the maximum number of textual tokens per file seen in previous experiments on synthetic datasets.

In the first experiment, the starting point was the checkpoint obtained after

training on the synthetic Bootstrap dataset. The sliding window mechanism was employed to process sentences of 4096 tokens, using chunks of 1024 and contexts of 256 tokens.

After fine-tuning the model for 10 epochs, it exhibited a combination of relatively positive and negative results. Specifically, some predictions for the validation set displayed an issue with the repetition of tokens. After a certain point in the prediction, the model "loses focus" and repeats the same pattern over and over again. This continues until the maximum generation length is reached, and because of the inability to complete the correct structure, this leads to syntactically incorrect code. Moreover, the repetition of the same pattern numerous times significantly impacts the resulting BLEU score, reaching values as low as 9% on the validation set. Consequently, the average BLEU score is quite low at 35%. On the other hand, for some samples, the model was able to predict functional HTML/CSS code. Figure 5.49 shows a comparison between answer and prediction screenshots. The principal textual components were correctly predicted; however, the model was not capable of positioning all the elements correctly. For example, the navigation bar at the top is missing, and the disposition of the three bottom elements is incorrect. The prediction also lacks the correct colors, and the buttons have a different style.

The greedy search strategy, utilized during the generation of text by the model, is characterized by the selection of the next token with the highest probability. This strategy can lead to the generation of repetitive words or sentences, even in thoroughly trained models. To mitigate this issue, Keskar et al. proposed an alternative sampling technique. While emulating the functionality of the greedy search, it also curtails repetitions by integrating a penalty mechanism [103].

The choice of the penalty value is crucial because, if it is too high, the model might generate unnatural or incoherent text. This could occur as it might overly avoid using the same words or phrases, thereby potentially sacrificing contextual accuracy.

Various repetition penalty values, ranging from 1.1 to 1.5, were tested. A higher value increases the model's aversion to repetitions but also tends to encourage shorter predictions. A penalty value of 1.4 emerged as optimal, yielding an average validation BLEU score of 51.6% and a minimum of 41%. In the validation set, the model successfully eliminated repetition loops, accurately completing the entire code structure predictions.

A second experiment was conducted, altering the context size in the sliding window mechanism from 256 to 512. The number of new tokens predicted in each iteration was reduced from 768 to 512, with a maximum of 7 windows needed to cover the 4096 sentences. This change did not bring any improvement to the final result and obtained its best validation BLEU score of 51.1%, with the repetition penalty set to 1.3.

A new experiment was conducted using a larger portion of the WebUI2Code

dataset. The selected threshold was 16384, corresponding to a maximum of 21 sliding windows, with a context of 256 and a chunk size of 1024. The resources required to train the model were significantly higher compared to previous experiments, even though it was trained for only 5 epochs, pushing our capabilities to their limits. The results were notably lower than those of previous trials, with the best average BLEU score on the validation set being 17.3%, achieved using a repetition penalty of 1.4. This version of the dataset introduces a new level of complexity, suggesting that additional experiments are necessary to find the optimal settings for the model. Given our computational limitations, a decision was made to focus on the smaller version of the dataset for subsequent experiments.

In the fourth experiment, the model was fine-tuned, starting from the original Pix2Struct-base weights, and not from those obtained in the previous training on the synthetic Bootstrap dataset. This new configuration achieved the best result to date, securing an average BLEU score of 56.1% on the validation set while utilizing a repetition penalty of 1.3.

An additional experiment attempted to mimic the behavior exhibited during the experiments on the HTML version of Pix2Code. The texts inside the HTML without syntactic function were replaced by Lorem Ipsum words, and new screenshots were obtained. This strategy reduced the maximum number of words inside the dataset. However, this time, the results did not improve, achieving an average BLEU score of 36.7% on the validation set, with a repetition penalty of 1.3.

Table 5.11 illustrates the average BLEU score, determined from the validation set across various experiments, by varying the repetition penalty from 1.0 (unused) to a peak of 1.5. Each row corresponds to an experiment, with "512" corresponding to the experiment done with a bigger context, "FULL" to the experiment fine-tuned from scratch, and "LI" to the Lorem Ipsum experiment. Across all experiments, the optimal repetition penalty value is identified between 1.3 and 1.4, as lower values still present the problem of repeated words, while higher values lose context.

	1.0	1.2	1.3	1.4	1.5
W2C 4096	0.350	0.426	0.486	0.516	0.378
W2C 4096 - 512	0.290	0.391	0.511	0.508	0.446
W2C 4096 - FULL	0.092	0.403	0.561	0.470	0.372
W2C 4096 - LI	0.088	0.273	0.367	0.357	0.232
W2C 16384	0.061	0.062	0.153	0.173	-

Table 5.11: Average validation BLEU scores across experiments on WebUI2Code datasets with varying repetition penalties.

The best model, which corresponds to experiment number four, and reached a validation BLEU score of 56.1%, was trained on the whole test set. Table 5.12

contains a statistical analysis of all the metrics calculated on the test set. The average BLEU score declines to 43.6%, indicating a potential discrepancy between the distributions of samples in the validation and testing sets. The difference is likely caused by the limited sample size in the validation set, which was chosen for its time and cost efficiency. The distribution of BLEU is displayed in Figure 5.11, indicating that some samples received a very low score, with seven of them registering a 0. Upon examination of the codes of the results, it is evident that a repetition penalty is still present in a portion of the results. Structural BLEU result is even lower with 43.3%, indicating that the difference between predictions and answers is not only related to the textual portions of the code but also to its structural components. Also in this case samples with repetition penalty massively influence the result, with multiple of them reaching 0. This becomes clear when looking at the HTML tree edit distance result, which averages 80 different HTML tags per website, between answer and prediction. The Structural Similarity Index could be calculated for only 221 out of the 233 samples because some samples presented errors during screenshot extraction. Overall, it exhibited an average value of 0.547, with a maximum of 0.990 and a minimum of 0.001.

When filtering out the samples that are clearly incorrect—due to their lack of a token that separates HTML and CSS code, resulting in empty CSS code—the total number of samples is reduced by 20%. Upon examining the remaining samples, the average BLEU score increases to 49.9%, and the structural BLEU score rises to 46.4%.

	BLEU	**ED**	**N.ED**	**S.BLEU**	**TED**	**N.TED**	**SSIM**
count	233	233	233	233	233	233	221
mean	0.436	6920.180	0.714	0.433	80.558	0.691	0.547
std	0.210	3087.700	0.108	0.227	57.073	0.286	0.249
min	0.000	252.000	0.166	0.000	0.000	0.000	0.001
25%	0.266	5155.000	0.658	0.254	52.000	0.602	0.367
50%	0.477	6008.000	0.726	0.457	76.000	0.735	0.620
75%	0.605	8129.000	0.779	0.621	102.000	0.906	0.746
max	0.835	21535.000	0.960	0.954	335.000	1.031	0.990

Table 5.12: Statistics on the metrics from the WebUI2Code-4096 Dataset's test set.

Figure 5.51 presents the screenshots for both answers and predictions of two test set samples. Overall, the experiments on WebUI2Code demonstrated how the Pix2Struct model struggles to predict real-world complex websites, showcasing lower performance compared to synthetic Datasets. New problems emerge, like repetition of words. The use of techniques, like repetition penalty, was shown to

partially mitigate the issue, suggesting that more experimentation and the usage of more data might lead to better results.

5.6.4 Experiments on Rico Dataset

The training on the Rico dataset lasted 15 epochs, employing a maximum sentence length of 4096, which was enough to cover the largest text, containing 2698 tokens. Final outcomes revealed mixed results, with both high and low BLEU scores on the validation set, mirroring observations from the WebUI2Code Dataset. Also in this case, some elements were characterized by repetitions of words and small phrases, resulting in very low value scores. On the other hand, the best predictions achieved relatively high BLEU scores, peaking at 82%. Attempting different repetition penalty values did not suffice to eliminate all repetition issues in the validation set. The best value found was 1.4, which attained an average BLEU score of 34% on the validation set, with the top prediction reaching 93%.

This dataset proved to be particularly tricky, with many samples yielding results close to 0, while others managed to obtain high values. This could potentially be related to the nature of the dataset, especially its code structure. For example, compared to the previous HTML codes, here there are only structural elements composing the text, and not visible textual elements. Moreover, although the tags were extracted directly from the application's layout, this doesn't necessarily mean that the information correlates directly with what is visible in the screenshot.

Given this premise, the research expanded to explore other hyper-parameters, including temperature, top-k, and top-p for sampling mode.

In contrast to the previously employed greedy search strategy for text prediction, we now employ sampling. Greedy search entails selecting the word with the highest probability as the next word in the sequence at each step. This approach is suitable when the desired output should be deterministic and coherent, aligning well with our current context.

However, in an effort to address the issue of word repetition, we opted to test the sampling mode. By choosing always the optimal word, a greedy search can lead to a repetition problem, and the model remains blocked. Sampling selects the next word by drawing from the probability distribution of potential next words, and this intrinsic randomness of the choice can avoid repetition problems. It can be adjusted using three parameters, which can be employed either simultaneously or individually.

The first parameter, Top-k, involves sampling from the top k most probable next words. The second parameter, Top-p (also known as nucleus), selects samples from the smallest set of words whose cumulative probability surpasses a specified threshold. Lastly, Temperature adjusts the probability distribution, making it either flatter (more random) or sharper (more deterministic).

When these three parameters are combined, they provide control over both the diversity and the quality of the generated text. This allows for more varied yet controlled text generation, potentially addressing the aforementioned issue.

Experiments were conducted on the validation set by varying three parameters and observing the resultant model behavior, as detailed in Table 5.13. Initial experiments, represented by the first three rows, report the previous three results using a greedy search alongside varied repetition penalty values. Subsequent rows detail variations in Temperature, Top-k, and Top-p, utilizing sampling. Though the hyperparameter space was not exhaustively explored, the impact of these parameters is evident, improving both the average and the maximum BLEU score with 0.4 and 0.96, respectively.

Id	Sampling	R. Pen.	Temp.	Top-k	Top-p	Mean	Min	Max
0	False	1.0	None	None	None	0.14	0.01	0.82
1	False	1.2	None	None	None	0.34	0.01	0.82
2	False	1.4	None	None	None	0.34	0.0	0.93
3	True	1.4	0.9	40	None	0.36	0.0	0.94
4	True	1.2	0.9	40	None	0.37	0.0	0.95
5	True	1.2	0.5	None	None	0.38	0.02	0.96
6	True	1.3	0.9	10	None	0.4	0.0	0.95
7	True	1.3	0.9	5	None	0.36	0.0	0.86
8	True	1.2	0.7	5	None	0.35	0.0	0.95
9	True	1.4	0.5	5	None	0.38	0.0	0.86
10	True	1.3	0.8	None	0.2	0.32	0.01	0.91
11	True	1.3	0.4	10	None	0.38	0.01	0.92
12	True	1.3	0.9	5	None	0.39	0.01	0.89

Table 5.13: Analysis on the impact of different hyperparameters, including repetition penalty, temperature, top-k, and top-p, on the BLEU Score for the Rico Dataset's validation set.

Figure 5.52 shows the distribution of the BLEU scores for samples in the experiments conducted on the validation set. Compared to the first experiment, which did not employ any repetition penalty, all subsequent experiments have a much-improved distribution of BLEU scores, with higher median values, indicated by black lines. Additionally, most of the samples are positioned in a higher portion of the plot, reflecting elevated BLEU values. The most consistent experiment is the number 6, with sampling, repetition penalty of 1.3, Temperature of 0.9, and Top-k equal to 10. It presents not only the highest average BLEU score, but the distribution of the majority of the samples is higher, compared to other experiments.

This setting, together with the best setting with repetition and without sampling is chosen for the final evaluation on the test set. The results show a big drop in performance for the greedy setting, reaching an average BLEU score of 22.9%. On the contrary, the sampling setting maintained a similar behavior, attaining an average BLEU score of 41.9%.

Table 5.14 presents the statistical analysis of the results using sampling configuration. The average normalized edit distance is notably high, with erroneous character predictions exceeding 50% between answers and predictions on average. BLEU scores peak at a maximum value of 1.0, indicative of a perfect prediction, while 75% of the scores surpass the 65% mark. The heterogeneity of the results once again underscores the challenges Pix2Struct encounters when predicting codes for this specific dataset. The employment of sampling proved to be a viable methodology, offering a balanced and enhanced performance in comparison to the greedy search approach.

	BLEU	**ED**	**N.ED**
count	125	125	125
mean	0.419	910.840	0.560
std	0.272	1070.783	0.199
min	0.000	0.000	0.000
25%	0.206	332.000	0.447
50%	0.421	610.000	0.546
75%	0.663	1041.000	0.707
max	1.000	7524.000	0.948

Table 5.14: Statistics on the metrics from the Rico Dataset's test set using sampling.

5.6.5 Experiments on UI2Code Dataset

The model was trained using a subset of the UI2Code dataset, proposed by Chen et al. [9]. A total of 10,000 samples were extracted from the original training split, mirroring the methodology utilized in preceding experiments and constrained by available resources. The maximum sentence length was established at 512, sufficiently covering all samples, which exhibited a maximum token count of 444.

Upon conclusion of the experiment, the model yielded an average BLEU score of 65% on the validation set. It demonstrated a varied predictive capability, with some samples that were exactly predicted (BLEU 100%), while others obtained scores as low as 15%.

Analysis of the predictions revealed a recurrent issue observed in previous experiments on real-world datasets: the unwarranted repetition of a word at the

end of a prediction. This phenomenon has a massive impact on the average performance since the incorrect word is repeated many times, sometimes reaching the maximum possible sentence length.

Considering the relatively low cost associated with conducting experiments using this dataset, compared to the previous ones, we opted to engage in further experimentation with this dataset and its validation set.

In experimenting with hyperparameters, we investigated their impact on model performance and identified the optimal values to achieve a higher average score.

The first hyperparameter considered is the repetition penalty. This factor was already showcased in previous experiments and relates to penalizing the model when it repeats the same token. Values ranging from 1.05 to 1.4 were tested, where a higher value indicates a higher penalty and 1.0 indicates no penalty. The mean BLEU values obtained with different repetition penalties are shown in Table 5.15

Figure 5.53 displays the BLEU scores achieved by individual samples throughout various experiments. When the repetition penalty is set to a high value, an increase is observed in the number of samples yielding a BLEU score of 0. This behavior is attributed to the high repetition penalty's inclination toward smaller predictions, which can sometimes be entirely inaccurate and therefore penalized during BLEU score computation. Utilizing a smaller repetition penalty value can alleviate the original issue of word repetitions while still avoiding the tendency to short predictions. The optimal value was found to be 1.05, marking a 9-percentage-point improvement compared to the initial experiment without repetition penalty.

Repetition penalty	Mean BLEU
1.0	0.65
1.05	0.74
1.1	0.73
1.2	0.64
1.3	0.57
1.4	0.51

Table 5.15: Analysis on the impact of repetition penalty on the BLEU Score for the UI2Code Dataset's validation set

After conducting the repetition penalty experiment, the research expanded to explore other hyper-parameters, including Temperature, Top-k, and Top-p for sampling mode. These parameters were already used during experiments for Rico, and they provide control over both the diversity and the quality of the generated text, influencing the balance between prediction randomness and coherence. Based on our initial test results and considering our problem context, we have defined specific constraints for the parameters as follows: Temperature is constrained to

values between 0.60 and 1.0. Values higher than 1.0 increase randomness, while values lower than 1.0 reduce it. Therefore, we have decided to limit it within this second range. The Top k parameter is restricted to values between 0 (no effect) and 50. The Top p is constrained between 0.3 and 1 (no effect). Lower values lead to a more focused and deterministic generation. To analyze the impact of these parameters with reasonable values, we decided to truncate the lower half of Temperature (below 1) and Top-p, as well as the higher values of Top-k. This approach allows us to assess the effects of these parameters when activated, as we have already addressed the "deterministic" search using greedy search. Additionally, we have introduced a repetition penalty ranging from 1.0 to 1.3.

"Weights & Biases" (Wandb) [104] platform offers functionality that enables hyperparameter sweeps for machine learning experiments. A hyperparameter sweep is a technique used to search for the best set of parameters for a machine-learning model. This method navigates through the hyperparameter space, using a Bayesian probabilistic model to predict which configurations are likely to yield improved model performance.

After 72 runs, the model was able to identify the optimal configuration, resulting in a mean BLEU score of 76.3%, not only matching the result of the greedy "deterministic" search but also surpassing it by 2 percentage points. This achievement was realized while utilizing a Temperature of 0.7, a Top-k value of 47 (one of the highest possible values with minimal impact), a Top-p value of 1 (non-active), and a repetition penalty of 1.02 (very low).

All runs that achieved a BLEU score greater than 74% had a repetition penalty ranging between 1.01 and 1.07, a Temperature of 0.6, 0.7, or 1.0 (inactive), indicating a preference for values related to more deterministic outputs. The Top-k value varied from 2 to 49, and Top-p values were typically very high when active, so their impact seems to be either non-influential or limited.

In particular, configurations that allowed for a lot of freedom and creativity in the choice of words did not perform well. The same also occurred with settings with high repetition penalties, similar to what was observed in the greedy search experiments.

Final tests were conducted on the model using the two best configurations: one with greedy search and a repetition penalty, and the other with sampling using the best-discovered hyperparameters. These tests were conducted on a test set consisting of 1000 samples.

Tables 5.16 and 5.17 display the statistical analysis of the results obtained from the two methodologies. The sampling method appears to exhibit slightly superior performance in terms of BLEU, boasting an average value of 74.4% as opposed to 73.0% achieved by the greedy search approach. Furthermore, it demonstrates a higher level of consistency, as evidenced by a lower standard deviation. Additionally, both the edit distance and normalized edit distance are lower in the sampling

method. Overall, the performance of the two methods is quite similar, with the sampling method holding a slight advantage.

	BLEU	**ED**	**N.ED**
count	1000	1000	1000
mean	0.730	290.822	0.268
std	0.264	395.743	0.241
min	0.000	0.000	0.000
25%	0.568	48.000	0.074
50%	0.820	139.500	0.198
75%	0.942	362.250	0.432
max	1.000	2365.00	0.967

Table 5.16: Statistics on the metrics from the UI2Code Dataset's test set.

	BLEU	**ED**	**N.ED**
count	1000	1000	1000
mean	0.744	210.691	0.255
std	0.253	232.057	0.225
min	0.000	0.000	0.000
25%	0.606	53.750	0.081
50%	0.833	132.500	0.186
75%	0.931	284.000	0.398
max	1.000	1487.000	0.963

Table 5.17: Statistics on the metrics from the UI2Code Dataset's test set using sampling.

Overall, the final result is not far behind what was achieved with the model presented together with the UI2Code dataset. That model combines a Convolutional Neural Network (CNN) for visual encoding and a Recurrent Neural Network (RNN) for code prediction and achieved a BLEU score of 79.09% with a greedy search strategy. However, it's important to note that the model in our experiment was trained on a smaller dataset, consisting of only 9000 samples for training and not the original 180000 samples, due to resource limitations. Furthermore, it's worth mentioning that the original experiment used a strategy of encoding words as entire tokens, which imposes significant limits on sentence length. This factor could potentially be advantageous for the network, as it needs to predict sequences with shorter dependencies. It could be interesting to test the Pix2Struct model on the entire dataset while employing a similar strategy, as opposed to the sub-word tokenizer used in this particular setting.

To summarize, experiments on real-world mobile UIs experienced the same issues discovered during experimentation on real website datasets. A more complete analysis of hyperparameters leads to considerable improvements, suggesting that further experiments on WebUI2Code could also see results enhancements.

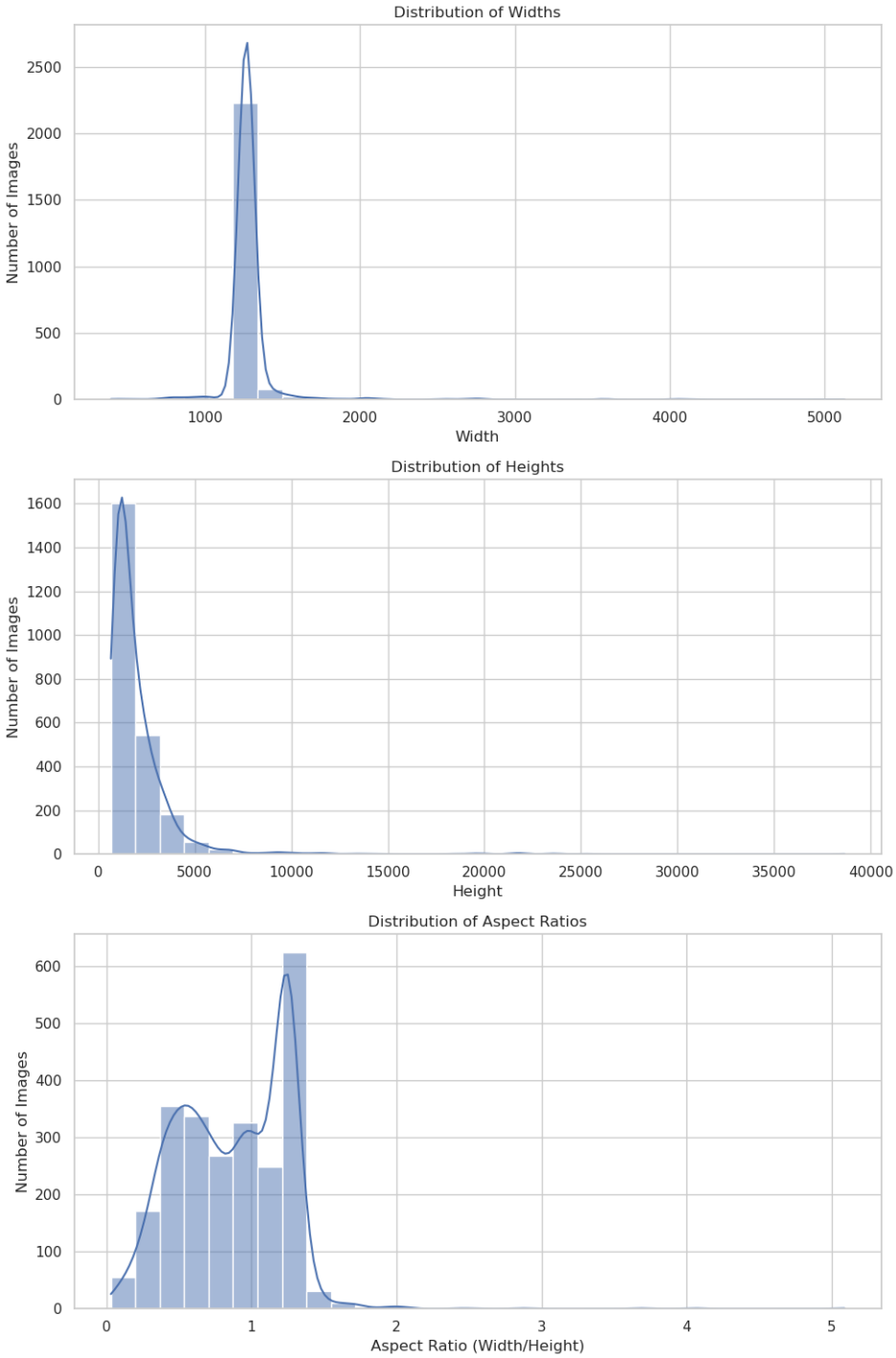

Figure 5.15: WebUI2Code-4096: distributions of image widths, heights and aspect ratios prior to data cleaning.

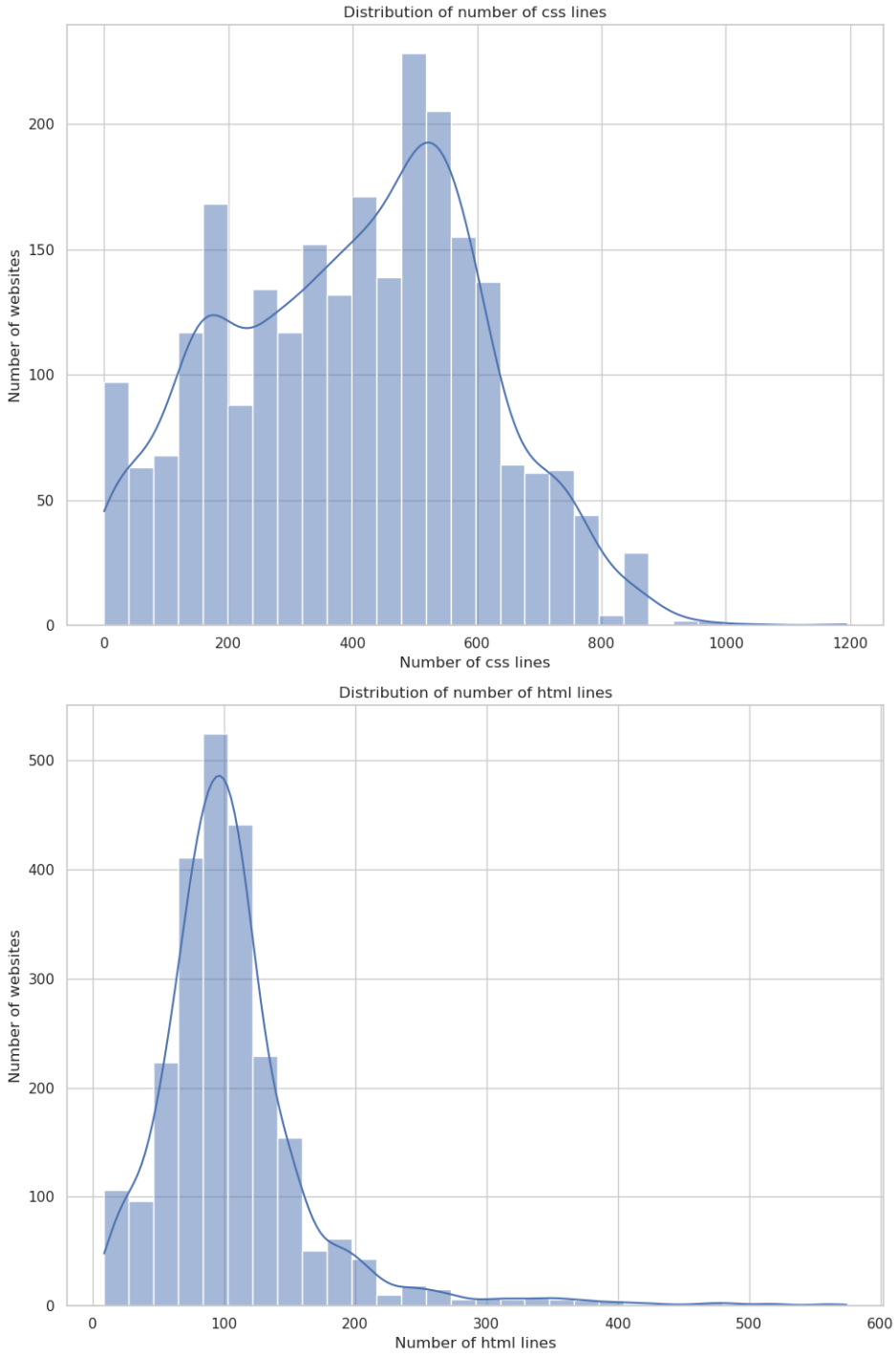

Figure 5.16: WebUI2Code-4096: distributions of CSS and HTML numbers of lines prior to data cleaning.

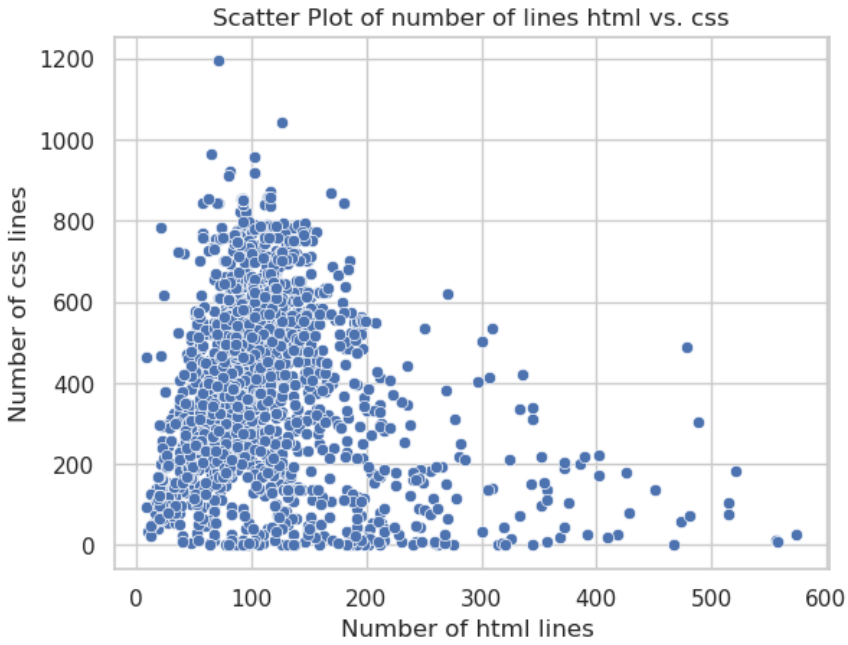

Figure 5.17: WebUI2Code-4096: scatter plot illustrating the distribution of CSS and HTML numbers of lines, prior to data cleaning.

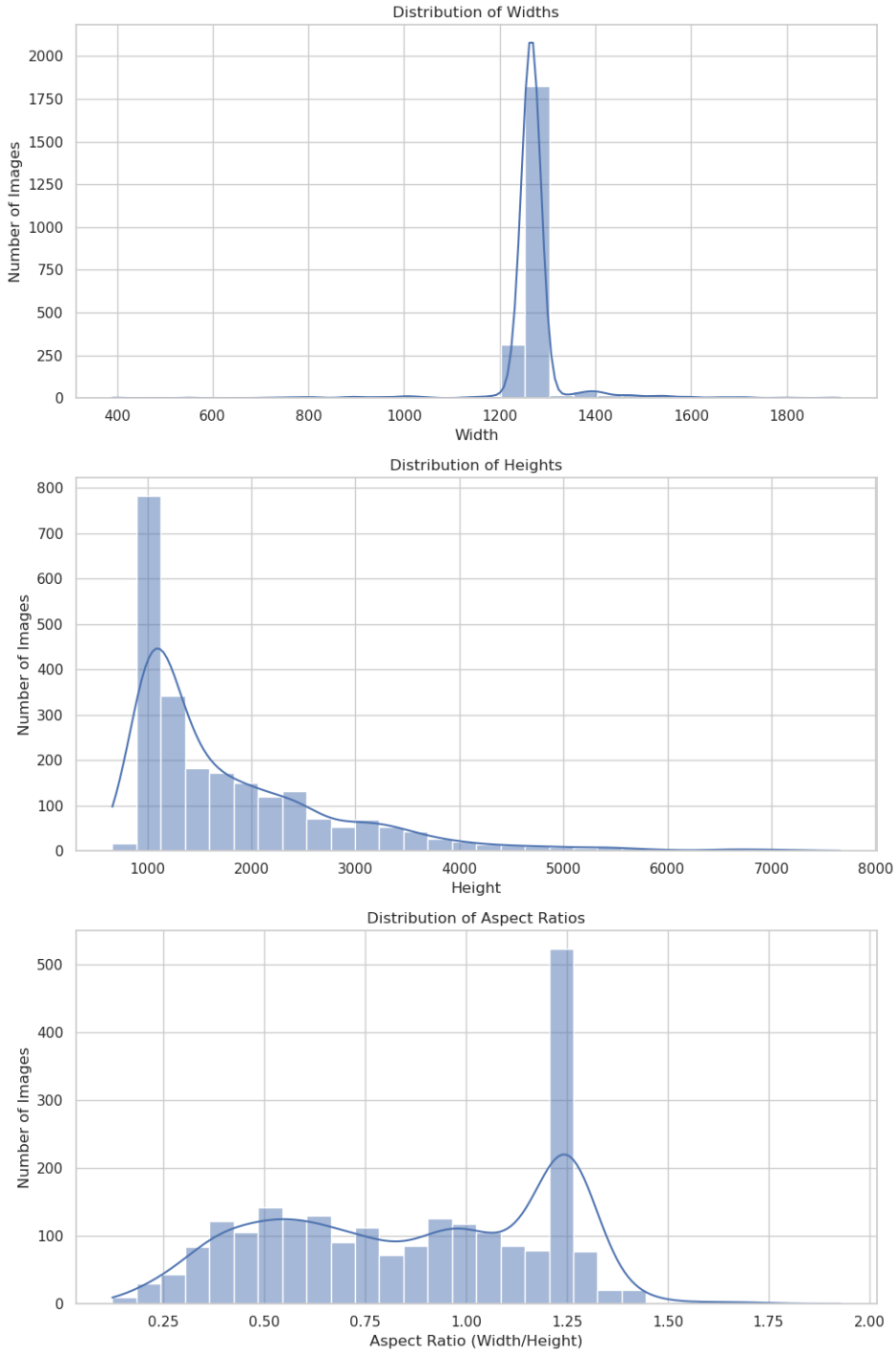

Figure 5.18: WebUI2Code-4096: distributions of image widths, heights and aspect ratios after data cleaning.

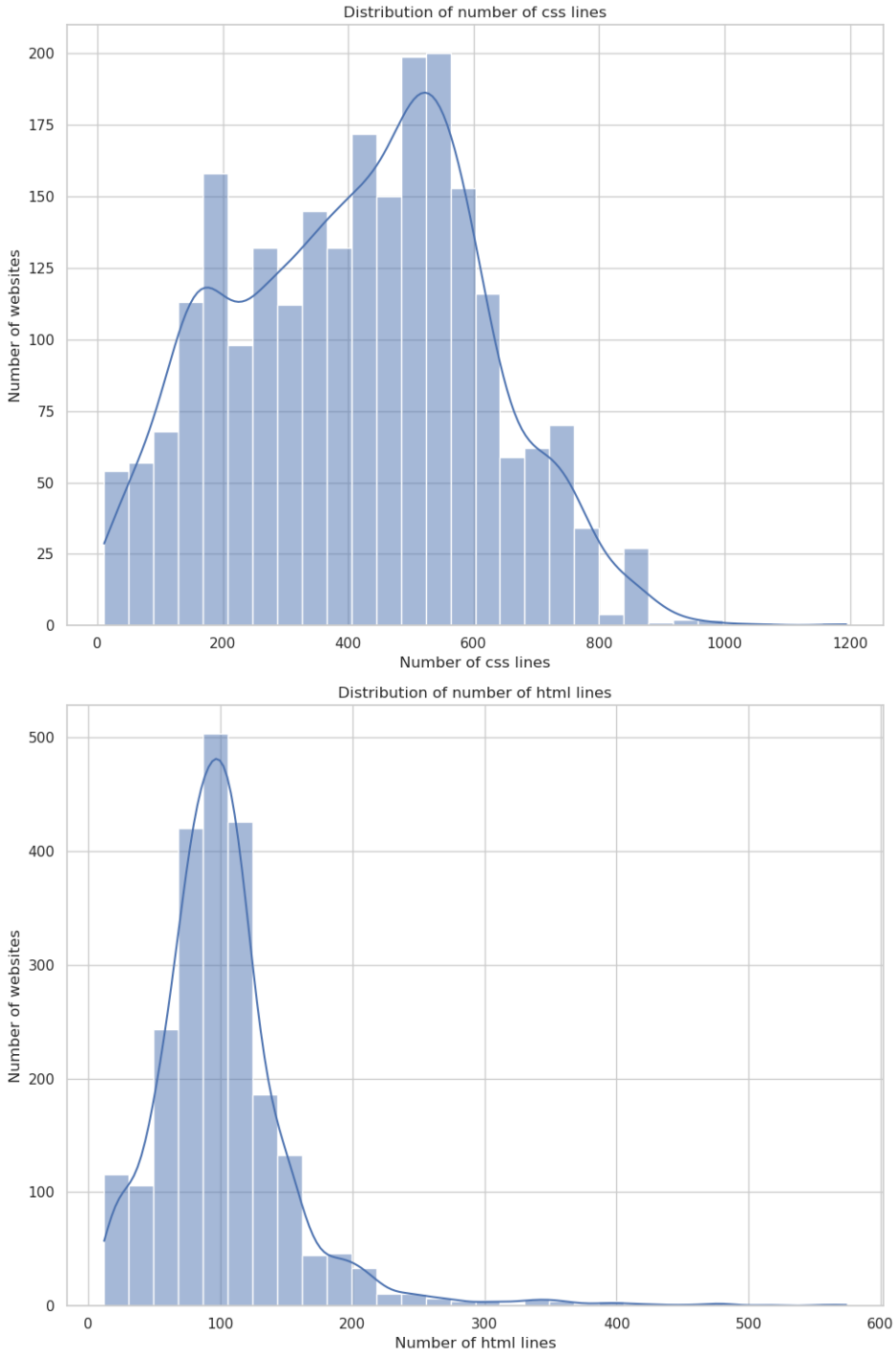

Figure 5.19: WebUI2Code-4096: distributions of CSS and HTML numbers of lines after data cleaning.

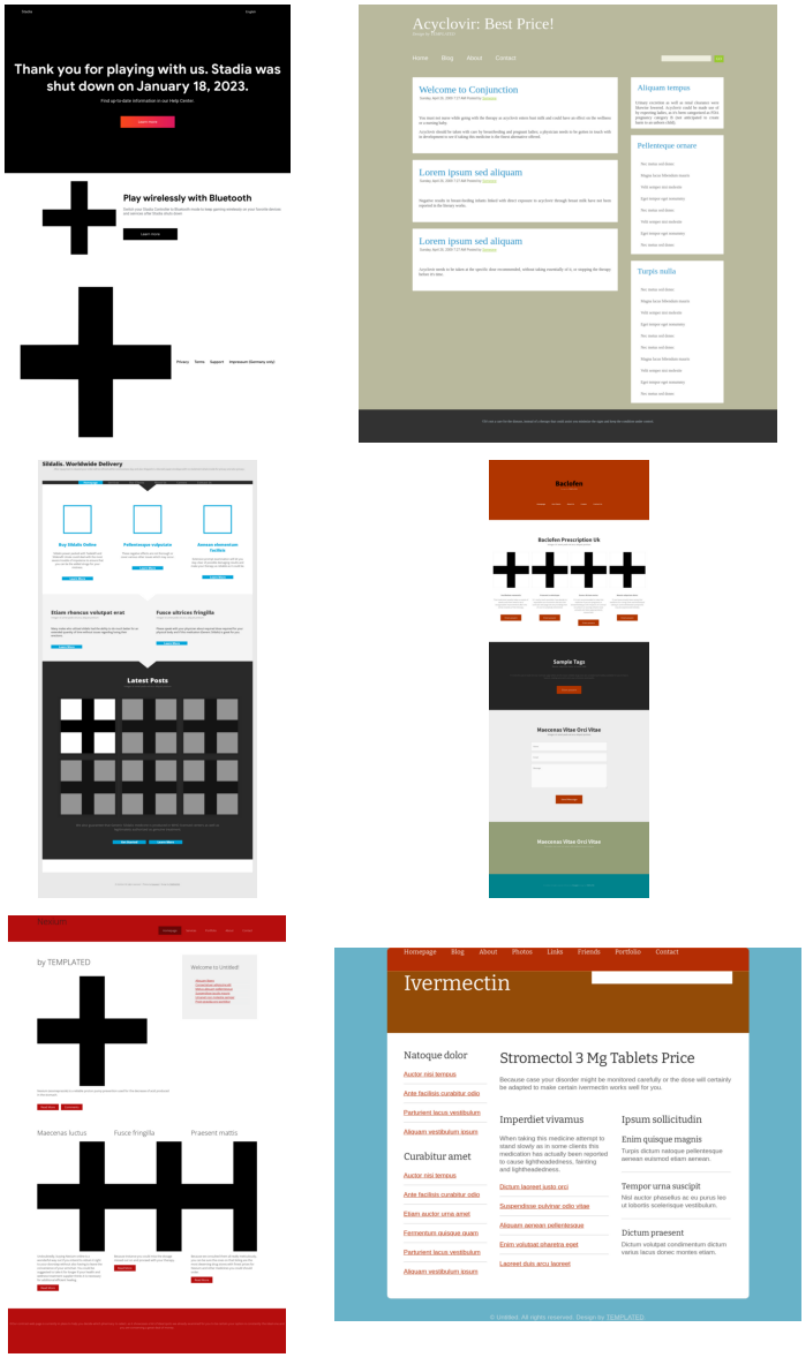

Figure 5.20: Six representative screenshots taken from the WebUI2Code-4096 Dataset.

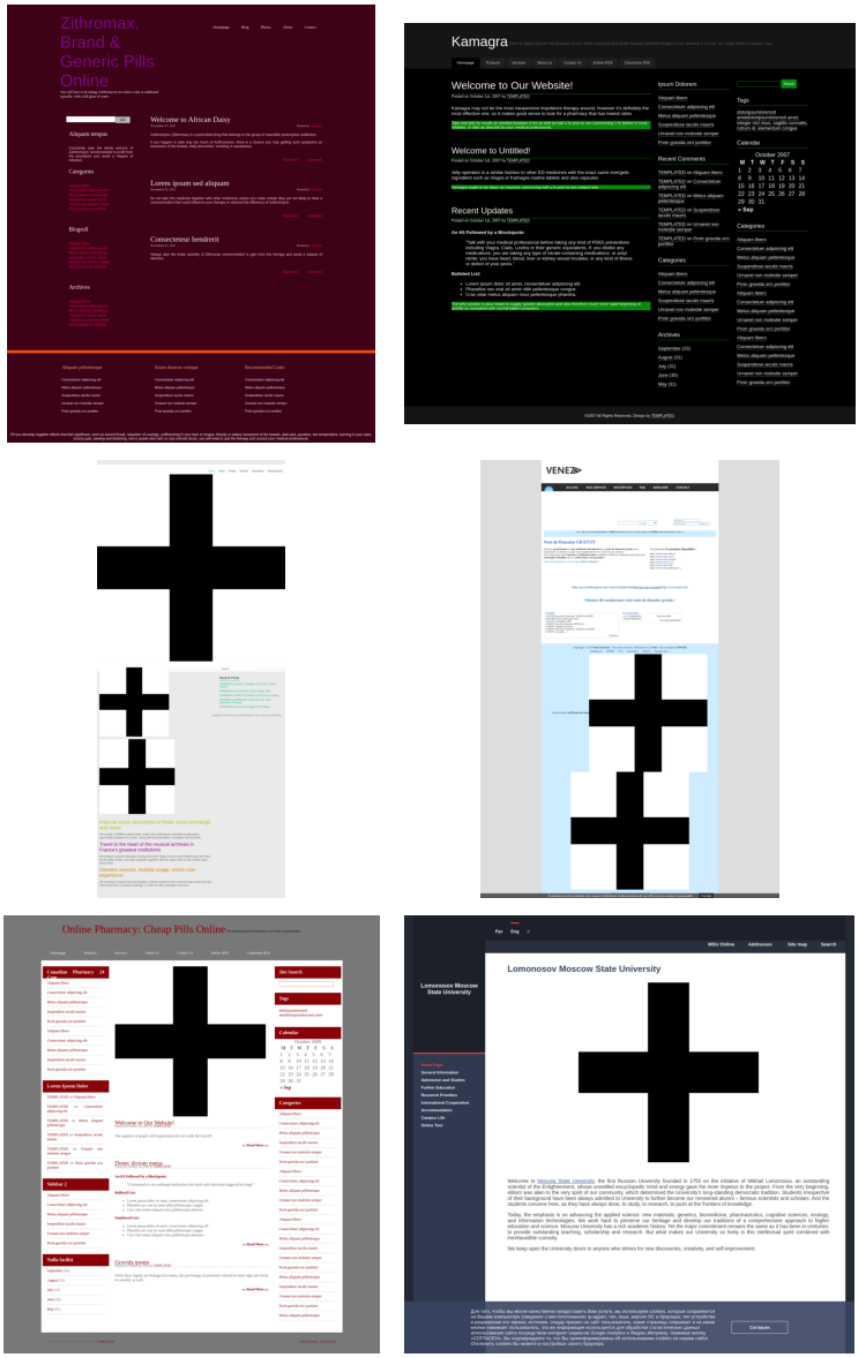

Figure 5.21: Six representative screenshots taken from the WebUI2Code-8192 Dataset.

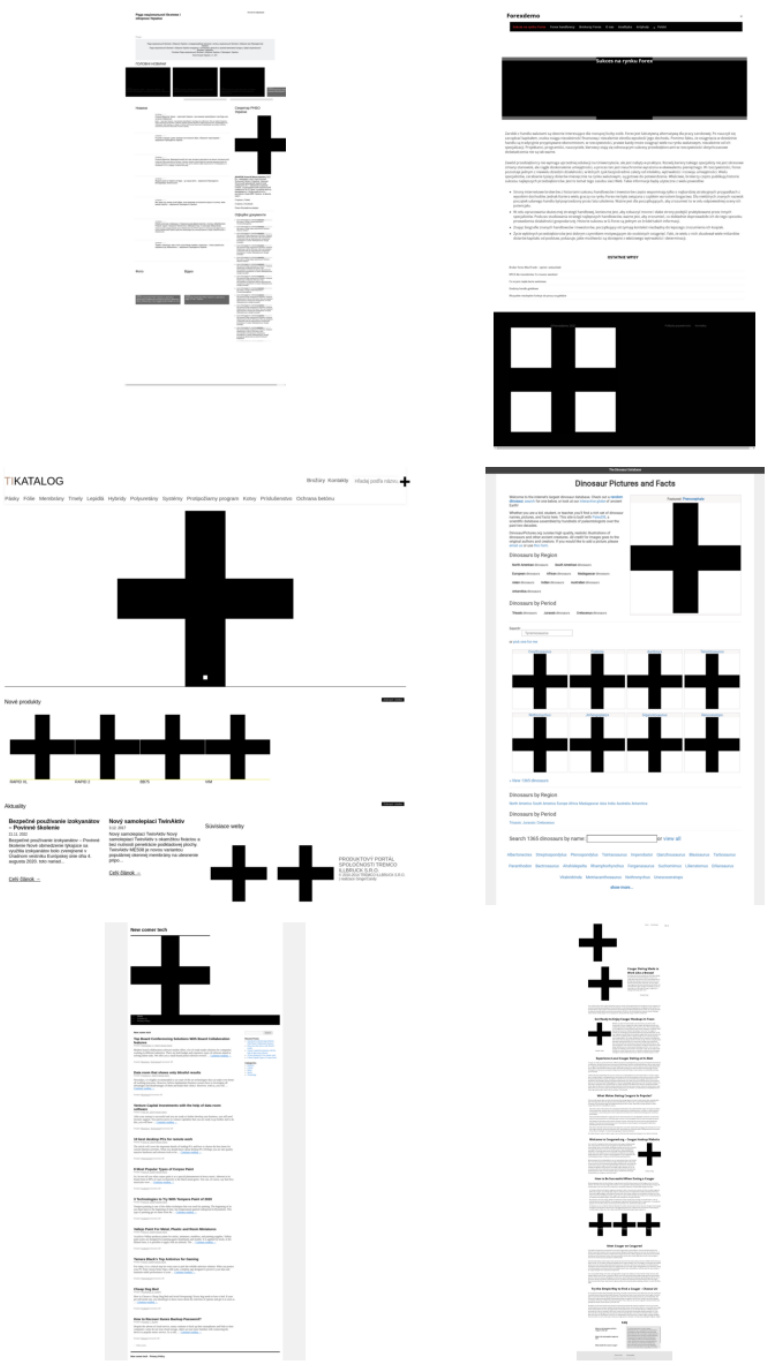

Figure 5.22: Six representative screenshots taken from the WebUI2Code-12288 Dataset.

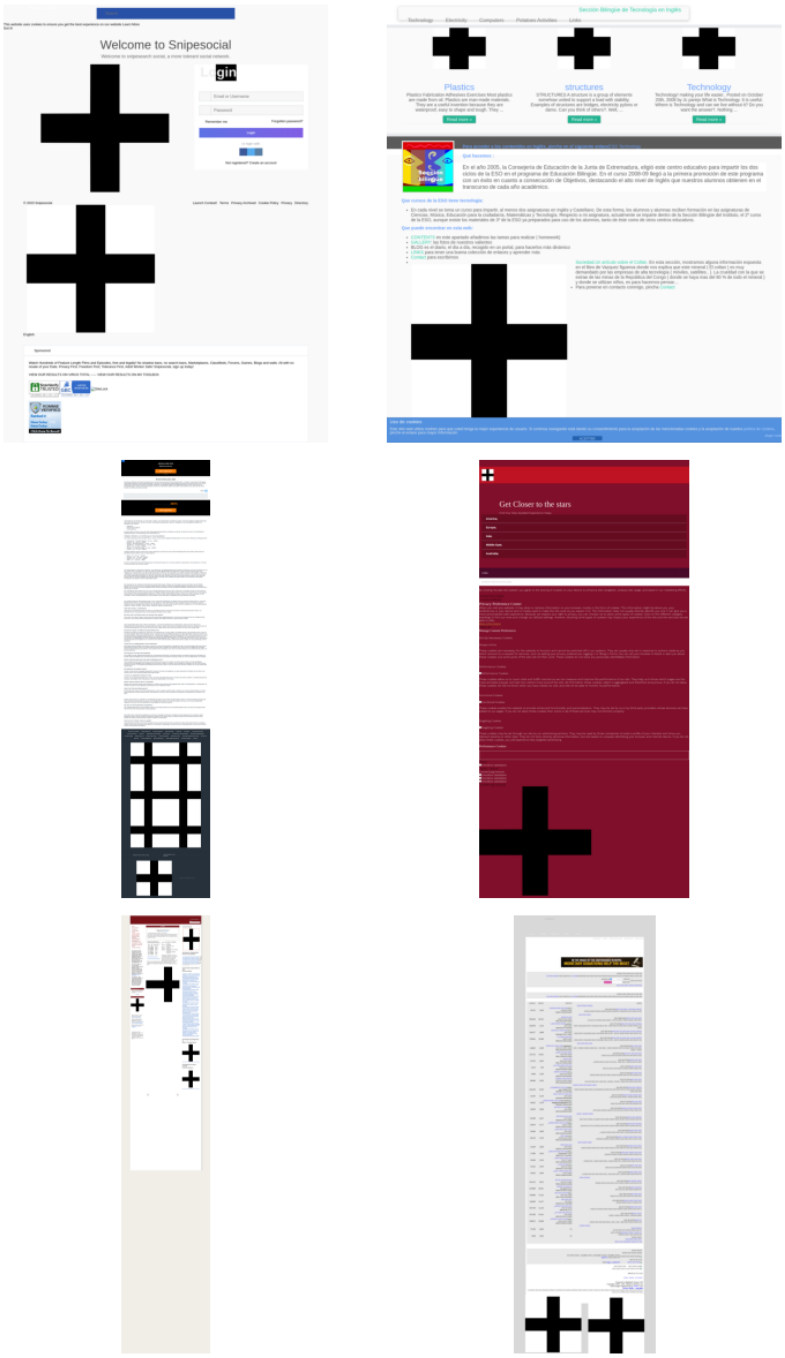

Figure 5.23: Six representative screenshots taken from the WebUI2Code-16384 Dataset.

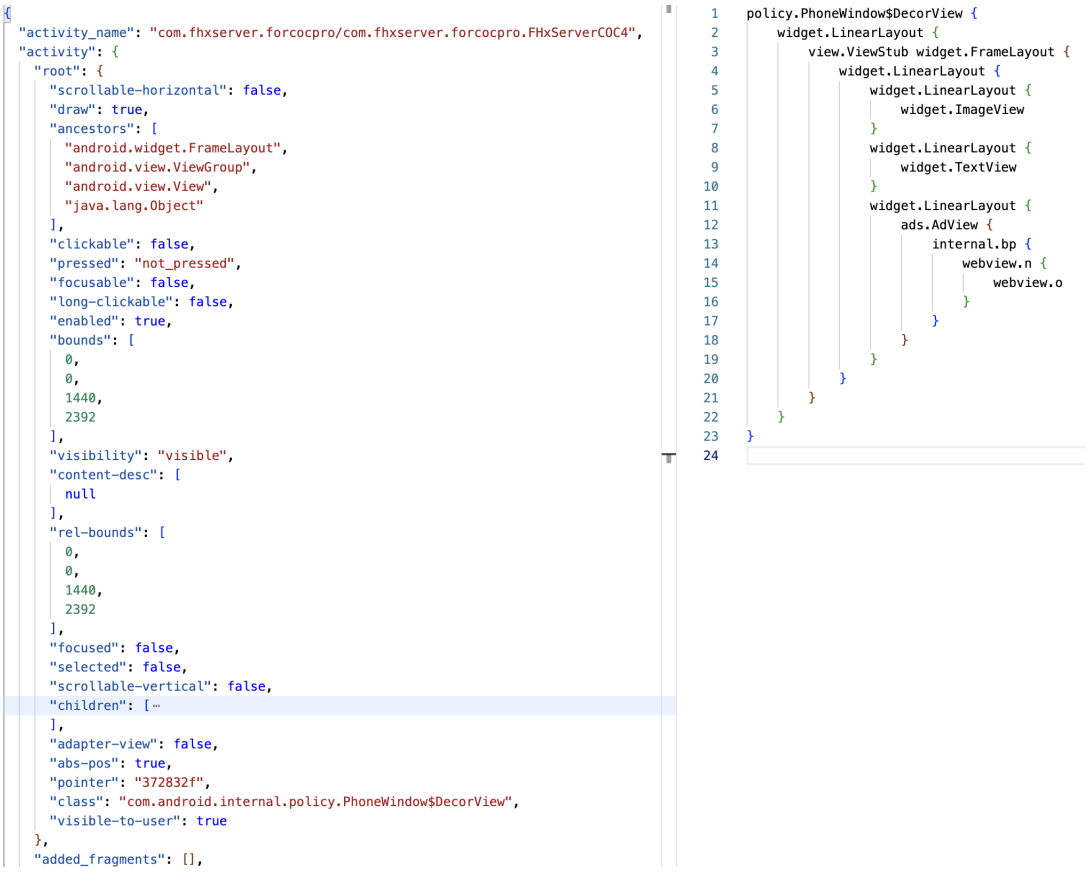

Figure 5.24: Rico: comparison between sample View Hierarchy code and the extracted structural code.

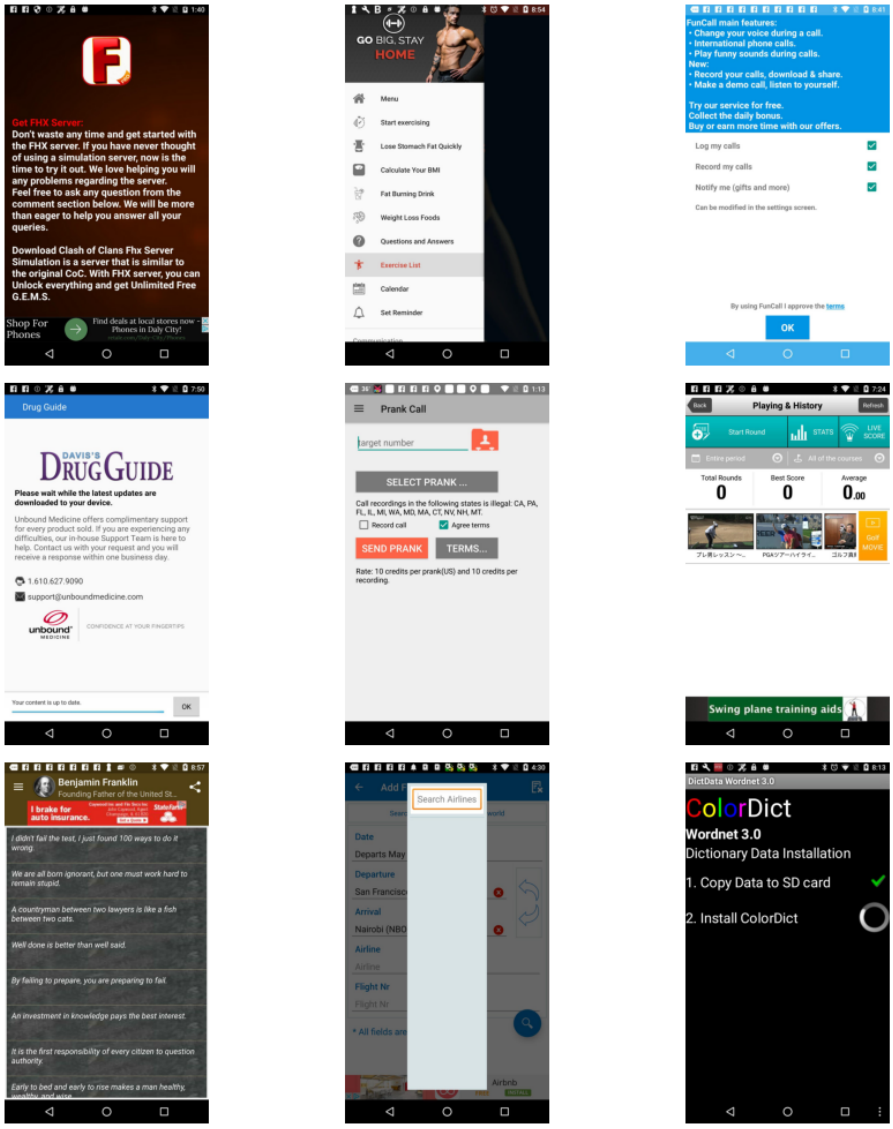

Figure 5.25: Nine representative screenshots taken from the Rico Dataset.

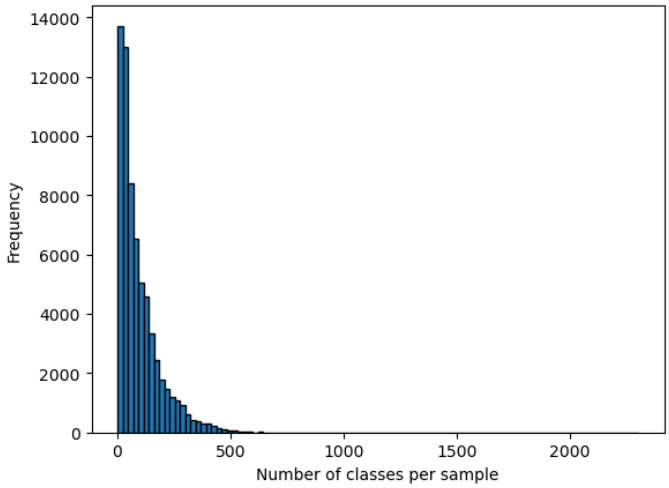

Figure 5.26: Distribution illustrating the number classes associated with various samples codes.

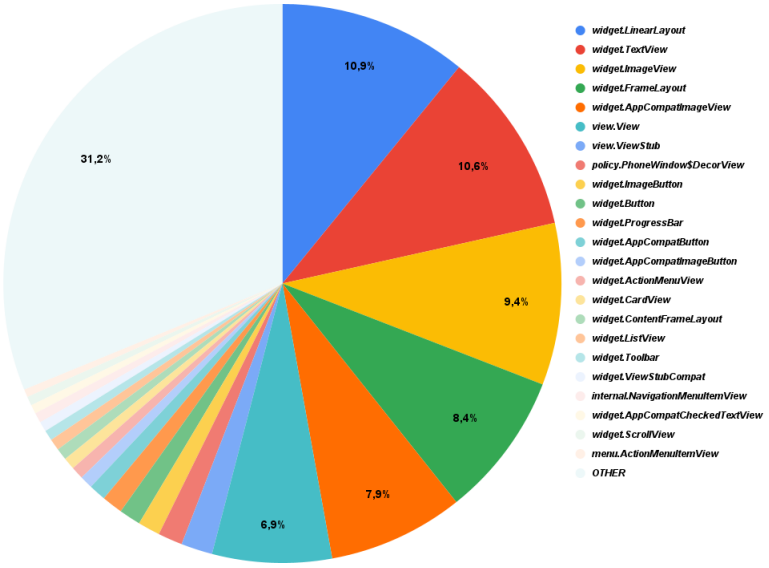

Figure 5.27: Pie chart visualizing the frequency distribution of different classes used in the Rico codes.

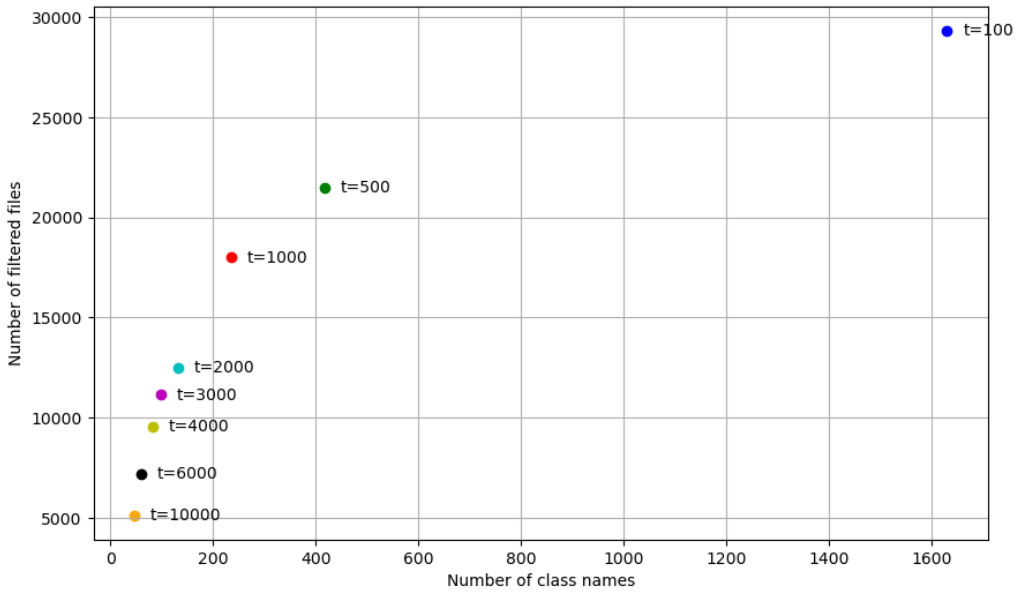

Figure 5.28: Rico: visualization of the number of filtered files corresponding to various frequency thresholds for valid class names.

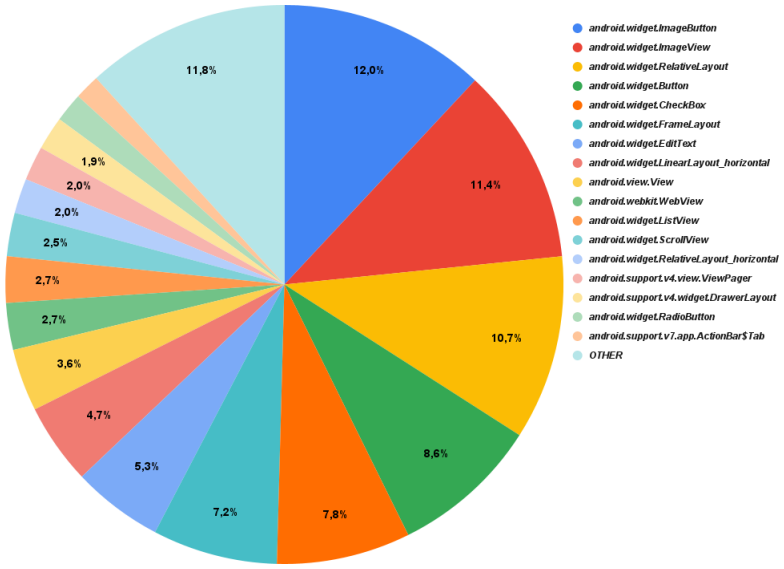

Figure 5.29: Pie chart visualizing the frequency distribution of different elements used in the UI2Code codes.

```
android.widget.FrameLayout android.widget.RelativeLayout {
    android.widget.TextView
}
android.widget.FrameLayout {
    android.widget.TextView android.widget.TextView android.widget.TextView android.widget.TextView
}
android.widget.TextView android.widget.ListView {
    android.widget.RelativeLayout {
        android.widget.TextView android.widget.TextView
    }
    android.widget.RelativeLayout {
        android.widget.TextView android.widget.TextView
    }
    android.widget.RelativeLayout {
        android.widget.TextView android.widget.TextView
    }
}
```

Figure 5.30: Code of a sample taken from UI2Code Dataset

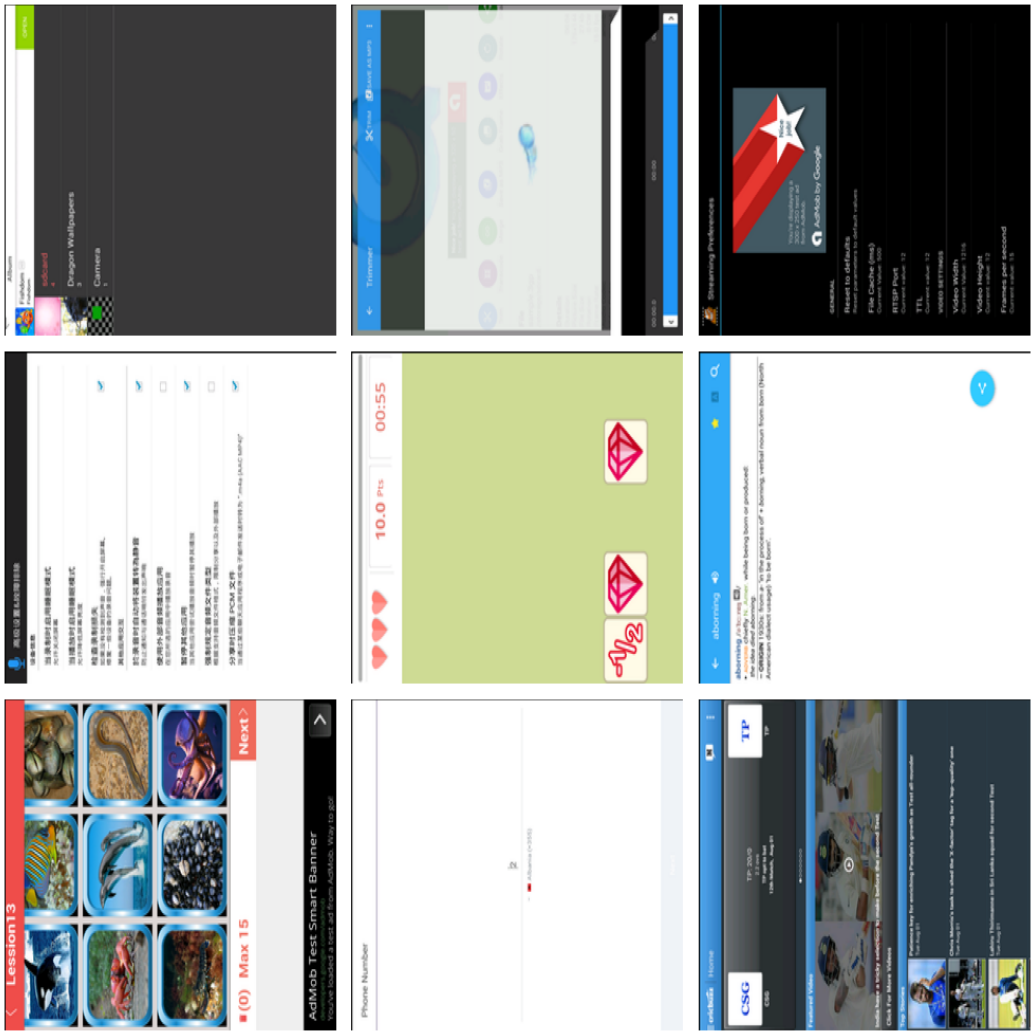

Figure 5.31: Nine representative screenshots taken from the UI2Code Dataset (original orientation).

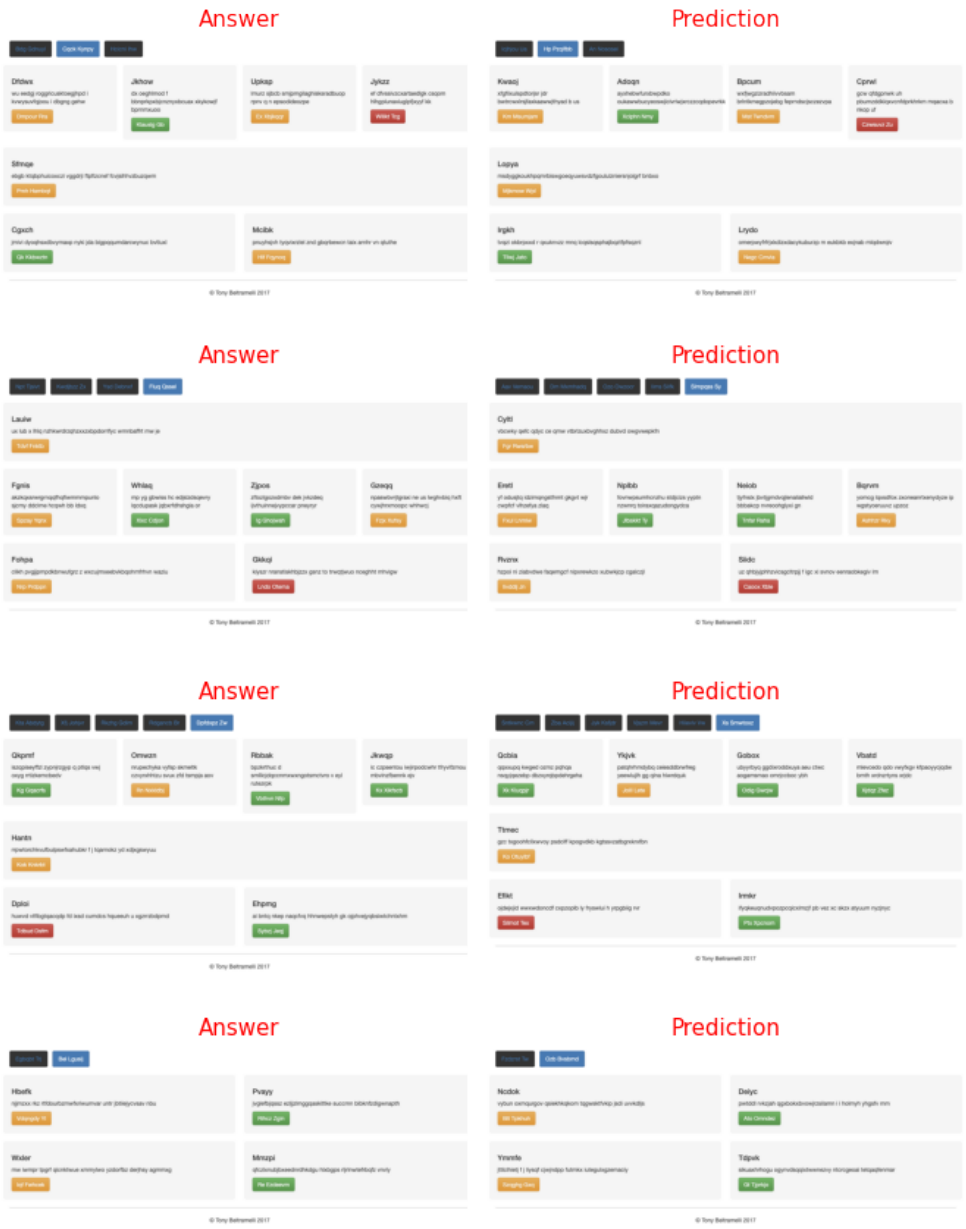

Figure 5.32: Comparison of answers and predictions screenshots from the Pix2Code Dataset's test set.

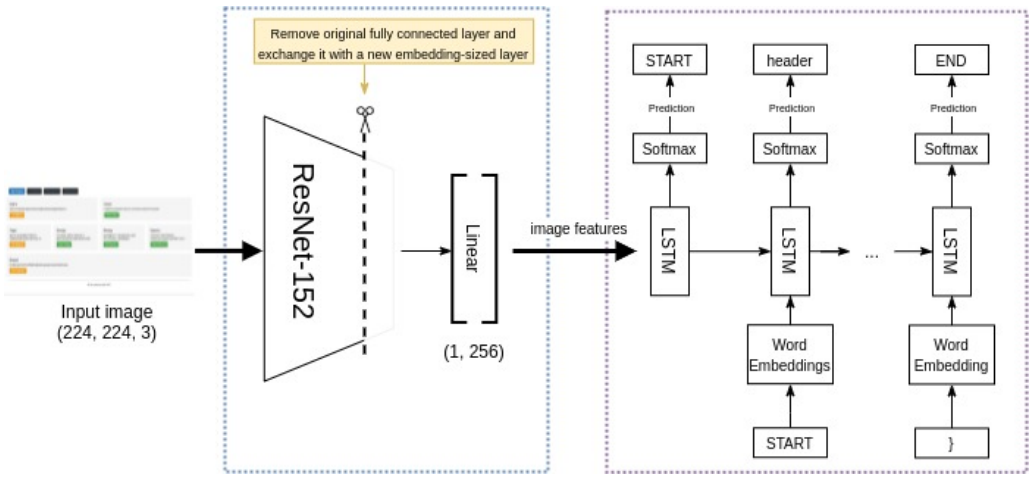

Figure 5.33: Architecture schema of Pix2Code Pytorch model
Source: `https://github.com/timoangerer/pix2code-pytorch`

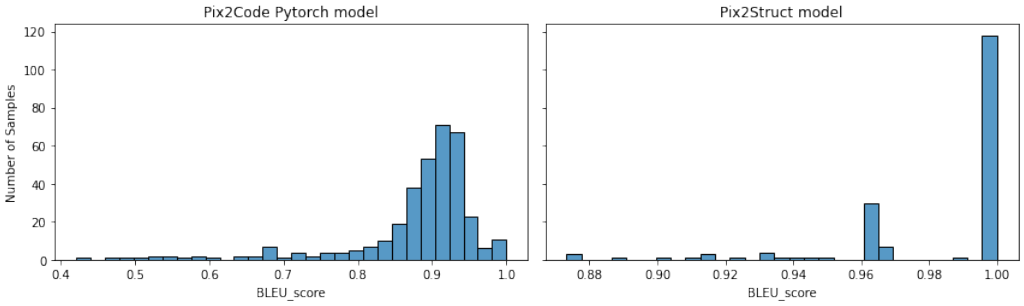

Figure 5.34: Comparison of BLEU score distributions between the LSTM-based model and Pix2Struct for Pix2Code's test set.

ANSWER vs PREDICTION

Start:

```
<html> <body> <main class="container"> <div class="header clearfix"> <nav> <ul class="nav …
<html> <body> <main class="container"> <div class="header clearfix"> <nav> <ul class="nav …
```

Middle:

```
… <a href="#">Jxayks Ssm</a></li> <li><a href="#">Hvc Ckaqvg</a></li><li><a href="#"> …
… <a href="#">Xs Sssssss</a></li> <li><a href="#">Xs Sssssss</a></li><li><a href="#"> …
```

End:

```
… <div class="col-lg-6"> <h4>Htlvo</h4><p>ofwimqetlk adgqo kk dapsganzcay wtncfdkhvrl k
pdjldvyiw</p> <a class="btn btn-danger" href="#" role="button">Hww Wzkxhl</a> </div> <div
class="col-lg-6"> <h4>Ibzyc</h4><p>okc rwvn dzioaic kvv u qmlfsq cgvjoric tivwowhbleqpwimxj</p>
<a class="btn btn-success" href="#" role="button">Uniy Yefau</a> </div> </div> <div
class="row"><div class="col-lg-3"> …
… <div class="col-lg-6"> <h4>Htlvo</h4><p>ss s s s s s s s s s s s s s ss ss ss ss ss ss ss ss ss ss ss
ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss
ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss
ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss ss
ss ss ss ss ss ss ss ss ss ss ss ss ss ss …
```

Figure 5.35: Comparison of answers and predictions codes from the Pix2Code HTML Dataset's validation set.

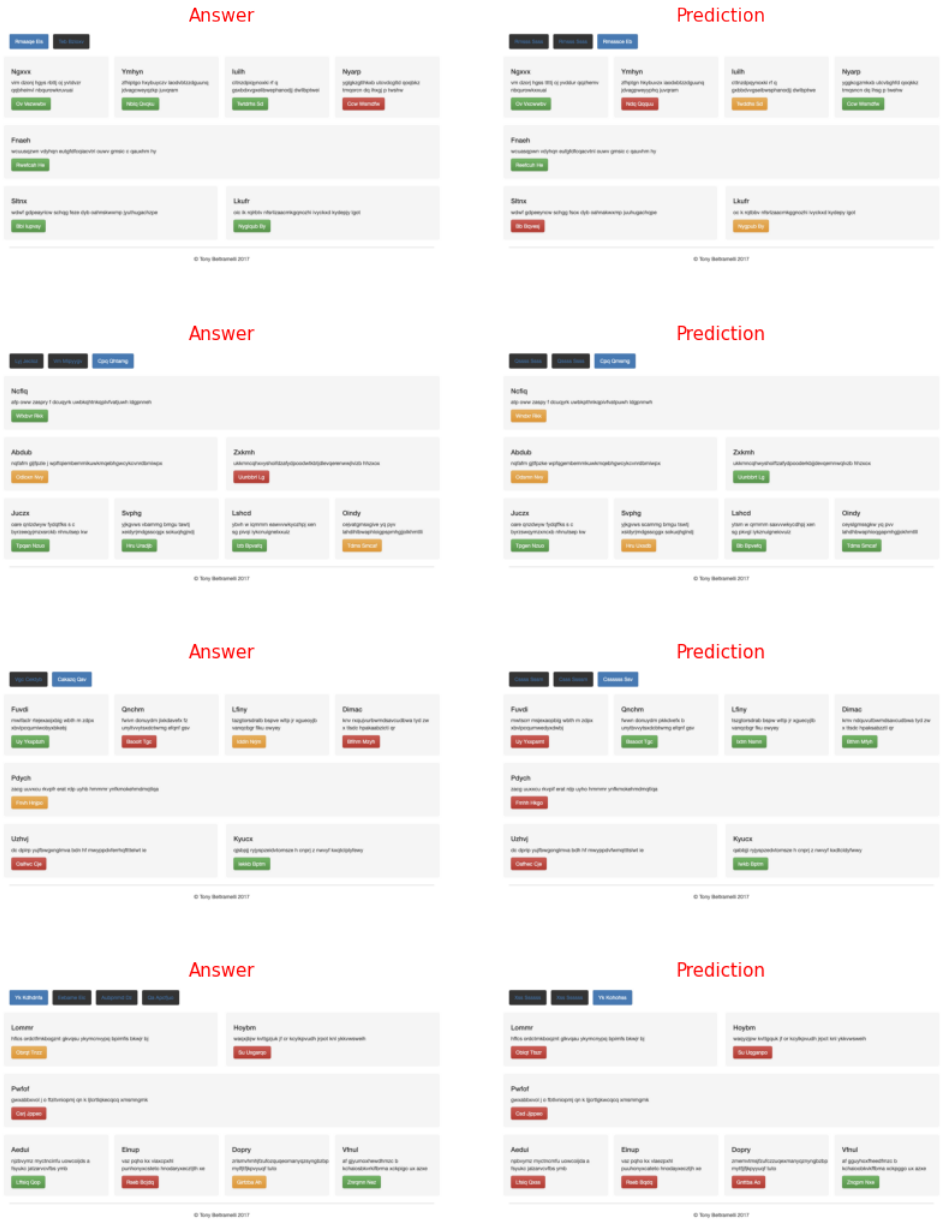

Figure 5.36: Comparison of answers and predictions screenshots from the Pix2Code HTML Dataset's validation set.

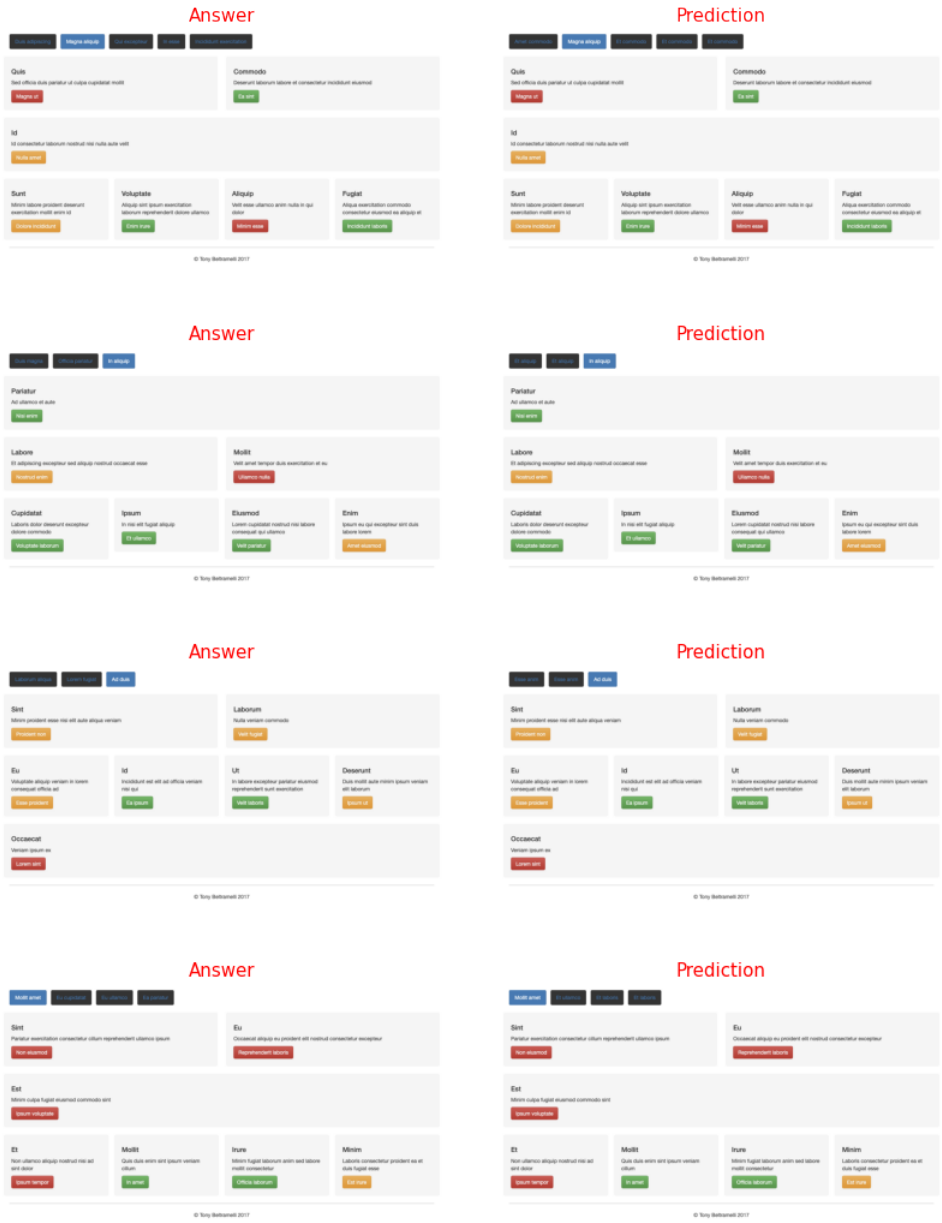

Figure 5.37: Comparison of answers and predictions screenshots from the Pix2Code HTML Lorem Ipsum Dataset's test set.

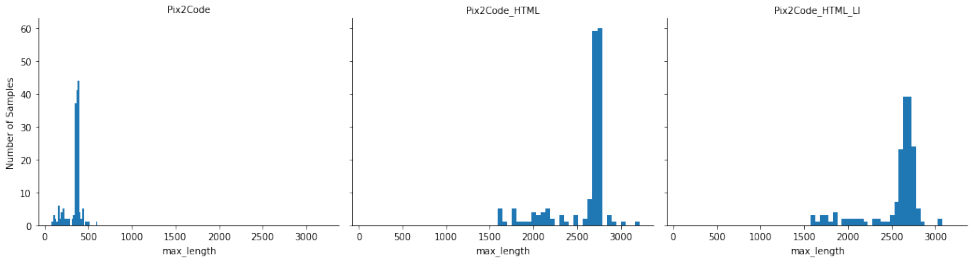

Figure 5.38: Comparison of max text length distributions across different versions of the Pix2Code Dataset.

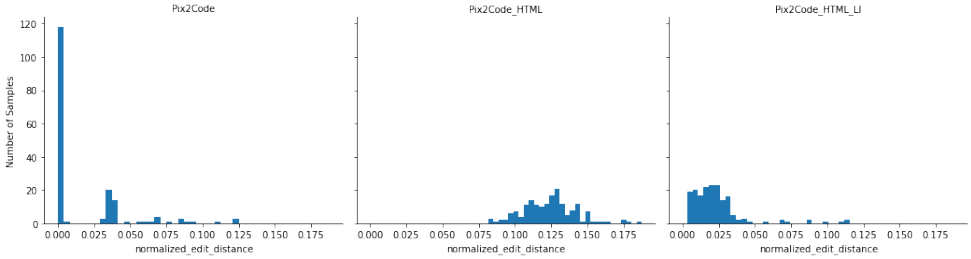

Figure 5.39: Comparison of Normalized Edit Distance distributions across different versions of the Pix2Code Dataset.

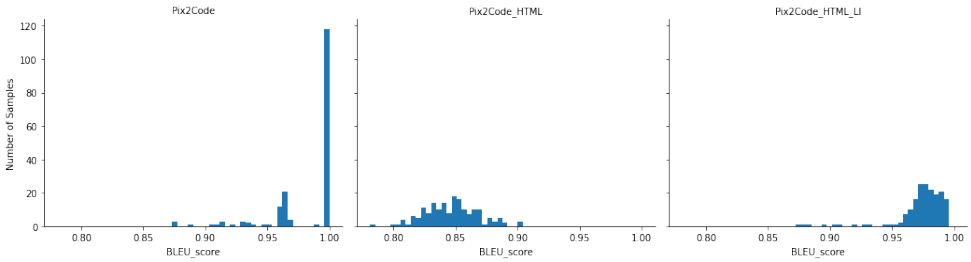

Figure 5.40: Comparison of BLEU Score distributions across different versions of the Pix2Code Dataset.

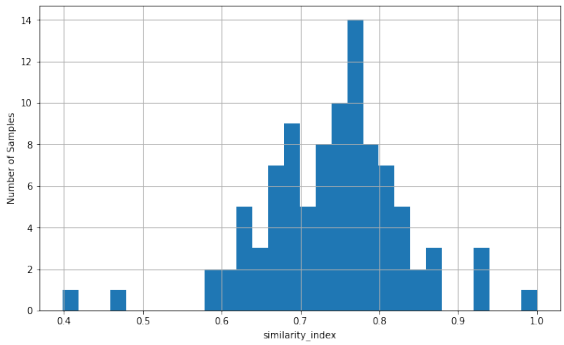

Figure 5.41: Distribution of the Structural Similarity Index (SSIM) for test set experiments on the Synthetic Bootstrap Mini Dataset.

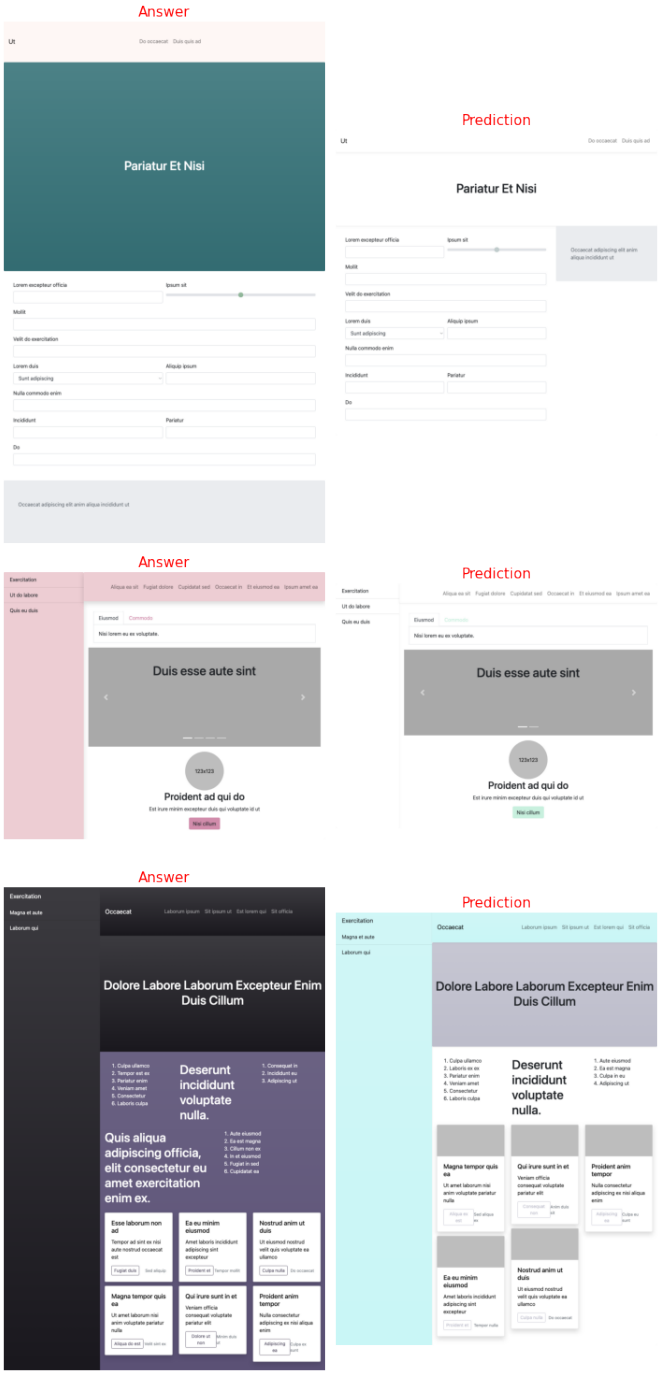

Figure 5.42: Comparison of answers and predictions screenshots from the Synthetic Bootstrap Mini Dataset's test set.

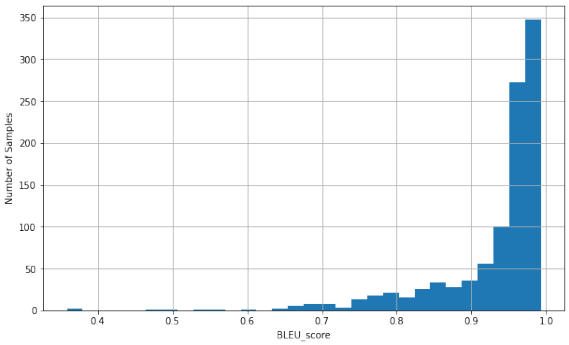

Figure 5.43: Distribution of BLEU Score for test set experiments on the Synthetic Bootstrap Dataset.

Figure 5.44: Comparison of answers and predictions screenshots from the Synthetic Bootstrap Dataset's test set.

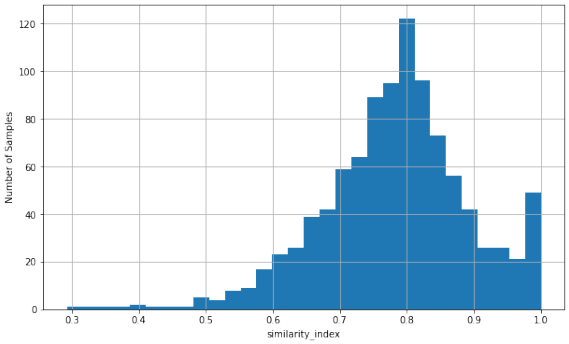

Figure 5.45: Distribution of the Structural Similarity Index (SSIM) for test set experiments on the Synthetic Bootstrap Dataset.

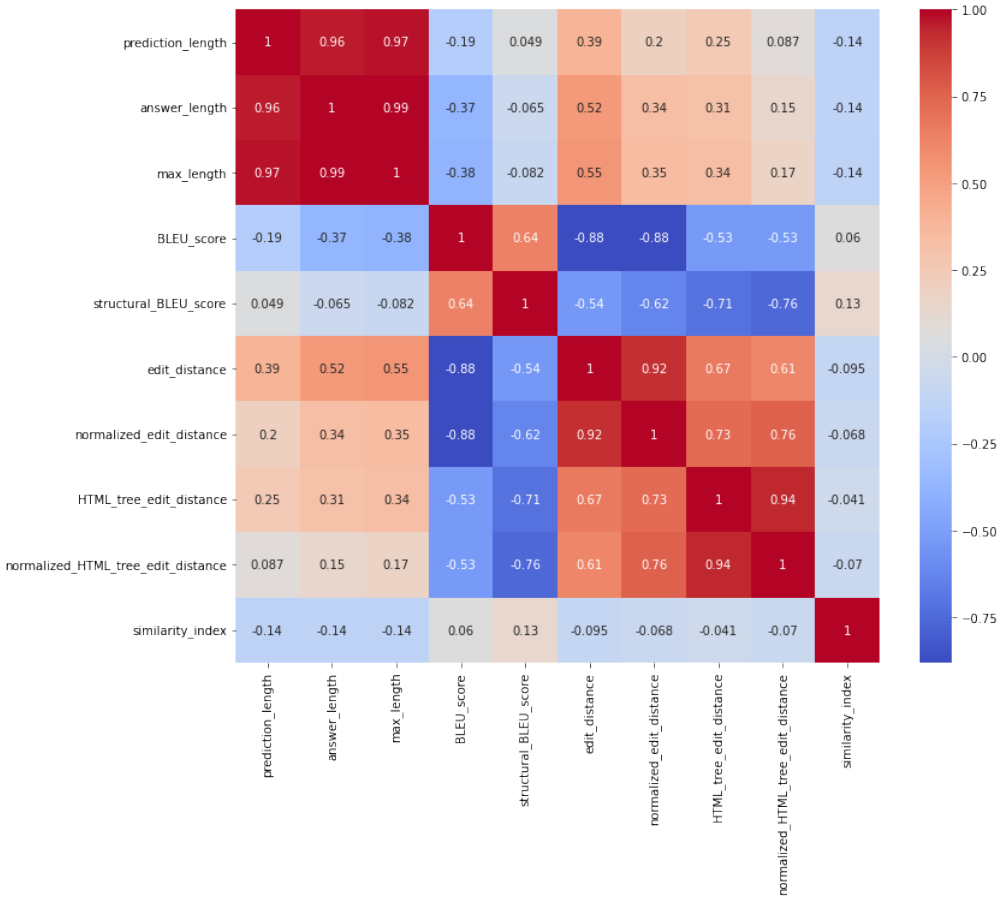

Figure 5.46: Correlation matrix for Synthetic Bootstrap Dataset's test set metrics

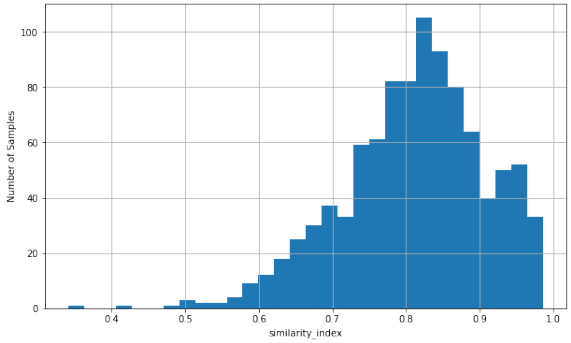

Figure 5.47: Distribution of the Structural Similarity Index (SSIM) for test set experiments on the Sketch Synthetic Bootstrap Dataset.

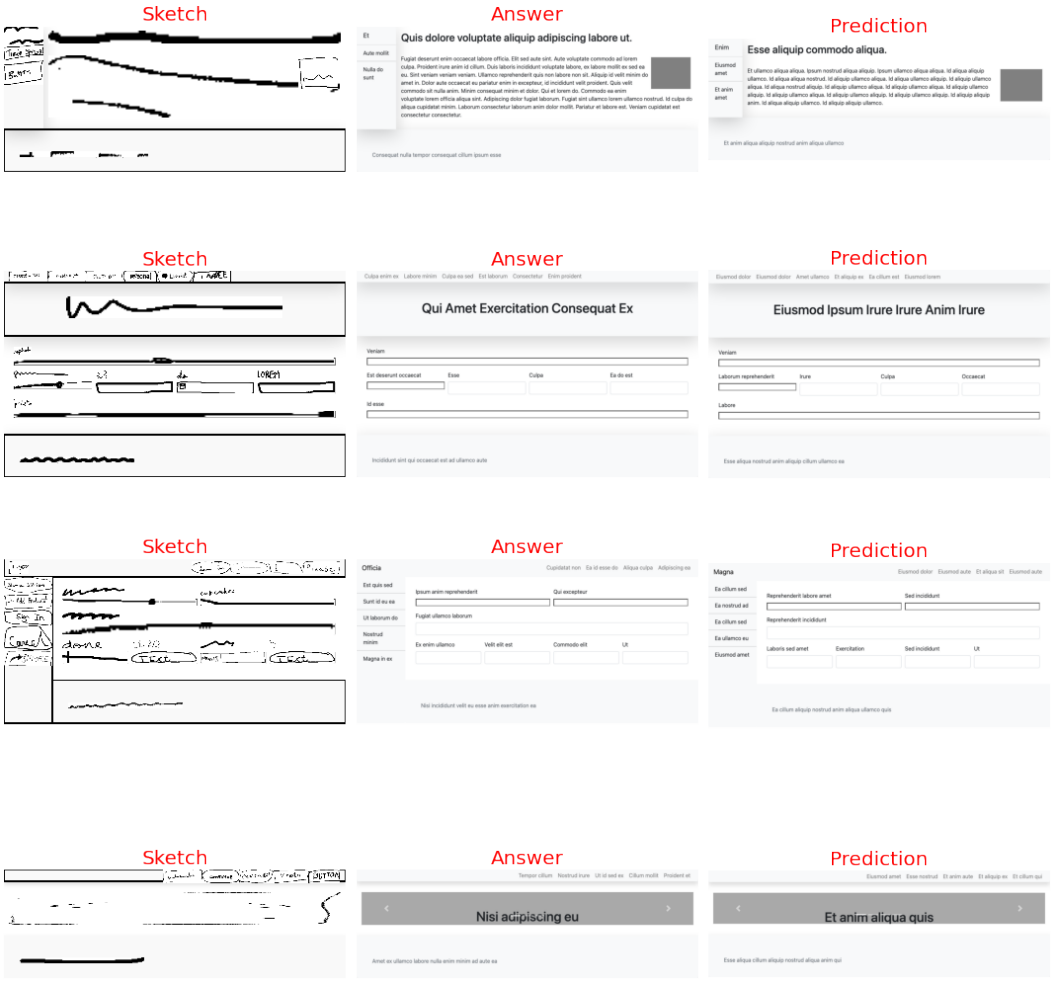

Figure 5.48: Comparison of answers and predictions screenshots from the Sketch Synthetic Bootstrap Dataset's test set.

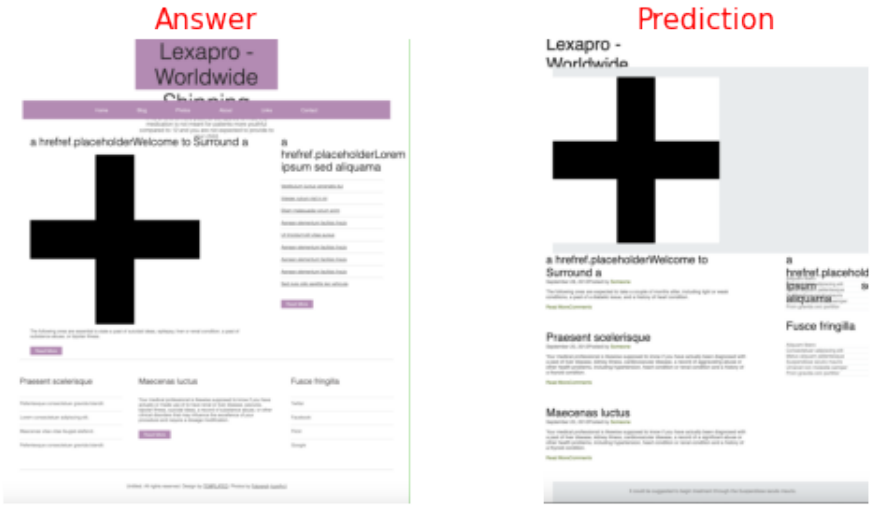

Figure 5.49: Comparison of answers and predictions screenshots from the WebUI2Code-4096 Dataset's test set.

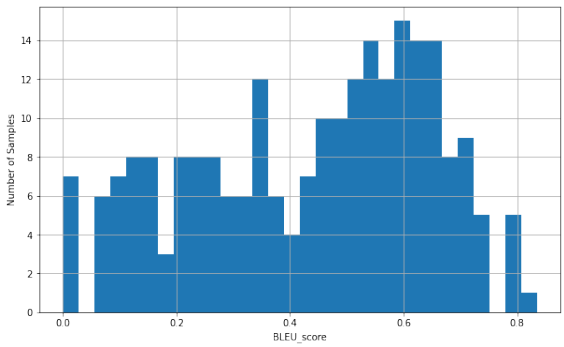

Figure 5.50: Distribution of BLEU Score for test set experiments on the WebUI2Code-4096 Dataset.

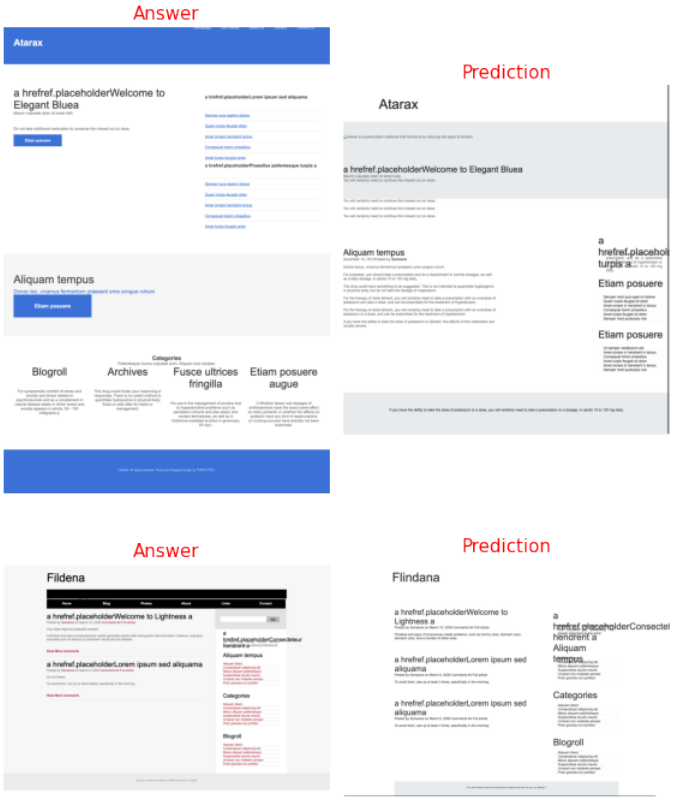

Figure 5.51: Comparison of answers and predictions screenshots from the WebUI2Code-4096 Dataset's test set.

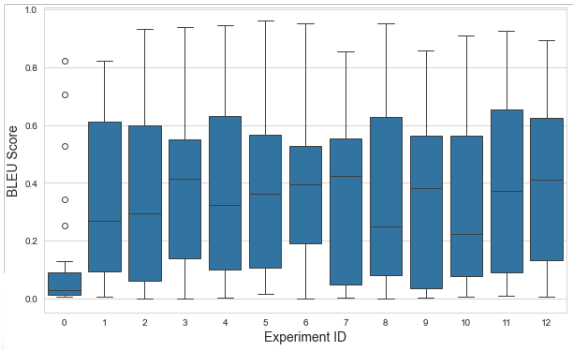

Figure 5.52: Distribution of BLEU Score for validation set experiments on the Rico Dataset using sampling.

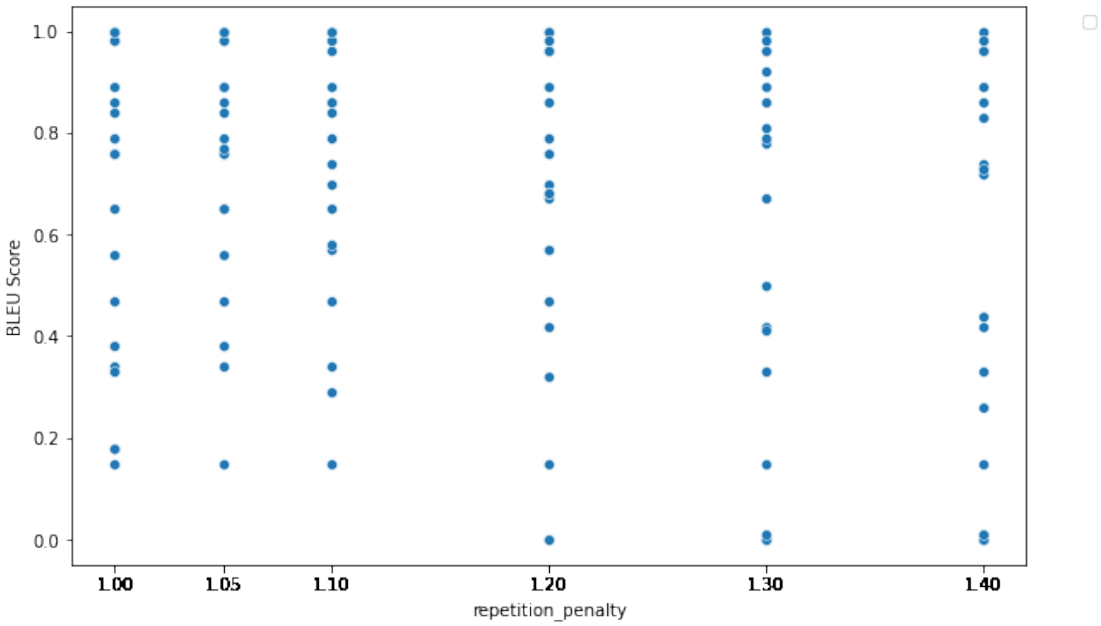

Figure 5.53: Visual representation of BLEU Scores values for UI2Code's Dataset validation set samples

Chapter 6

Conclusions

The thesis aimed to achieve two main objectives: firstly, to use web scraping techniques to generate a public dataset of websites reflecting real-world complexity, and secondly, to employ a transformer-based model for the automatic generation of website code.

The first part of this research focused on creating an automated pipeline to extract code and screenshots from websites while minimizing noise. This tool was subsequently utilized to construct **WebUI2Code**: a dataset comprising more than 34000 samples, derived from 100000 websites. All challenges encountered when dealing with scraping tools and real websites were described. A variety of techniques and strategies were used to reduce the samples' code while ensuring quality.

The second part investigated applying Pix2Struct to website code generation tasks. It was fine-tuned using various datasets, including a public simple synthetic dataset called Pix2Code, a new more complex synthetic dataset of HTML Bootstrap websites, and its variant with hand-written sketches. Additionally, it was tested on the newly created WebUI2Code dataset and on other real-world datasets of mobile application UIs, like RICO and UI2Code.

Overall, the model can be considered suitable for the automatic generation of synthetic websites, being able to improve performance on Pix2Code compared to other models, and to obtain good results also for other, more complex datasets. The transition to real-world data saw a drop in performance and the emergence of new hurdles, like the prediction of repetitive phrases. An initial exploration of different hyper-parameters managed to mitigate this problem and indicates that additional training and experimentation might further improve the results.

Exploring the complexities of extensive transformer models has revealed the challenges brought forth by these powerful architectures, especially in scenarios bound by time, cost, and hardware resources.

These limitations have constrained the usage to a smaller portion of the available

data, highlighting an opportunity for future research to enhance model performance and applicability in real-world website code generation.

In conclusion, by implementing an automated website extraction pipeline, the generation of large and diverse datasets has become achievable. This advancement may enable more comprehensive experiments and promote the development of improved transformer models, particularly for website code generation.

	Pix2Struct	P2C	UI2C
Pix2Code	0.98	0.88	-
Pix2Code HTML	0.85	-	-
Pix2Code HTML LI	0.97	-	-
Synthetic Bootstrap	0.93	-	-
Sketch Bootstrap	0.83	-	-
WebUI2Code-4096	0.44	-	-
RICO	0.42	-	-
UI2Code	0.74	-	0.79

Table 6.1: Summary of test results across all the datasets: mean BLEU Score

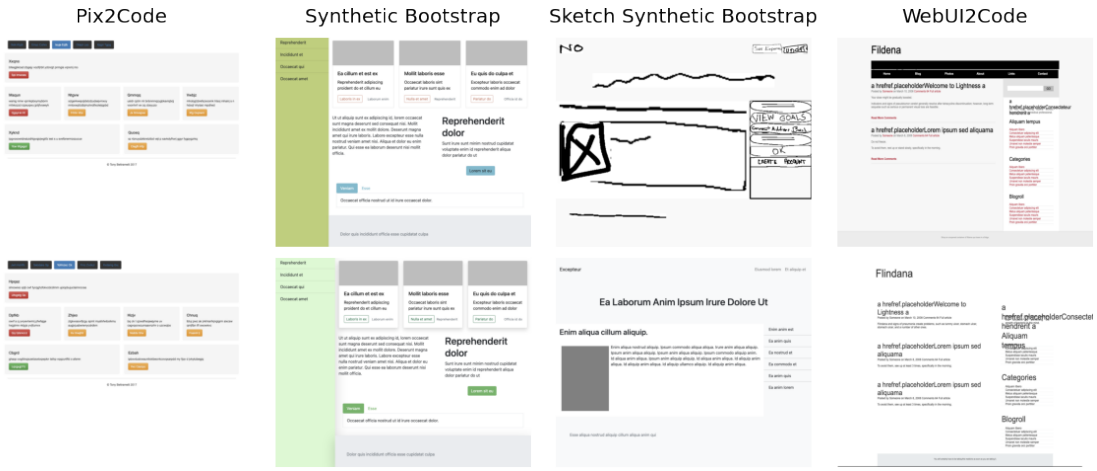

Figure 6.1: Comparison of ground truth screenshots(1st row) with corresponding predictive outputs (2nd row) across experiments on various datasets

Bibliography

[1] G.J. James. *The Elements of User Experience: User-Centered Design for the Web and beyond*. 2nd ed. Pearson Education, 2010 (cit. on p. 1).

[2] R. Hartson and P.S. Pyla. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience*. 2nd ed. Morgan Kaufmann, 2019 (cit. on p. 1).

[3] T. Memmel and H. Reiterer. «Support Collaboration, Model-based and prototyping-driven user interface specification to and creativity». In: *Int. J. Univ. Comput. Sci.* 14.19 (2009), pp. 3217–3235 (cit. on p. 1).

[4] T. Beltramelli. «pix2code: Generating code from a graphical user interface screenshot». In: *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 2018, pp. 1–6 (cit. on pp. 3, 6, 9, 10, 47, 48, 68).

[5] T. Beltramelli. *Hack your design sprint: Wireframes to prototype in under 5 minutes*. 2019. URL: `https://uizard.io/` (cit. on p. 6).

[6] Y. Han, J. He, and Q. Dong. «CSSSketch2Code: An automatic method to generate web pages with CSS style». In: *Proceedings of the 2nd International Conference on Advances in Artificial Intelligence*. Spain, 2018, pp. 29–35 (cit. on pp. 6, 10).

[7] A. Kumar. *Automated front-end development using deep learning*. 2018. URL: `https://blog.insightdatascience.com/automated-front-end-development-usingdeep-learning-3169dd086e82` (cit. on pp. 7, 10).

[8] Y. Xu, L. Bo, X. Sun, B. Li, J. Jiang, and W. Zhou. «image2emmet: Automatic code generation from web user interface image». In: *J. Softw.: Evol. Process* 33.8 (2021), e2369 (cit. on pp. 7, 10).

[9] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. «From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation». In: *Proceedings of the 40th International Conference on Software Engineering*. ACM. New York, NY, USA, 2018, pp. 665–676 (cit. on pp. 7, 10, 11, 81).

[10] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. «Learning to infer graphics programs from hand-drawn images». In: *Advances in Neural Information Processing Systems*. 2018, pp. 6059–6068 (cit. on p. 7).

[11] Y. Deng, A. Kanervisto, J. Ling, and A.M. Rush. «Image-to-markup generation with coarse-to-fine attention». In: *International Conference on Machine Learning*. PMLR. 2017, pp. 980–989 (cit. on p. 7).

[12] Kim Bada, Sangmin Park, Taeyeon Won, Junyoung Heo, and Bongjae Kim. «Deeplearning based web UI automatic programming». In: *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems*. ACM, 2018, pp. 64–65 (cit. on p. 7).

[13] B. Aşıroğlu, B.R. Mete, E. Yıldız, Y. Nalçakan, A. Sezen, M. Dağtekin, and T. Ensari. «Automatic HTML code generation from mock-up images using machine learning techniques». In: *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science, EBBT*. IEEE, 2019, pp. 1–4 (cit. on p. 7).

[14] A. Halbe and A.R. Joshi. «A novel approach to HTML page creation using neural network». In: vol. 45. International Conference on Advanced Computing Technologies and Applications (ICACTA). 2015, pp. 197–204. URL: `http://www.sciencedirect.com/science/article/pii/S1877050915003580` (cit. on p. 7).

[15] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu. «Automated cross-platform GUI code generation for mobile apps». In: *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 2019, pp. 13–16 (cit. on p. 7).

[16] J. Ferreira, A. Restivo, and H.S. Ferreira. «Automatically generating websites from hand-drawn mock-ups». In: *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*. 2021 (cit. on p. 7).

[17] Saad Hassan, Manan Arya, Ujjwal Bhardwaj, and Silica Kole. «Extraction and classification of user interface components from an image». In: *Int. J. Pure Appl. Math.* 118.24 (2018), pp. 1–16 (cit. on p. 7).

[18] P. Myznikov and Y. Huang. «A new method for hierarchical image segmentation from visual designs». In: *2020 54th Annual Conference on Information Sciences and Systems, CISS*. IEEE, 2020, pp. 1–6 (cit. on p. 7).

[19] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao. «Iconintent: Automatic identification of sensitive UI widgets based on icon classification for android apps». In: *2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE*. IEEE, 2019, pp. 257–268 (cit. on p. 7).

[20] S. Kim, J. Park, J. Jung, S. Eun, Y.S. Yun, and S. So. «Identifying UI widgets of mobile applications from sketch images». In: *J. Eng. Appl. Sci.* 13.6 (2018), pp. 1561–1566 (cit. on p. 7).

[21] H. Pham, T. Nguyen, P. Vu, and T. Nguyen. «Toward mining visual log of software». In: (2016). arXiv preprint arXiv:1610.08911 (cit. on p. 7).

[22] Young-Sun Yun, Jinman Jung, Seongbae Eun, Sun-Sup So, and Junyoung Heo. «Detection of GUI elements on sketch images using object detector based on deep neural networks». In: *Proceedings of the Sixth International Conference on Green and Human Information Technology*. Singapore: Springer Singapore, 2019, pp. 86–90 (cit. on p. 7).

[23] Microsoft AI labs. *Sketch2Code*. 2019. URL: `https://sketch2code.azurewebsites.net/` (cit. on p. 7).

[24] Benjamin Wilkins. *Sketching interfaces - airbnb design*. 2017. URL: `https://airbnb.design/sketching-interfaces/` (cit. on p. 7).

[25] M. Bajammal, D. Mazinanian, and A. Mesbah. «Generating reusable web components from mock-ups». In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2018, pp. 601–611 (cit. on p. 7).

[26] S. Suleri, V.P. Sermuga Pandian, S. Shishkovets, and M. Jarke. «Eve: A sketch-based software prototyping workbench». In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–6 (cit. on p. 6).

[27] J.M. Rivero, G. Rossi, J. Grigera, J. Burella, E.R. Luna, and S. Gordillo. «From mock-ups to user interface models: An extensible model driven approach». In: *International Conference on Web Engineering*. Berlin, Heidelberg: Springer, 2010, pp. 13–24 (cit. on p. 7).

[28] Y. Liu, Q. Hu, and K. Shu. «Improving pix2code based bi-directional LSTM». In: *2018 IEEE International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*. IEEE. 2018, pp. 220–223 (cit. on pp. 7, 10).

[29] Ó.S. Ramón, J.S. Cuadrado, J.G. Molina, and J. Vanderdonckt. «A layout inference algorithm for graphical user interfaces». In: *Inf. Softw. Technol.* 70 (2016), pp. 155–175 (cit. on p. 7).

[30] N. Sinha and R. Karim. «Compiling mock-ups to flexible UIs». In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 312–322 (cit. on p. 7).

[31] B. Mihalcea. *User Interface Construction with Mock-Up Images*. U.S. Patent 8, 650, 503. 2014 (cit. on p. 7).

[32] V.P.S. Pandian, S. Suleri, and M. Jarke. «Blu: What GUIs are made of». In: *Proceedings of the 25th International Conference on Intelligent User Interfaces Companion*. USA, 2020, pp. 81–82. URL: https://blu.blackbox-toolkit.com/ (cit. on p. 7).

[33] S. Ren, K. He, R. Girshick, and J. Sun. «Faster R-CNN: Towards real-time object detection with region proposal networks». In: *Advances in Neural Information Processing Systems*. 2015, p. 28 (cit. on pp. 7, 10).

[34] R.J. Williams and D. Zipser. «A learning algorithm for continually running fully recurrent neural networks». In: *Neural Comput.* 1.2 (1989), pp. 270–280 (cit. on p. 7).

[35] Z. Zou, Z. Shi, Y. Guo, and J. Ye. «Object detection in 20 years: A survey». In: (2019). arXiv preprint arXiv:1905.05055 (cit. on p. 7).

[36] T. Kohonen. «Learning vector quantization». In: *Self-Organizing Maps*. Berlin, Heidelberg: Springer, 1995, pp. 175–189 (cit. on p. 7).

[37] K.P. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. «Machine learning-based prototyping of graphical user interfaces for mobile apps». In: *IEEE Trans. Softw. Eng.* (2018) (cit. on p. 6).

[38] Z.Q. Zhao, P. Zheng, S.T. Xu, and X. Wu. «Object detection with deep learning: A review». In: *IEEE Trans. Neural Netw. Learn. Syst.* 30.11 (2019), pp. 3212–3232 (cit. on p. 7).

[39] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu. «A survey of deep learning-based object detection». In: *IEEE Access* 7 (2019), pp. 128837–128868 (cit. on p. 7).

[40] *Worldwide Usage of JavaScript Front-end Libraries*. the annual developer survey of the JavaScript ecosystem. 2020. URL: https://2020.stateofjs.com/en-US/ (cit. on p. 7).

[41] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. «RICO: A mobile app dataset for building data-driven design applications». In: *Proceedings of the 30th Annual ACM Symposium on UIST (UIST '17)*. New York, NY, USA: ACM, 2017, pp. 845–854. DOI: http://dx.doi.org/10.1145/3126594.3126651 (cit. on pp. 7, 11).

[42] D. Gibson, K. Punera, and A. Tomkins. «The volume and evolution of web page templates». In: *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*. ACM, 2005, pp. 830–839 (cit. on p. 7).

[43] Huang Ruozi, Yonghao Long, and Xiangping Chen. «Automatically generating web page from a mock-up». In: *International Conference on Software Engineering & Knowledge Engineering*. USA, 2016, pp. 589–594 (cit. on pp. 6, 9).

[44] Nguyen Tuan Anh and Christoph Csallner. «Reverse engineering mobile application user interfaces with REMAUI». In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2015, pp. 248–259 (cit. on pp. 6, 9).

[45] S. Natarajan and C. Csallner. «P2A: A tool for converting pixels to animated mobile application user interfaces». In: *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2018, pp. 224–235 (cit. on pp. 6, 9).

[46] Z. Zhu, Z. Xue, and Z. Yuan. «Automatic graphics program generation using attention-based hierarchical decoder». In: *Asian Conference on Computer Vision*. Springer, Cham, 2018, pp. 181–196 (cit. on pp. 6, 10).

[47] T.F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar. «Learning design semantics for mobile apps». In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 2018, pp. 569–579 (cit. on p. 6).

[48] Kerry Shih-Ping Chang and Brad A. Myers. «WebCrystal: Understanding and reusing examples in web authoring». In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2012, pp. 3205–3214 (cit. on p. 7).

[49] Y. Hashimoto and T. Igarashi. «Retrieving web page layouts using sketches to support example-based web design». In: *SBM*. 2015, pp. 155–164 (cit. on p. 7).

[50] F. Huang, J.F. Canny, and J. Nichols. «Swire: Sketch-based user interface retrieval». In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. USA: ACM, 2019, pp. 1–10 (cit. on p. 7).

[51] X. Ge. «Android GUI search using hand-drawn sketches». In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 141–143 (cit. on p. 7).

[52] F. Behrang, S.P. Reiss, and A. Orso. «GUIFetch: Supporting app design and development through GUI search». In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018, pp. 236–246 (cit. on p. 7).

[53] *The Grid*. Available at: `https://the-grid.org/`. 2023 (cit. on p. 8).

[54] *Bookmark*. Available at: `https://www.bookmark.com`. 2023 (cit. on p. 8).

[55] *Firedrop*. Available at: https://firedrop.ai. 2023 (cit. on p. 8).

[56] *Wix ADI*. Available at: https://www.wix.com/. 2023 (cit. on p. 8).

[57] *Leia*. Available at: https://heyleia.com. 2023 (cit. on p. 8).

[58] *Zyro*. Available at: https://zyro.com. 2023 (cit. on p. 8).

[59] T.K. Ho. «Random decision forests». In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. IEEE, 1995, pp. 278–282 (cit. on p. 9).

[60] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. «WebUI: A Dataset for Enhancing Visual UI Understanding with Web Semantics». In: CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. URL: https://doi.org/10.1145/3544548.3581158 (cit. on p. 11).

[61] Vinoth Pandian Sermuga Pandian, Sarah Suleri, and Matthias Jarke. «SynZ: Enhanced Synthetic Dataset for Training UI Element Detectors». In: *26th International Conference on Intelligent User Interfaces - Companion*. IUI '21 Companion. College Station, TX, USA: Association for Computing Machinery, 2021, pp. 67–69. ISBN: 9781450380188. DOI: 10.1145/3397482.3450725. URL: https://doi.org/10.1145/3397482.3450725 (cit. on p. 11).

[62] *Selenium*. Available at: https://www.selenium.dev/. 2023 (cit. on pp. 16, 20, 51).

[63] *sanitize-html*. Available at: https://github.com/apostrophecms/sanitize-html. 2023 (cit. on p. 16).

[64] *htmlparser2*. Available at: https://github.com/fb55/htmlparser2. 2023 (cit. on p. 16).

[65] *tinycss2*. Available at:https://github.com/Kozea/tinycss/. 2023 (cit. on p. 18).

[66] *bootstrap*. Available at:https://getbootstrap.com/. 2023 (cit. on pp. 18, 51).

[67] *tailwind CSS*. Available at:https://tailwindcss.com/. 2023 (cit. on p. 18).

[68] *bulma*. Available at:https://bulma.io/. 2023 (cit. on p. 18).

[69] *react*. Available at:https://it.reactjs.org/. 2023 (cit. on p. 25).

[70] *gatsby*. Available at:https://www.gatsbyjs.com/. 2023 (cit. on p. 25).

[71] *Next*. Available at:https://nextjs.org/. 2023 (cit. on p. 25).

[72] *Nuxt*. Available at:https://nuxtjs.org/. 2023 (cit. on p. 25).

[73] *backbone*. Available at:https://backbonejs.org/. 2023 (cit. on p. 25).

[74] *Vue*. Available at:https://vuejs.org/. 2023 (cit. on p. 25).

[75] *Angular*. Available at:https://angularjs.org/. 2023 (cit. on p. 25).

[76] *Ember*. Available at: https://emberjs.com/. 2023 (cit. on p. 25).

[77] *Majestic Million*. Available at: https://majestic.com/reports/majestic-million. 2023 (cit. on pp. 26, 30, 31).

[78] *Computing@PoliTO*. Available at: https://computing.polito.it/servizi-per-la-ricerca. 2023 (cit. on p. 32).

[79] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL] (cit. on p. 38).

[80] *Bard*. Available at: https://ai.google/static/documents/google-about-bard.pdf. 2023 (cit. on p. 38).

[81] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, Zhen Ming, and Jiang. *GitHub Copilot AI pair programmer: Asset or Liability?* 2023. arXiv: 2206.15331 [cs.SE] (cit. on p. 38).

[82] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG] (cit. on p. 38).

[83] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG] (cit. on pp. 38, 41).

[84] Kenton Lee et al. *Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding*. 2023. arXiv: 2210.03347 [cs.CL] (cit. on p. 39).

[85] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html (cit. on p. 39).

[86] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV] (cit. on p. 40).

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL] (cit. on p. 40).

[88] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL] (cit. on pp. 40, 41).

[89] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL] (cit. on p. 42).

[90] *Colab*. Available at: https://colab.research.google.com/. 2023 (cit. on pp. 43, 66).

[91] Kishore Papineni, Salim Roukos, Todd Ward, and Wei Jing Zhu. «BLEU: a Method for Automatic Evaluation of Machine Translation». In: (Oct. 2002). DOI: 10.3115/1073083.1073135 (cit. on p. 45).

[92] Edward Loper Bird Steven and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc, 2009 (cit. on p. 46).

[93] Benjamin Paaßen. *Revisiting the tree edit distance and its backtracing: A tutorial*. 2022. arXiv: 1805.06869 [cs.DS] (cit. on p. 46).

[94] *ZhangShashaPython*. Available at: https://github.com/timtadh/zhang-shasha. 2023 (cit. on p. 47).

[95] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. «Image quality assessment: from error visibility to structural similarity». In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861 (cit. on p. 47).

[96] *Scikit-image*. Available at: https://scikit-image.org/. 2023 (cit. on p. 47).

[97] *WebGenerator*. Available at: https://github.com/agsoto/webgenerator. 2023 (cit. on p. 51).

[98] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG] (cit. on p. 56).

[99] *Html-Tidy*. Available at: http://www.html-tidy.org/. 2023 (cit. on p. 63).

[100] Noam Shazeer and Mitchell Stern. *Adafactor: Adaptive Learning Rates with Sublinear Memory Cost*. 2018. arXiv: 1804.04235 [cs.LG] (cit. on p. 66).

[101] M. Knoll T. Angerer. *pix2code pytroch implementation*. 2021. URL: https://github.com/timoangerer/pix2code-pytorch/tree/master (cit. on p. 68).

[102] Yun Luo, Zhen Yang, Fandong Meng, Yafu Li, Jie Zhou, and Yue Zhang. *An Empirical Study of Catastrophic Forgetting in Large Language Models During Continual Fine-tuning*. 2023. arXiv: 2308.08747 [cs.CL] (cit. on p. 70).

[103] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. *CTRL: A Conditional Transformer Language Model for Controllable Generation*. 2019. arXiv: 1909.05858 [cs.CL] (cit. on p. 76).

[104] *Weight and Biases*. Available at: https://wandb.ai/site. 2023 (cit. on p. 83).