Crawling for Data Breach Reports

Design Document

Team SDDEC19-01

Client: Benjamin Blakely (Argonne National Laboratory)
Advisor: Thomas Daniels

Team Members:
Alec Lones, Jeremiah Brusegaard,
Mark Schwartz, Nolan Kim

Email: sddec19-01@iastate.edu
Website: http://sddec19-01.sd.ece.iastate.edu/

1: Frontal Materials

1.1: Table of Contents

1.2: List of Figures

1.3: List of Definitions

ML: Machine learning
CSO: Chief Security Officer
DNS: Domain Name System
UI: User Interface

2: Introductory Materials

2.1: Acknowledgement

We would like to acknowledge Benjamin Blakely of Argonne National Laboratory for his technical advice and for sponsoring this project.

2.2: Problem and Project Statement

Currently, there is no good way to be notified about every data breach that occurs, so important data breaches can easily fly under the radar. This can be dangerous for a company's security team, who needs to stay up to date with the latest data breaches in order to make sure their company is secure. In addition, it is important for these data breaches to be enumerated for security teams to reference later.

The purpose of this project is to serve as an early warning for CSO's on data breach reports that may affect their company. We plan to do this by implementing a web scraper to traverse the internet and identify data breach reports using ML. Our scraper will then store the breach reports in a database for future reference. With this information, the CSO will remain informed on current security threats to their organization.

2.3: Operating Environment

At its core, our scraper is a Python program, and can run on any machine with Python 3 and an internet connection. The machine must also be able to run a MongoDB database, which the scraper will use to store breach reports. This environment will require that we have constant uptime or at least near constant uptime to allow for uninterrupted web crawling and training for the ML algorithm. When the final product is finished, we will need an analyst to be on standby to train the model while its crawling after we do the initial training.

2.4: Intended Users and Uses

The first user we will discuss will be the CSO who generally will be very busy with day to day tasks and doesn't have time to spend sifting through data. With our tool, they will be able to see new breach reports and learn about potential security breaches that could have an effect on their company, and react accordingly. The purpose of this tool is to gather information, so the CSO would need to act on the information given to them.

The second user is an Analyst who will train our ML model. When the ML model makes a decision on whether or not a page is a breach report, the analyst can tell it whether it was right or not. Effectively, the analyst will have a supervisor role to the crawling model. The analyst will eventually not be a necessary user once the model is fully trained as it will be able to accurately classify data breach reports and directly share the findings with the CSO.

2.5: Assumptions and Limitations

2.5.1: Assumptions

- Internet connection will not be interrupted
- Power will not be interrupted
- Will not get blacklisted from too many DNS requests (We will attempt to limit requests to avoid this)
- Analyst training model will be fully competent in what a breach report is or is not
- There won't be resource limitations for crawling - hardware, etc.

2.5.2: Limitations

- Dark web is off limits
- Budget is $0
- Breach reports are limited to english
- One year to be created with taking into account we have other classes
- Reduced crawling speed to avoid getting blacklisted

2.6: Expected End Product and Deliverables

2.6.1: Core scraper

This deliverable is a Python scraper that crawls the internet and uses ML to identify breach reports. This deliverable will also have an interface for an analyst to train the ML model, as well as an interface for a CSO to view and sort the found data breaches.

2.6.2: Natural Language Processing Module

This deliverable trims down and processes the raw text from the scraper module. This processed text will then be fed into the ML code.

2.6.3: ML Module

This deliverable contains all the ML code and feature detection code. This module determines what constitutes a data breach and what does not.

2.6.4: Python Front-End

This deliverable will allows the two main users of the system to access it. The CSO will be given a data breach report interface while the analyst will be given tools to train the ML module.

2.6.5: Breach Report Database

This deliverable is a MongoDB database that the scraper will use to store breach reports. The database is solely for the crawler's backend functionality, and will not be accessed directly by the users.

2.6.6: Documentation

For this deliverable there needs to be adequate documentation on our project. Python is a loosely defined language so documentation will be much more important.

3: Specifications and Analysis

3.1: Proposed Design

So far we have done some prototyping with scrapy for doing web scraping. It seems to be the most widely used and accessible scraper for Python, and it has the most compatibility with our other libraries. So so far each of us have implemented some sort of prototype with scrapy. Others have added on to those prototypes to have other features we might use. Currently the farthest along prototype we have scrapes the given website as long as that website is not part of the blocked domains. It then parses that website and tokenizes the words of it. Then the words get lemmatized so that we can vectorize the webpage to be used by our ML algorithm. Currently we are in the process of prototyping a vectorizer that can take the processed website and turn it into statistical data. We have not prototyped a ML model yet, but are exploring different libraries.

Functional Requirements:

- Web scraper starts from a list of known breach reports to start crawling
- The web scraper can crawl new links from the seed urls
- Web scraper parses website into a list of words
- Parsed list of words gets processed (lemmatized)
- Processed words are fed into ML algorithm (via vectorization)
- ML can evaluate a webpage and get feedback from supervisor
- Breach reports are stored in a database as a link to the website they came from
- Eventually after enough training the model should not need supervisor
- Front end UI should display new breach report

Non-Functional Requirements:

- For the security of the school we will not crawl the dark web for breach reports
- Can be run on any machine with Python3 and a network connection
- Can run without interruptions to crawl multiple websites without crashing

Our project is relevant to a few standards. First we are using the PEP8 Style Guide Standard to promote readability and consistency.[1] Additionally we are using IEEE 829 and 1008 to format testing documents and unit testing.[2][3] These three standards will help us to write readable and consistent code, documents, and tests so that our project can be built upon by others if need be.

3.2: Design Analysis

So far, we have implemented a prototype Scrapy crawler that pipes output into a lemmatization agent. It starts crawling from www.grahamcluley.com/quora-hacked, and excludes a list of blacklisted URLs. It then uses the BeautifulSoup library to extract data from the scraped pages. The data is then passed into a lemmatizer, and words with strange characters are removed.

The client has analyzed our prototype and given it his approval, which shows that our design most likely doesn't have any problems that would invalidate it. However, we do have a few concerns about the specific functions of each of our implemented modules. First of all, when our scraper encounters a Twitter link, it wastes time scraping useless links on Twitter. This is probably true for other social media sites like Facebook or Reddit. In order to overcome this issue, we plan to have the scraper follow Twitter links when they're linked from other websites, but not from Twitter itself. In addition, we're not sure which characters should be blacklisted from our lemmatizer, as some technical words that would be useful to our ML agent may include numbers and special characters. On the other hand, we don't want to bog down our lemmatizer with useless information such as non-technical websites and blogs.

Our design, figure 1, is fairly straightforward, as the dataflow is just a straight line from the scraper to the database, save for the training module. It also has high cohesion and low coupling, as we have clear, defined modules that handle each part of the scraping or ML process. This design makes implementation easy, as we can focus on one module at a time before linking them together. It will also make testing easier, as we can test the modules individually and treat adjacent modules as black boxes.

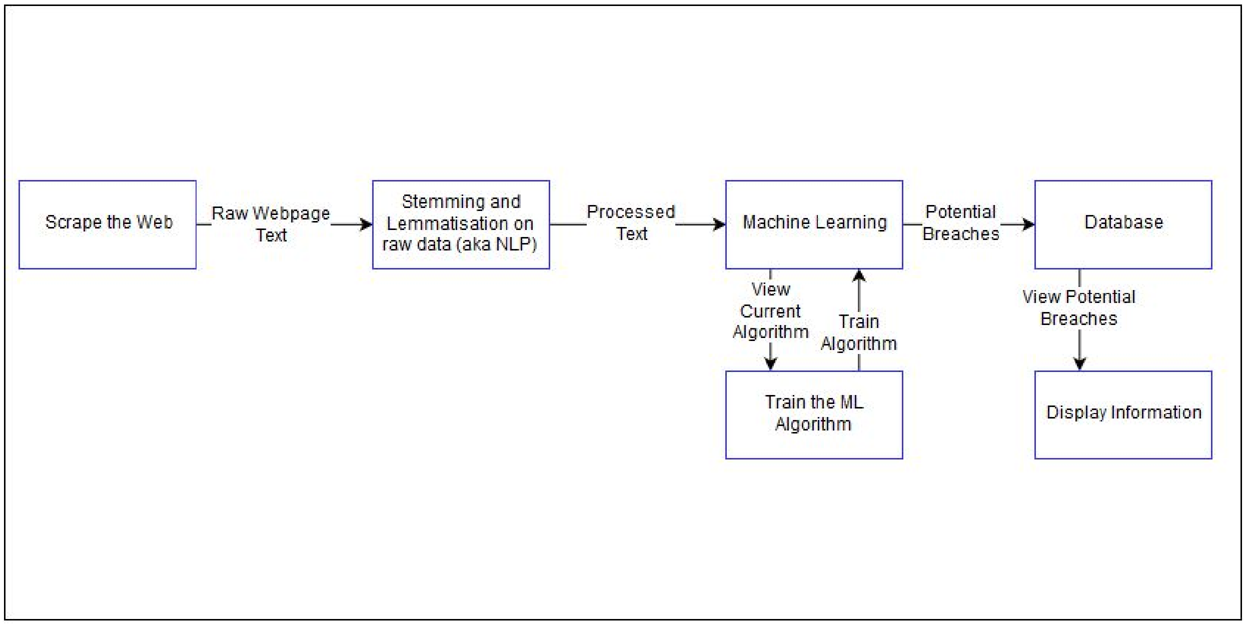

Figure 1: High Level Block Diagram

4: Testing and Implementation

4.1: Interface Specifications

The scraper, lemmatizer, stemmer, and vectorizer all stem from different libraries. Thus in order to connect them together, we will be writing the interfacing code. We plan to have separate modules so that the program could be either run on a single machine or distributed among multiple machines. PyUnit tests will test the interfacing between these different modules to verify correctness.

4.2: Hardware and Software

PyUnit - PyUnit is a Python equivalent of JUnit. It will be used to test different modules in order to verify correctness and interoperability. This will handle our unit, integration, and system tests.

Test Data - After the ML model has been fitted using the training data, a set-aside portion of Test Data will be fed to the model to determine accuracy. The Test Data will be helpful in determining the presence of overfitting our model to our data and correcting it if it exists. Additionally this data will be used to help tweak the model to balance accuracy, fit, and bias.

4.3: Functional Testing

PyUnit unit tests will be used to test the core functionality of the scraper, lemmatizer, and vectorizer. There will also be integration tests to verify the interoperability of the different modules. System tests will be written to make sure that the system works as a whole. Each of these three levels of tests will be run throughout development to ensure that adding new code doesn't break old code. To break down each component we will first test the scraper by monitoring that it is scraping all the pages we give it and not traversing sites that are blacklisted. For the lemmatizer this will consist of making sure that the lemmatized words don't consist of unreal words such as just an "s" with an apostrophe. For the vectorizer making sure that it is correctly adding features that we give it to the text.

4.4: Non-Functional Testing

We will test our code on both Windows and Linux in order to test for compatibility issues between operating systems. The code will also be tested to ensure that performance doesn't suffer when the scraper hits links that take it to Twitter or other link heavy pages. A performance test will also be implemented for the ML model to ensure that predictions and model generation is completed in a satisfactory time.

4.5: Process

First we will test the scraper, lemmatizer, stemmer, and vectorizer modules with PyUnit tests. After these modules have passed, we will test our ML model on a set of known and unknown data breach reports. We will change the model based on its ability or inability to recognize each as what category it belongs to. Finally we will test the overall system to make sure that it functions correctly and efficiently as a whole.

4.6: Results

As of the writing of this document, only prototypes have been created with no official testing performed yet. The only testing performed is verifying that different libraries work with each other and evaluating what interfacing code will need to be written.

5: Closing Material

5.1: Conclusion

Currently we have multiple prototypes for scraping websites and lemmatizing data. We aim to connect the scraper, lemmatizer, vectorizer, ML model, database, and front end through a series of interfaces similar to a pipeline. As much of the pipeline will be multithreaded as possible in order to maximize speed and efficiency. This solution is the most efficient solution that we have tested. Scrapy and the other libraries used are reputable and widely used along with being efficient. Additionally the code connecting these libraries will be multithreaded and aim to efficiently transfer data through the pipeline. With these goals in mind, this solution surpasses all others in efficiency and code elegance.

5.2: References

1. “PEP 8 -- Style Guide for Python Code.” *Python.org*, www.python.org/dev/peps/pep-0008/#introduction.
2. “829-1998 - IEEE Standard for Software Test Documentation.” *IEEE*, www.standards.ieee.org/standard/829-1998.html
3. “1008-1987 - IEEE Standard for Software Unit Testing.” *IEEE*, www.standards.ieee.org/standard/1008-1987.html