Surveying the .NZ Top Level Domain: Business Sector Categorisation

Tomas Borsje

***Abstract*—There is a vast amount of information present across ".NZ" domains, but no publicly accessible or viable tools or resources exist to categorise them. All currently existing solutions are either heavily rate-limited or require expensive monthly subscriptions, leaving no affordable option to categorise our dataset of 210,000 ".NZ" domains.**

Combining this new business category dimension of data with other datasets such as the Transport Layer Security (TLS) information or cookie usage of ".NZ" domains would provide us greater insight into the landscape of New Zealand's digital presence. With such a rich dataset, we can explore aspects like which of New Zealand business sectors are the most or least secure, which sectors have the greatest online presence, and more - providing a solid foundation for further research into how New Zealand's online presence impacts our economy, cyber-security, and more.

This project surveyed publicly available options for domain business categorisation, and developed a system capable of rapidly extracting website information and classifying it into one of 27 business categories. The trained classifier uses the 'transformers' Python library and categorises a test dataset with 70% accuracy, using only a website's title, description, and keyword meta-tags.

A command-line interface was also developed using the 'click' Python library to allow for information extraction and classification via a scripting interface, enabling automation and integration with other systems. Thirdly, an SQLite database was designed and populated with both the list of domains and website data extracted from these domains.

***Index Terms*—Business sectors, classification, web scraping, multi-processing, natural language processing.**

I. Introduction

There exists a wealth of information across to be gathered across the ∼750,000 .NZ top-level domains (TLD) that exist on the world-wide-web [1]. To know the business category of each of these domains is invaluable for the purposes of digital marketing, cyber-security research, and more. With insight into the proportion of these categories across .NZ websites, we can develop a solid foundation for the comparison of websites in different business sectors through the cross-referencing of auxiliary datasets such as website traffic, website security, cookie usage, and more. The ability to compare websites in this manner will provide us with a lens to understand how businesses in New Zealand handle their online presence in an increasingly online world. As this project is done in tandem with other projects aimed at extracting other facets of information from the same dataset (website TLS details, cookie usage), this presents an exciting opportunity to collaborate and enrich our insights through the combination of our datasets. Through the use of these supportive datasets, we would be able to investigate industry-sector specific information, allowing us to investigate questions such as which business sectors have the least secure websites, or which business sectors use the most tracking cookies. Such information would indeed be valuable for purposes such as cyber-security research, as the ability to identify widespread problems with New Zealand's digital presence would allow for the implementation of preventative measures and raising of public awareness, improving our country's online security.

Machine learning and natural language processing (NLP) approaches have proven to be an increasingly powerful way to classify text, with website content and metadata being no exception. Through feature design based on web page content and machine learning models tailored to our classification task, machine learning can be an effective way to classify websites into business categories with high accuracy [2]. Through this, we see that machine learning may present an avenue for implementing our own domain categorisation system.

Although the environmental impact of this project is negligible, the sustainability goals set out by the United Nations [3] were kept in mind during development. The processing time and computational power required of this project could prove to be non-negligible if efficient design was not kept in mind, so as to keep our environmental impact to a minimum we aimed to maximise efficiency and reduce the project's processing time so that our project does not violate the sustainable consumption goal through excess electricity consumption and strain on computer hardware.

A. Motivation

Despite the clear benefits of having the ability to survey a country's digital landscape, no free or non-commercial solutions exist that provide this functionality. Existing commercial solutions such as SimilarWeb [4] or Inovvo's Web Categorisation API [5] require expensive monthly subscriptions and enforce strict rate-limits, making such solutions unfeasible for the widespread surveying required to survey our known list of 210,000 ".NZ" domains. Crowd sourcing websites such as Amazon Mechanical Turk [6] allow for human classification of websites, but these solutions are also costly and slow, rendering them unfit for our purpose. Finally, manual classification is not feasible due to the sheer volume of websites that would need to be investigated. Additionally, malicious websites pose a threat as connecting to a website containing malware puts the user's security and personal device at risk.

Therefore, the motivation for this project is to develop and implement our own solution capable of scraping bulk

This project was supervised by Dr. Arman Khouzani.

website data, classifying website data, and interfacing with the command-line to allow for scripting-based automation. With a system developed that is capable of these tasks, we are able to build the foundational dataset for use in further industry-sector related cyber-security research. When our produced dataset would be combined with those of the projects being executed in tandem, our insight into the state of New Zealand's web presence would be exceptional - setting out the motivation for our project. Additionally, the system would be able to be executed on datasets of not just ".NZ" domains, but also those of any country, allowing for the surveying of any country's websites (and possibly the entire globe's).

B. Final Product

Our final implementation consists of three subsystems that interact to provide, in full, the functionality set out previously by the project requirements outlined in the preliminary report. These subsystems are the following:

1) Website Data Extraction Subsystem: We developed a Python-based system capable of bulk-processing batches of website URLs from a local database. This subsystem utilises multi-processing to process a configurable number of websites at a time, sending a single web-request to each website to query its content that is then sanitised, extracted, and stored back in the local database.

During project execution, this system was able to extract website data for a total of 1.1 million .NZ and .AU domains in 1.5 days, resulting in a total processing speed of $\sim$8.5 websites per second - surpassing the previously set project requirement of 5 websites per second.

2) Website Data Classifier: We developed a Python-based system capable of classifying previously extracted website information into a business category. After investigation of multiple other classification models (Naive Bayes, XGBoost, SVM), we chose the HuggingFace *transformers* library's DistilBERT model [7] as it surpassed the performance of the previously tested models, creating a wrapper class that allows for the other subsystems to call on the classification model.

3) Command-Line Coordinator: We developed a command-line interface for the project using the *Click* [8] Python library. With this implementation, the system can be controlled via the command-line or other scripting interfaces through a selection of command-line flags, invoking the other subsystems. Using the developed flags, this subsystem is capable of the following:

- Creating the local database and auxiliary configuration files, along with populating the database using a provided input file of URLs.
- Using the Google Safe Browsing API [9] to detect any malicious websites in the database.
- Calling on the other subsystems to scrape and classify websites in the database, storing this information.
- Interfacing with the command-line, allowing users to classify websites through scripting.

Through this command-line interface, we met the project requirements of providing support for scripting-based automation.

II. Background

This section aims to compare the website classification solutions implemented by other groups, illustrating the unfeasibility of using such approaches. Additionally, a variety of artificial intelligence-based approaches for website classification are compared and evaluated. Finally, all the tools and methodologies used in our system are described in detail.

A. Related Works

1) Similarweb: Similarweb [4] is a commercial website categorisation and digital research platform. They provide an API capable of classifying website URLs into business categories in bulk, but this requires a monthly subscription. Additionally, the API only serves at a maximum rate of 10 requests / second with their 'Team' plan - the lowest plan that provides API access. The pricing for their 'Team' package is not public as a demo must be requested, but the next lowest price plan is $333/user/month. Their methodology for website classification is also not public, rendering the use of Similarweb an untenable approach to our problem.

2) Inovvo: Industry-Leading Website Categorization API [5]: Innovo, LLC provide a website categorisation API through Amazon's AWS Marketplace [5]. Their API provides support for classification of URLs into Interactive Advertising Bureau (IAB) categories, of which there are over 200. However, much like Similarweb's API, Inovvo's API is also not free, costing $750/month along with a flat fee of $0.00015 per website categorised - rendering Inovvo's API also unfeasible for use in our system.

3) Amazon Mechanical Turk: Amazon Mechanical Turk is a crowdsourcing platform that allows users to enlist workers to complete Human Intelligence Tasks (HITs) for a configurable price and fee [6]. Through this website, one can enlist workers to manually classify websites, ensuring a high standard of accuracy. However, Amazon Mechanical Turk is very expensive, with each separate website costing a minimum of $0.02 to list - being a reward of $0.01 to the worker and a minimum fee of $0.01 to Amazon. With a total of 210,000 ".NZ" websites, this would cost over $4000! Additionally, enlisting workers to complete classification tasks for an incredibly small reward is exploitative and unethical, rendering the use of Amazon Mechanical Turk unacceptable.

Hence, the use of commercial solutions was not feasible for this project, so we researched into the feasibility of developing our own artificial intelligence classification model.

B. Literature Review

In this section, three closely related works are discussed and compared with our approach of

1) Classifying websites by industry sector: a study in feature design [2]: Berardi et al. describe a methodology for categorising website data into business sector categories through a tree of one-vs-all (OVA) Support Vector Machine (SVM) learners for hierarchical classification. They use the Interactive Advertising Bureau (IAB)'s three-level content taxonomy standard as category labels, allowing for a 2-depth labelling system of 216 total labels.

Through the use of a one-vs-all tree of Support Vector Machines, each SVM is trained to differentiate between instances of its assigned category label versus the rest. At each depth of the tree, the branch with the highest confidence classification is selected. This process is repeated until a leaf node is selected and the instance is classified.

The study engineers input features through website content including the number of outgoing URLs on a page, the number of paragraphs on a page, the number of headings on a page, and other page content attributes. Raw website data such as the website's title, URL, meta description and meta keyword tags are also used as input features to the classifiers.

Through the use of this technique, Berardi et al. achieve a true-class accuracy upwards of 80%. Despite this high level of accuracy, this approach was not viable for our system as it required a dataset that contained a high enough count of instances per deepest-level label, which we did not possess. Additionally, hierarchical classification was outside of the scope of this project.

2) Content-based and link-based methods for categorical web-page classification [10]: Choudhury et al. evaluate a selection of methods for web-page categorisation by web-page content including SVM, Multinomial Naive Bayes, decision trees, and Word2Vec embeddings. The study also investigates methods to augment content-based website classification, including the traversal of hyperlinks to gather related web-page data and the construction of a graph for use in graphical classification methods.

This study preprocesses website data by tokenizing the raw HTML file of the web-page and removing any tokens with non-alphanumeric characters. Afterwards, the approach extracts a vector of token lemma counts, lemmatising tokens like *sit, sat, sitting* into the singular headword *sit*.

Choudhury et al. report a Support Vector Machine model having the highest accuracy in comparison to the other evaluated models at an accuracy of 90.26%, although this was only across 4 categories. In comparison, the Multinomial Naive Bayes model was reported as having an accuracy of 84.07%, the decision tree model as having an accuracy of 83.19%, and the Word2Vec model as having an accuracy of 84.6%. This implementation is inferior to Giacomo et al.'s implementation as the accuracies presented are lower despite only 4 classes existing. However, Choudhury et al. additionally investigated a novel method that additionally scraped web pages connected via hyperlink. This method experienced a reduction in overall accuracy but corrected misclassifications present in the singular page only approach.

3) An evaluation of machine learning methods for domain name classification: [11] Garg et al. present numerous natural language processing (NLP) methods and their classification accuracy on domain names through the use of fine-tuning pre-trained models. Accuracies across multiple pre-trained models including BERT, DistilBERT, and roBERTa are all reported to be above 95% over 13 categories. Differing combinations of input features (url, title, description) are also evaluated.

We see that with only website URLs, titles, and descriptions, Garg et al. achieve an exceptional accuracy even with light-weight models like DistilBERT. Although the implementations presented in this paper are more computationally expensive than Giacomo et al. and Choudhury et al.'s approaches due to the architectures of specialized natural language processing models, Garg et al. achieve exceptional performance without lemmatisation, hyper-link scraping, or use of the entire HTML web page. Additionally, the pre-trained models used in this study are available in the popular Python machine learning library HuggingFace [12], whereas the implementations in the previous articles are closed-source requiring us to re-implement them ourselves.

C. Tools and Methodology

This section outlines and describes the tools, libraries, and methodologies used in our system.

1) Website Scraping - requests: requests is a core Python library that provides the ability to send web-requests to URLs, allowing for configurable options like time-outs and custom request headers [13]. As our system needed a way to contact website URLs to extract their website homepage information, *requests* was chosen to fulfill this requirement for a number of reasons. Firstly, *requests* is very light-weight library. Sending only a single web-request to a website and getting a response is very fast. Additionally, as there is no browser interface for a web-request, no rendering is done client-side. This greatly speeds up processing as rendering is slow, so avoiding it allows for much faster processing speeds.

The *requests* library was therefore our choice for this system as it provided the required functionality at a high level of efficiency and performance.

Selenium [14], a browser-automation based web scraping solution, was also evaluated. Selenium is a software driver that allows users to control a browser, visiting websites and extracting the fully-rendered web-pages displayed in the browser. A Python *selenium* library also exists [15], allowing control of the Selenium web driver through Python code. Although Selenium allows for proper rendering of web pages unlike *requests*, the overhead of loading and rendering a website is high. We investigated a Selenium-based website scraping approach and found it performed far below the performance threshold required of the system, with little gain compared to the web-request and raw HTML based approach of *requests*. Additionally, parallelization is much harsher on the system as multiple web-browser processes must be created. Overall, the benefits of the *requests* library were clear, as outlined in Table I.

Table I
WEBSITE SCRAPING IMPLEMENTATION COMPARISON

Name	Method	Speed	Overhead	Native
Selenium	Browser automation	Slow	High	No
requests	Web requests	Very Fast	None	Yes

2) Database - SQLite: As our system aimed to classify a very large dataset of websites, we required a way to store and process this data efficiently. SQLite is a Python-supported Database Management System (DBMS), with a core Python module sqlite3 developed to allow easy interfacing with in-memory or file-based databases. In contrast to traditional

Table II
DATABASE IMPLEMENTATION COMPARISON

Name	Paradigm	Speed	Overhead	Native Support
MySQL	Server-based	Very Fast	High	No
SQLite	File-based	Fast	None	*sqlite3*

database systems that require a live server running, SQLite allowed us to use local file-based databases. These databases do not need to be running 24/7 and can instead be read and connected to through the aforementioned sqlite3 module as required. In a sustainability regard, this reduced our carbon footprint and electricity usage as we did not require a dedicated database server to be online at all times, instead being able to read from local files with SQLite's SQL-based interfaces directly.

We used SQLite to store three different tables in our database:

1) *urls*: This table contains all the URLs imported through the `--csv` flag of the command-line interface, and whether or not these websites are malicious.
2) *website_features*: This table contains all the website data ('features') extracted by the website data extraction subsystem. The url, title, description, keywords, and error type (if an error was encountered during scraping) for each website.
3) *website_category*: This table contains only website URLs and their accompanying category, populated by the classification model invoked with the `--classifyall` flag.

With the sqlite3 module included with Python, we were able to have the system create and initialise this database when the `--setup` flag was invoked. Using a chain of SQL commands, the tables are created and the *urls* table populated - removing the need for the user to set up and integrate their own database with the system. As SQLite is still an SQL-based implementation, users can write their own scripts to invoke SQL statements on the system's database, providing extra functionality to the user and providing yet another avenue for further automation of the system.

We also investigated MySQL, an open-source relational database management system (RDBMS) [16]. MySQL is a database service, meaning that a database server must be running and accessible from any machine that wishes to interface with the database. This is a considerable sustainability overhead as an additional server must be running at all times to provide database access, consuming electricity and increasing the carbon footprint of the system. As SQLite does not require a database server, we chose to use SQLite over MySQL as the overhead of running an additional database was unnecessary. The benefits of SQLite are shown in Table II. Additionally, the differences in paradigm between SQLite and MySQL are detailed in Fig. 1.

3) AI Model - transformers: *transformers* is a Python library developed by HuggingFace that provides pre-trained models for numerous natural language processing tasks, including text classification and sentiment classification [17]. *transformers* provides Python interfaces for the BERT-based

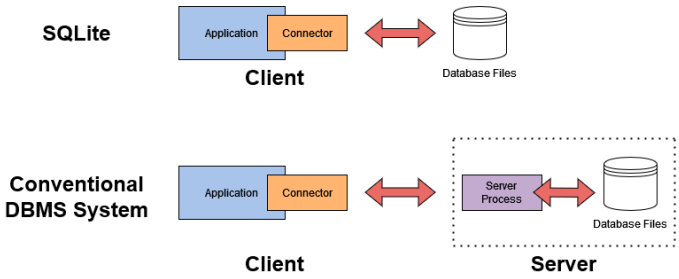

Figure 1. Paradigm comparison between SQLite and MySQL.

Table III
CLASSIFICATION MODEL IMPLEMENTATION COMPARISON

Library	Model	Accuracy	Initialisation	Classification
Scikit-learn	Naive Bayes	42.15%	Slow	Fast
Scikit-learn	SVM	44.45%	Slow	Fast
Scikit-learn	XGBoost	46.40%	Slow	Fast
transformers	DistilBERT	69.70%	Fast	Slow

models evaluated in [11], which allowed us to utilise one of these models in our system. We used the DistilBERT model, a lightweight natural language processing model from the BERT family of models. By fine-tuning the DistilBERT model using labeled website data collected from Kaggle [19], an online machine learning community, we were able to train DistilBERT on website titles, descriptions, and metadata to output business categories. However, fine-tuning DistilBERT with consumer hardware took upwards of three days, so to avoid this problem we used Google Colab's free TPU machines accessible online [20]. With this hardware specialised for machine learning, we were able to fine-tune our DistilBERT model in under 12 minutes, resulting in the final classification model used in our system. With this implementation, we were able to store the model's weights and biases as a ".pth" file, allowing us to quickly load the model into memory at a later time.

scikit-learn, a Python machine learning library, was also investigated. Scikit-learn is a library that provides implementations for a variety of classical AI models, such as Naive Bayes, support vector machine (SVM), and XGBoost classifiers [18]. As the models provided by scikit-learn are less specialized towards text processing, their performance and test accuracy was lower than that of the DistilBERT model. However, these models are also much more lightweight than the bulkier NLP models so their classification speed is faster, but due to the nature of the architectures, they are not able to take advantage of TPUs. Ultimately, considering the benefits shown in Table III, we chose to use the DistilBERT classification model.

As the lower classification speed of DistilBERT was not a major restriction due to this project being intended to run only once rather than live, we chose the DistilBERT model as our website classification model. This provided the highest accuracy overall, along with a faster initialisation speed - meaning our `--classify` flag that classified only a single website had less delay than if we were to use a classical model. The confusion matrix of the DistilBERT model also displayed a significant decrease in confusion compared to the scikit-learn models, shown in Fig. 1 and Fig. 2.

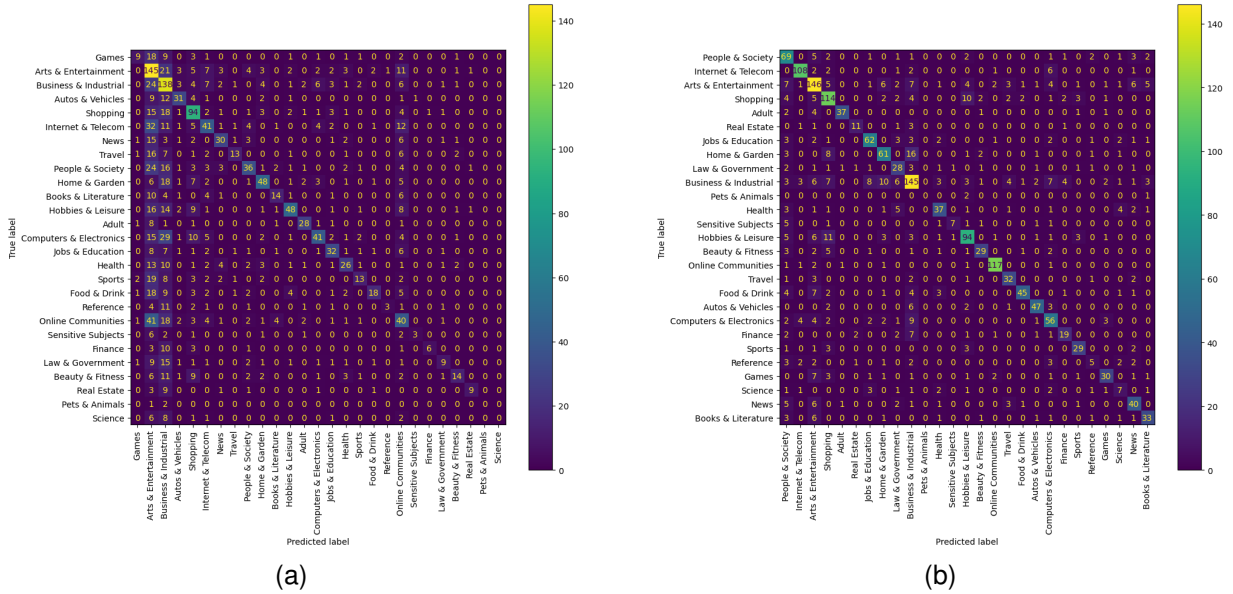

Figure 2. Confusion matrices of the (a) SVM and (b) DistilBERT classification models on the test data.

III. Design and Implementation

This section outlines both the conceptual design of the system and its final implementation.

A. Conceptual Design

The system's design consists of four modules that interact with each other through specific function calls and interfaces, the structure of which is shown in Fig. 3.

1) Database: The first module is the SQLite database and wrapper class. This is a .db file generated through the command-line interface using the `--setup` flag, storing the list of URLs, scraped website data, and website categories. The database is interfaced with through the *sqlite3* library included with Python, allowing for reading, writing, updating, and deletion of rows in the database's various tables through SQL commands tailored to the needs of the other modules. Our other modules do not execute their own SQL commands; all functionality is contained within a specialised database connection class that forms the interface for other modules to integrate with the database and the data stored within. This was a design choice to reduce the coupling between the modules and increase cohesion, as only the database module is responsible for data retrieval and storage. This module is used by the command-line module to read website URLs, store scraped website data, and to store website categories.

2) Website Data Scraper: The second module is the website data scraper. This module is responsible for extracting website data for a provided list of URLs quickly and efficiently using a configurable number of sub-processes to achieve parallelization. This module interacts with websites through the external interface of web requests as a means to retrieve website data, namely through the Python *requests* library. Additionally, as this module has to manage multiple sub-processes and progress information, its only interface is the ability to receive a list of URLs and return the website data for each URL.

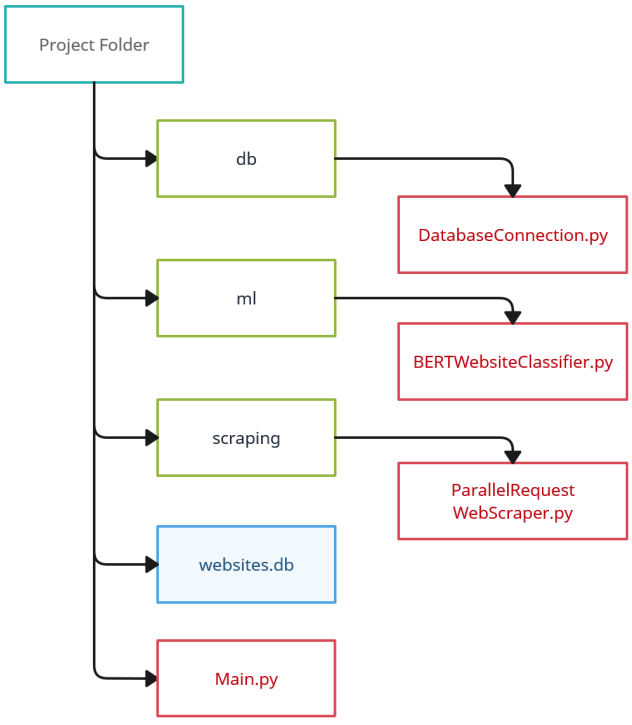

Figure 3. Project directory structure showing the SQLite database and the four modules.

This improves cohesion with the clearly defined interface, and reduces coupling between the modules as this is the only module responsible for extracting website data and interfacing with websites. Additionally, having this as a separate module with a defined interface allowed us to swap out this module for a Selenium-based scraping module during testing, without having to change the code of other modules. The command-

line interface module utilises this module's interface to supply URLs and extract data for the database's stored websites.

3) AI Classification Model: The third module is the website data classification model. This module is responsible for receiving website data previously extracted by the website data scraper module and classifying it into a business category; that is, returning the website data's business category as text to the module calling this interface. By having the AI classification module as a separate module with a clearly defined interface, we were able to test multiple different classification models by swapping them out through their interface in the command line module - a benefit of the low coupling and high cohesion this module split provides. This module's interface is used by the command-line module to dispatch scraped website data and receive corresponding categories in return.

4) Command-Line Coordinator: The fourth and final module is the command-line coordinator. This module is in charge of both receiving commands from users through the command line and coordinating other modules to achieve these requests. This module also interfaces with the Google Safe Browsing API through the *pysafebrowsing* library. This module receives input through the command-line interface through a set of command-line flags:

- `--setup`: Creates the local database file and configuration files used by the other flags.
- `--csv <path>`: Invokes the Database module to load the provided file path into the database as URLs.
- `--scan`: Invokes the Database module to retrieve batches of URLs, invokes the Google Safe Browsing API to detect malicious sites, and then invokes the Database module to mark any malicious sites in the database. Malicious sites are excluded from use by the other modules.
- `--scrapeall`: Invokes the Database module to retrieve batches of URLs, invokes the Website Data Scraper module to scrape website data for the batches of URLs, and then invokes the Database module to store website data in the database.
- `-classifyall`: Invokes the Database module to retrieve batches of scraped website data, invokes the AI Classification Model module to classify the website data of each batch, and then invokes the Database module to store website categories in the database.
- `-classify <url>`: Invokes the Website Data Scraper module to scrape data for the provided URL, then invokes the AI Classification Model module to classify the scraped data. Afterwards, the category is printed to the standard output.

B. Implementation

1) Database Connection: The database connection used to read and write data by the other modules is implemented as a single Python class, *DatabaseConnection.py* in the *db* folder. This class has a number of functions used by other modules. The implementation is as follows:

- *Initialisation*: The class can be instantiated with an optional database connection string, being the path to the website .db file. Upon initialisation, the *sqlite3* library is used to create a connection to the provided websites file, using 'websites.db' in the current folder if not provided, stored in the 'self.connection' field. The connection's cursor is also stored in the 'self.cursor' field. A number of SQL statements are also stored as local fields in the class, used by the other functions to read and write data.
- *storeWebsiteFeatures(self, features: WebsiteFeatures)*: This function receives a *WebsiteFeatures* instance (a wrapper object that stores only scraped website data for a website). The fields of the website data are sanitised to remove any invalid data, being encoded to UTF-8 and having any apostrophes escaped for use in SQL statements. Finally, an SQL statement that stores the website's information in the *website_features* table is called, replacing any existing website data if applicable.
- *uploadCSV(self, path: str)*: This function receives a path to a CSV file, calling on the *pandas* library to read it into a DataFrame which is then stored in the database using the DataFrame's *to_sql(table_name, connection)* function, storing the list of website URLs in the *urls* table of the database.
- *storeWebsiteCategory(self, url: str, category: str)*: This is a simple function, receiving a URL and a category to associate with it. The function executes an SQL statement that adds or replaces any data in the *website_category* table for the given URL, storing the URL and its associated category. This function is used by the command-line coordinator to store categorised websites during the `--classifyall` command.
- *storeWebsiteMaliciousness(self, url: str, malicious: bool)*: This is a simple function, receiving a URL and a boolean indicating if the website is malicious. The function executes an SQL statement that adds or replaces any data in the *urls* table for the given URL, storing the URL and its malicious status. This function is used by the command-line coordinator to store malicious websites during the `--scan` command.
- *getWebsiteBatch(self, size: int, skip: int)*: This function receives a batch size and skip count. The function calls an SQL statement on the database cursor that returns a list of URLs of the provided size from the *urls* table, skipping the first *skip* websites (sorted alphabetically). Through this function, the command line coordinator is able to paginate through the database's URLs, processing them in batches to reduce the chance of an error undoing vast progress.
- *getFeatureBatch(self, size: int, skip: int)*: This function works the same as the getWebsiteBatch() function, instead getting data from the *website_features* table and creating a new list of *WebsiteFeatures* instances to store all the returned features. The size parameter determines how many website features are returned, and the skip parameter determines how many websites are skipped (sorted alphabetically by URL) to allow for pagination by the command-line coordinator module.
- *getNumberOfWebsites(self)*: This is a simple function, calling an SQL statement to retrieve and return the

number of records in the *urls* table. This is used by the command-line coordinator to know how many websites and thus batches must be processed.

2) Website Data Scraper: The website data scraper is implemented as a single Python class, *ParallelRequestWebScraper.py* in the *scraping* folder. The implementation is as follows:

- *Initialisation*: The class can be instantiated with a configurable number of processes and a verbosity flag, storing these parameters. However, no work is done until the *get_features(self, URLs: [str])* function is called (in this case, by the command line coordinator).
- *get_features(self, URLs: [str])*: This function begins the scraping process for the given batch of URLs provided through the URLs parameter. First, two *multiprocessing* queues are created for the URL input and website data output. *multiprocessing* queues must be used so that the data is available across the multiple sub-processes instantiated later. If the verbosity flag is true, a *tqdm* progress bar is initialised and displayed. Then, each provided URL is added to the input queue and a new subprocess running the *runProcess(self, inputQueue: Queue, outputQueue: Queue)* function is created and started, repeated to match the number of processes requested through the class constructor. This function then waits until the input data queue is empty, signifying completion of the sub-processes' website data scraping. If the verbosity flag is set, the progress bar is constantly refreshed to display the current status to the user. Finally, all sub-processes are terminated, the progress bar is closed (if verbose), and the output queue, now populated with website data, is returned as the return value of the function.
- *runProcess(self, inputQueue: Queue, outputQueue: Queue)*: This function is what each sub-process created by the previous function runs, taking in an input queue and an output queue. This function loops infinitely, grabbing a new URL from the input queue. With this URL, a new *WebsiteFeatures* instance is created with the provided URL (*WebsiteFeatures* is a simple wrapper class designed to hold scraped website data and nothing else). Then, the *requests* library is invoked to send a single web request to the URL. If the website responds, the *BeautifulSoup* library is used to extract the title, description, and keyword metadata HTML tags of the website if they exist. These values are then stored in the previously created *WebsiteFeatures* instance. If an error occurs during the web-request or during scraping, parsing of the data is aborted and the error type and message are instead stored on the *WebsiteFeatures* instance. Finally, this website data is placed into the output queue and the process is repeated. If there are no URLs left in the input queue, the sub-process breaks the loop and stops execution.

With this implementation, the website data scraper module is responsible only for the gathering and parsing of website data, and nothing else. Other modules can thus call on this module to extract website data for a list of URLs for them.

3) AI Classification Model: The AI classification module is implemented as a single Python class, *BERTWebsiteClassifier.py* in the *ml* folder. The model is able to classify website data into one of the following categories: Adult, Arts & Entertainment, Autos & Vehicles, Beauty & Fitness, Books & Literature, Business & Industrial, Computers & Electronics, Finance, Food & Drink, Games, Health, Hobbies & Leisure, Home & Garden, Internet & Telecom, Jobs & Education, Law & Government, News, Online Communities, People & Society, Pets & Animals, Real Estate, Reference, Science, Sensitive Subjects, Shopping, Sports, or Travel.

The implementation of the classification model is as follows:

- *Initialisation*: The class can be instantiated with a variety of parameters: `model_path`, `input_data`, `eval_data`, `epochs`, `target_col`, `output_dir`, and `verbose`. The model is then initialised as follows: If input_data, a *pandas* DataFrame, is not provided, the model loads the 'training_data_clean.csv' in the same directory into the DataFrame as replacement. Then, the model builds a mapping of each class label to a corresponding integer id and vice versa. This is used during inference to map the model's numerical output to the corresponding business category text, as the DistilBERT model only outputs ids and not the class labels themselves. Then, if the `model_path` is provided, the model is loaded from the given folder. This parameter is used in the command-line coordinator to load the pre-trained model so that the model does not need to be fine-tuned again. Otherwise, if the `model_path` is not provided, the model is trained using the *transformers* library's Trainer class, training and evaluating the model on the input training and evaluation data over the provided number of epochs. This takes upwards of 3 days on consumer hardware and was thus done on Google Colab's free online TPU machines. Finally, the model is finished and ready for classification through the other classify_website(self, features: WebsiteFeatures) function.
- *classify_website(self, features: WebsiteFeatures)*: This function is responsible for providing a classification interface for the other modules, given a provided instance of *WebsiteFeatures*. This function concatenates the url, title, description, and keywords of the website features and then tokenizes them using the model's tokenizer. This tokenized data is a conversion of textual data into data the model can parse, e.g. "Hello World" becoming "193, 102". The model then predicts on this tokenized input and outputs a set of logits, where each logit is a float value that indicates the model's confidence that the input is of the respective business category.
  The highest value logit's index is the index of the model's predicted id. Using this information, the index of the logit is translated to the final business category predicted by the model using the id to label mapping dictionary created during initialisation, and the class label is returned to the caller of this function. This is the only other function used by the other modules.

4) Command-Line Coordinator: The command-line coordinator is the module that interfaces with the user through the command-line, additionally allowing for automation through scripting. This module is implemented in *Main.py*, in the root directory of the project. This module handles a lot of general functionality, split through multiple command-line flags. If any of the provided flags require database access, a new DatabaseConnection is created using the Database module. The flags, and their implementations, are as follows:

- `--setup`: This command-line flag creates the websites.db file used as the SQLite database, as well as creating the .env config file. First, user is asked to confirm that they wish to create and overwrite any existing files. Then, the .env file is created using a template that contains the number of processes to run at once, and stores the user's Google SafeBrowsing API key. After creating this .env config file, the module uses the *sqlite3* library to create a new websites.db file and connects to it, executing a number of SQL statements on the connection to create the required tables (*urls*, *website_features*, and *website_category*). Finally, the nz_domains.csv and au_domains.csv are loaded as DataFrames and input into the database using the *pandas* to_sql() function, if they are present. At this point, the database is ready to be populated with website data and additional URLs, if desired. The .env config file can also be configured by the user to provide their own API key for SafeBrowsing scanning, or to change the number of processes used during website scraping.
- `--csv` <path>: This command-line flag takes in a path to a .csv file containing URLs split by new lines. This file is read into a DataFrame using the *pandas* library and uploaded to the database using the DatabaseConnection instance.
- `--scan`: This function paginates through the list of URLs in the database using the DatabaseConnection instance, with a batch size of 5000 (the max the Google SafeBrowsing API can receive per request). Then, for each batch, the Google SafeBrowsing API is queried using the user's API key stored in the .env config file. Any malicious websites are marked as such in the database using the DatabaseConnection's storeWebsiteMaliciousness function. Additionally, all detected malicious websites are written to a malicious_websites.csv file created in the current directory.
- `--scrapeall`: This function uses the DatabaseConnection instance to paginate through all the website URLs in the database with a batch size of 10,000. With each batch of URLs retrieved from the database, a new instance of the website data scraper module is instantiated and supplied with the list of URLs. Once the URLs have been processed and the scraped website data is returned, each WebsiteFeatures instance is stored in the database using the DatabaseConnection. These batches are repeated until all the URLs in the *urls* table have been processed, populating the *website_features* table.
- `--classifyall`: This function uses the DatabaseConnection instance to paginate through all the scraped website data in the database with a batch size of 1,000. First, a new instance of the AI classification model is created. Then, the database is paginated through in batches, creating a new *tqdm* progress bar for each batch. Each instance of website features is provided to the website classifier, returning a business category. This returned business category is stored in the database using the DatabaseConnection's storeWebsiteCategory() function. This batch process is repeated until all the websites in the *website_features* table have been processed and classified, populating the *website_category* table.
- `--classify` <url>: This function does not use the database connection, instead creating a single process instance of the website data scraper. This scraper is provided the given url, scraping and returning the website data for it. Then, this website data is fed into a new instance of the website classification model, returning the website's business category as text. Finally, this business category is printed to the standard output. Note that the verbosity of each other module is turned off in this function, so that the only text printed to the standard output is the business category of the URL. This allows for automation-based scripting through the command line, providing an interface for other languages or applications to use this system.

C. Sustainability Considerations

This project does not pertain very strongly to the sustainability rules set out by the United Nations, but the electricity consumption and hardware strain caused by the long run-time of this system are still considerable. To mitigate high electricity consumption and strain on hardware used, the following steps were taken:

- Firstly, the choice was made to use the file-based SQLite implementation as our database instead of the MySQL implementation. This allowed us to bypass the need for a perpetually running database server, reducing electricity consumption and e-waste generation significantly.
- Secondly, we designed our system to be efficient through the use of singular web-requests and multi-processing to reduce the processing time taken to execute the web scraping and website classification of our dataset. Had we used a Selenium-based approach for web-scraping and not employed multi-processing, the run-time of the program would have been greatly extended, consuming more electricity and increasing the environmental impact of the system.
- Thirdly, by using singular web-requests instead of browser automation, we reduce the amount of web traffic the system emits. As loading a web page through a browser queries a variety of scripts, embedded images, etc. utilising only a singular web request in place of browser automation greatly reduced the bandwidth used by the system, ensuring no impact was had on other users of the same internet network.

IV. Evaluation

This section details the performance metrics used to evaluate the solution, the results under these performance metrics, and the limitations of the solution - along with possible improvements through future work.

A. Performance Metrics

The performance metrics used to evaluate this system are the following:

1) Websites Scraped Per Second: As our system will be scraping website data for a vast list of websites, it is imperative that the website scraping is efficient so as to reduce the amount of time the project executes. A higher rate of websites scraped per second results in a reduction of the time needed to run the project, in turn reducing the amount of electricity consumed by the system's machine. Additionally, the faster websites can be re-scraped, the more often we can update our dataset with any changes to the websites. With these points, it is clear that the system must be able to scrape website data at an acceptable rate. As set out by the project requirements, we expect a rate of at least 5 websites per second - allowing a dataset of 210,000 domains to be scraped in approximately half a day.

2) Classification Accuracy: As our system will be classifying scraped website data into business categories, we aim to achieve as high a classification accuracy as possible, defined by the model's performance on a test set of data after model training is complete. However, we are not able to gather the accuracy of the model on the entire list of 210,000 ".NZ" domains as these domains are not labeled. Thus, we measure the accuracy of the model such that we can extrapolate the data to the rest of the unlabelled data.

3) Classification Confusion Matrices: We will also measure our AI model's classification accuracy through the use of confusion matrices. Confusion matrices display the predicted category and ground truth category of data points. This allows us to see how often the model 'confuses' different categories with others. An adequate confusion matrix will have a brightly colour line down the diagonal, indicating that the model is predicting websites to be their true category.

B. Performance Results

1) Websites Scraped Per Second: Over the course of the project, the system was executed a total of three times on datasets of differing sizes using a count of 16 sub-processes. This project was executed using a 3.6GHz processor with 12 CPU cores and 16GB of RAM. The website scraping speeds from these runs are presented in Table IV.

Table IV
Website Scraping Speeds

Dataset	Website Count	Time Taken	Websites/second
.nz	210,000	7 hours	~8.33
.au	900,000	30 hours	~8.42
.nz + .au	1,110,000	35 hours	~8.48

We see that all the project executions resulted in a processing speed of above at least 8 websites per second. This shows the system has exceeded the project requirement of a website scraping speed of at least 5 websites per second, achieving its goal of efficient website scraping. If the Selenium-based approach was to be used, it is likely the system would not have met the requirement. Additionally, if multi-processing was not employed, the website scraping speeds would have been roughly 16x lower, resulting in a scraping speed of approximately 0.5 websites/second, falling far short of the rate set out in the project requirements.

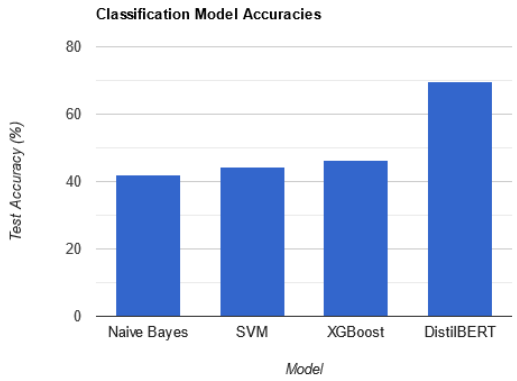

Figure 4. Classification model accuracies on test data.

2) Classification Accuracy: As mentioned in previous sections, our final DistilBERT-based model achieved a classification accuracy of 69.70% on the training data. This implies that the classifier classifies 7/10 websites into their correct categories, which is impressive given only the website title, description, and keywords - combined with the fact that there are 27 categories. The previously investigated classical models did not perform as well, achieving 42.15% accuracy with Naive Bayes, 44.45% accuracy with Support Vector Machines, and 46.40% with an XGBoost classifier, as shown in Table V and Fig. 4.

However, considering that approximately 40% of websites do not have a description tag, there is room for improvement through the use of other website features, as discussed in the next section.

Table V
Classification Model Performance

Name	Test Accuracy
Naive Bayes	42.15%
Support Vector Machine	44.45%
XGBoost	46.40%
DistilBERT	69.70%

3) Classification Confusion Matrices: Presented in Figures 5, 6, and 7 are the confusion matrices of the models investigated.

We see that the Naive Bayes and SVM models have distinct noise around the non-diagonal values, indicating a significant level of 'confusion' between the ground truth labels and predicted website labels. We see a distinct difference in the confused categories between the SVM and XGBoost

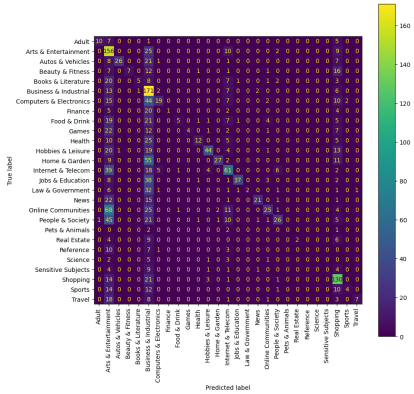

Figure 5. Confusion matrix of the Naive Bayes classification model on test data.

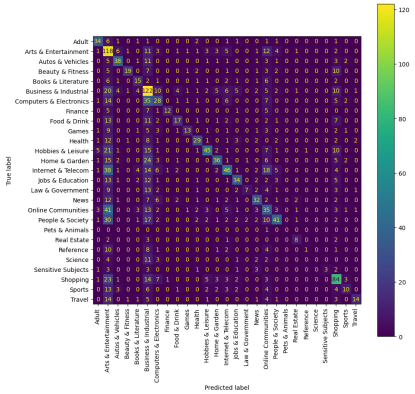

Figure 6. Confusion matrix of the SVM classification model on test data.

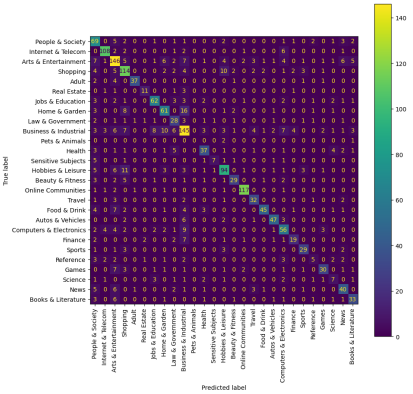

Figure 7. Confusion matrix of the DistilBERT classification model on test data.

classifiers. However, looking at the confusion matrix of the DilstilBERT model, we see a notable improvement. There are no distinct vertical or horizontal lines on the confusion matrix, indicating that the DistilBERT model does not often confuse one category for another. Instead, it performs well on the classification task regardless of the true business category of the website data. It is likely that with further feature engineering and more specialized website data extraction, an even higher level of accuracy could be achieved.

C. Project Results

As outlined in Table IV, this project was executed multiple times. The final run of the project was executed with the ".NZ" and ".AU" datasets combined, resulting in a total of approximately 1,110,000 URLs processed. The results of the website scraping and classification are presented below.

1) Malicious Website Results: As the websites in each dataset were checked for maliciousness using the Google Safe Browsing API before scraping, a number of websites were detected as malicious. In total, 930 websites were detected as malicious, with 716 of these websites being ".AU" websites and 214 being ".NZ" websites. Interestingly, a majority of these websites appear to be legitimate websites, often owned by small businesses that are now marked as malicious and deceptive by Google's Safe Browsing API. This could indicate that smaller websites are more likely to be targets for attack by bad faith actors, but investigating this is outside the scope of the project and could be an interesting avenue for future work.

2) Scraping Results: During website scraping, multiple errors were encountered and handled. The errors and their counts are presented in tables VI and VII, split between ".NZ" and ".AU" websites.

We see that the `ConnectionError` error was the most common error type. This error signifies a website no longer exists and can be used as an indicator of how many websites are no longer online across the datasets. Using this metric, we see that 50,302 of the websites in the ".NZ" dataset no longer exist, a percentage of approximately 24%. Additionally, 235,722 of the websites in the ".AU" dataset were also no longer online, a percentage of approximately 21%. This is understandable as the dataset of URLs used is a few years old as of 2023; this is discussed more in the *Limitations and Possible Improvements* section. Additionally, these somewhat high values suggest ∼20% of websites cease to exist after a few years. This is quite a high turnover rate and could be an interesting point to investigate in future.

Alongside the `ConnectionError` errors are many more less common errors. An `AttributeError` indicates that the website does not have a title, caused by the website scraping program expecting one (however, this error is still handled), being encountered 27,613 across the two datasets. An `SSLError` indicates that the website's SSL certificate configuration is invalid and thus it is not safe to connect to the

Table VI
.NZ DATASET SCRAPING ERRORS

Error Type	Count
AttributeError	3718
ChunkedEncodingError	2
ConnectTimeout	3506
ConnectionError	50302
ContentDecodingError	2
Exception	23
InvalidURL	3
ReadTimeout	1419
SSLError	1429
TooManyRedirects	82

Table VII
.AU DATASET SCRAPING ERRORS

Error Type	Count
AttributeError	23895
ChunkedEncodingError	12
ConnectTimeout	13200
ConnectionError	235722
ContentDecodingError	18
Exception	102
InvalidSchema	3
InvalidURL	15
LocationParseError	1
MemoryError	1
ReadTimeout	9664
SSLError	6453
TooManyRedirects	557

website, being encountered 7,882 times. The `ReadTimeout` and `ConnectTimeout` errors were encountered a combined total of 27,789 times during scraping. These errors indicate a website that is online but taking too long to respond; this could be caused by either heavy server load or an improperly configured web server. The `TooManyRedirects` error occurs when the web server attempts to redirect the connection too many times; this was encountered a total of 639 times during scraping and can often indicate a malicious website. The remaining error types were encountered only very rarely, the majority of which indicate either an invalid response by the web server or invalid formatting of the response.

Additionally, not every scraped website contained title, description, or keyword meta tags. Of the websites scraped, only ∼62% contained a title tag. Every website scraped that was missing a title tag also did not have any other tags. For a classification model that expects at least a title, 38% of websites not having any form of title can result in decreased classification accuracy, as the model has only the URL to use in classification.

3) Classification Results: Presented in Tables VIII and IX are the category counts predicted for all the websites in each dataset.

Table VIII
.NZ DATASET WEBSITE CATEGORIES

Category	Count	Percentage
Adult	113	0.05%
Arts & Entertainment	30,882	14.52%
Autos & Vehicles	2,541	1.19%
Beauty & Fitness	1,446	0.68%
Books & Literature	253	0.12%
Business & Industrial	40,977	19.26%
Computers & Electronics	84,568	39.76%
Finance	704	0.33%
Food & Drink	2,039	0.96%
Games	373	0.18%
Health	3,234	1.52%
Hobbies & Leisure	2,779	1.31%
Home & Garden	8,826	4.15%
Internet & Telecom	9,844	4.63%
Jobs & Education	3,357	1.58%
Law & Government	1,012	0.48%
News	389	0.18%
Online Communities	8,203	3.86%
People & Society	3,242	1.52%
Real Estate	1,004	0.47%
Reference	31	0.01%
Science	45	0.02%
Sensitive Subjects	22	0.01%
Shopping	4,253	2.0%
Sports	1,059	0.5%
Travel	1,514	0.71%

We see that the distributions of each category across the two datasets are mostly similar, with minor variations per category. We note that the Computers & Electronics, Business & Industrial, and Arts & Entertainment make up the majority of the website categories. This is likely due to the nature of these websites being much more general than the other websites. For example, Business & Industrial, in the original training dataset, is used as a generic catch-all for most businesses and companies that do not fall into the other categories. It is likely

Table IX
.AU DATASET WEBSITE CATEGORIES

Category	Count	Percentage
Adult	518	0.06%
Arts & Entertainment	126,648	14.3%
Autos & Vehicles	9,030	1.02%
Beauty & Fitness	5,653	0.64%
Books & Literature	953	0.11%
Business & Industrial	165,715	18.71%
Computers & Electronics	375,836	42.43%
Finance	3,626	0.41%
Food & Drink	7,560	0.85%
Games	631	0.07%
Health	16,109	1.82%
Hobbies & Leisure	10,221	1.15%
Home & Garden	36,134	4.08%
Internet & Telecom	37,357	4.22%
Jobs & Education	9,980	1.13%
Law & Government	4,622	0.52%
News	1,154	0.13%
Online Communities	29,724	3.36%
People & Society	11,953	1.35%
Real Estate	6,142	0.69%
Reference	169	0.02%
Science	144	0.02%
Sensitive Subjects	81	0.01%
Shopping	16,008	1.81%
Sports	4,835	0.55%
Travel	4,886	0.55%

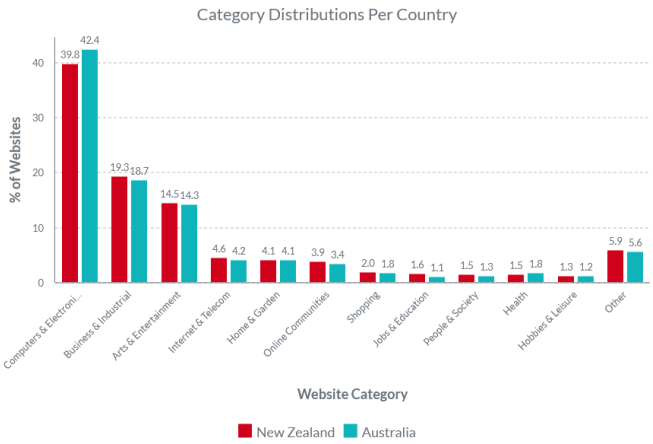

Figure 8. Website category distributions across the .NZ and .AU datasets.

that with either more specialized categories & labels a less weighted distribution would be seen.

However, there are some interesting points we can draw from this data. For example, we see that NZ has a 10.4% higher proportion of Shopping websites - possibly caused by our isolation from other countries, driving an increased online shopping market. The same is seen with Travel websites, with New Zealand having a 30% higher proportion of Travel websites than Australia (0.71% vs 0.55%) - likely caused by tourism being one of our largest exports, especially compared to Australia's comparatively empty regions.

With an improved training set, label set, and features the accuracy of this data could be improved as discussed in the following section.

D. Limitations and Possible Improvements

Although this system achieves its purpose, there is definite room for improvements and additional functionality.

1) Outdated URL Lists: Although the lists of URLs used with this system contains a vast number of websites, the list itself is outdated - resulting in a lot of URLs and their websites no longer existing online. With a more up-to-date version of this dataset we would be able to retrieve more accurate data and thus have a more accurate view of the web presence of the countries we survey, as the roughly 200,000 URLs in the ".NZ" dataset account for only 26% of the registered ".NZ" domains as of 2022 [21].

2) Generalised Training Data: As the training data of the classification model is very generalised with the Computers & Electronics, Business & Industrial, and Arts & Entertainment categories making up the vast majority of the labelled data, a more specific dataset would allow for more specific classification of websites. This would remove the heavily weighted distribution that currently exists in both the predicted and training datasets and would provide the classification models more data to train with, especially in the categories that have very few websites (Sensitive Subjects, Science, etc.).

3) Unreliable Website Data: As the classification model relies on a website's title, description, or keyword tags to classify a website, not each website can be classified accurately as many are missing these tags. Designing a system that can dynamically extract important or specialised website information in lieu of these tags could improve classification accuracy along with increasing the range of websites that can be classified.

4) Web Server and API: In its current state, the project is contained entirely to the machine it is installed on. It cannot be accessed from other machines through any means such as an Application Programming Interface (API), or an online GUI interface in the form of a website. A major improvement to the system would be to make it available both online and to other machines through both an API and website interface. The API could be rate-limited or unlimited, requiring an API key and allowing users to send website classification requests over the internet. The system could then invoke the command-line interface, or interface directly with the Python code, to scrape and classify the provided URL, returning the business category back through the API - or returning the cached website category if it exists in the database. Additionally, a website interface could be provided to showcase the API's functionality to users in a more user-friendly way. This could be done through the *Flask* library, as Flask is a Python library designed to allow for the creation of web-apps and web APIs [22]. Finally, this extension could open the door to commercialisation, as the API could provide batch processing or unlimited access to users who pay a one-off or monthly subscription fee.

5) International Website Dataset Classification: As this project has access to a dataset of not only ".NZ" websites but also other top-level domains (TLDs), the project could be extended to classify all the websites available in this dataset. This would require the setup of proper cloud infrastructure, as executing the project on the entire dataset would require multiple machines working in tandem. As the full dataset contains a list of approximately 247 million URLs across varying top-level domains, new approaches would need to be taken, either in scalability or delegation of work across multiple machines hosted in the cloud. Additionally, sustainability would become very important. It would be ethical to utilise a cloud hosting provider who uses renewable energy, like Oracle Cloud whose European datacenters are powered fully by renewable energy [23].

REFERENCES

[1] ".nz Statistics and Service Reports." [Online]. Available: https://docs.internetnz.nz/legacy/reports. Accessed October 9, 2023.

[2] G. Berardi, A. Esuli, T. Fagni, and F. Sebastiani, "Classifying websites by industry sector: a study in feature design," in Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1053–1059.

[3] "THE 17 GOALS - Sustainable Development." [Online]. United Nations. Available: https://sdgs.un.org/goals. Accessed October 9, 2023.

[4] "Website Traffic - Check and Analyze Any Website — Similarweb." [Online]. Available: https://www.similarweb.com/. Accessed October 9, 2023.

[5] "AWS Marketplace - Industry-Leading Website Categorization API." [Online]. Available: https://aws.amazon.com/marketplace/pp/prodview-n4uyzeoxotlpu. Accessed October 10, 2023.

[6] "Amazon Mechanical Turk." [Online]. Available: https://www.mturk.com/pricing. Accessed October 10, 2023.

[7] "HuggingFace - DistilBERT." [Online]. Available: https://huggingface.co/docs/transformers/model_doc/distilbert. Accessed October 9, 2023.

[8] "Welcome to Click - Click Documentation (8.1.x)." [Online]. Available: https://click.palletsprojects.com/en/8.1.x/. Accessed October 9, 2023.

[9] "Safe Browsing - Google Safe Browsing." [Online]. Google. Available: https://safebrowsing.google.com/. Accessed October 9, 2023.

[10] S. Choudhury, T. Batra, C. Hughes, and L. Lemmatizer, "Content-based and link-based methods for categorical webpage classification," 2016.

[11] A. Garg, N. Trivedi, J. Lu, M. Eirinaki, B. Yu, and F. Olumofin, "An evaluation of machine learning methods for domain name classification," in 2020 IEEE International Conference on Big Data (Big Data), December 2020, pp. 4577-4585, doi: 10.1109/BigData50022.2020.9377787.

[12] "Hugging Face - The AI community building the future." [Online]. Available: https://huggingface.co/. Accessed October 9, 2023.

[13] "Requests: HTTP for Humans™ - Requests 2.31.0 documentation." [Online]. Available: https://requests.readthedocs.io/en/latest/#requests-http-for-humans. Accessed October 9, 2023.

[14] "Selenium." [Online]. Available: https://www.selenium.dev/. Accessed October 10, 2023.

[15] "Selenium with Python - Selenium Python Bindings 2 documentation." [Online]. Available: https://selenium-python.readthedocs.io/. Accessed October 10, 2023.

[16] "MySQL." [Online]. Available: https://www.mysql.com/. Accessed October 10, 2023.

[17] "Transformers." [Online]. Available: https://huggingface.co/docs/transformers/index. Accessed October 10, 2023.

[18] "scikit-learn: machine learning in Python - scikit-learn 1.3.1 documentation." [Online]. Available: https://scikit-learn.org/stable/. Accessed October 10, 2023.

[19] "Kaggle: Your Machine Learning and Data Science Community." [Online]. Available: https://www.kaggle.com/. Accessed October 10, 2023.

[20] "Welcome to Colaboratory - Colaboratory." [Online]. Available: https://colab.research.google.com/. Google. Accessed October 10, 2023.

[21] ".nz Statistics and Service Reports - InternetNZ Product Documentation 8.0a1 documentation." [Online]. InternetNZ. Available: https://docs.internetnz.nz/legacy/reports/. Accessed October 10, 2023.

[22] "Welcome to Flask - Flask Documentation (3.0.x)." [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/. Accessed October 10, 2023.

[23] "Green Cloud - Oracle New Zealand." [Online]. Available: https://www.oracle.com/nz/sustainability/green-cloud/. Accessed October 9, 2023.