Machine Learning Engineer Nanodegree
Capstone Project

Bassim Lazem
November 22th, 2018

Detecting Duplicate URLs Using Deep Recurrent Networks

I. Definition

Project Overview

Many websites on the World Wide Web have multiple Uniform Resource Locators (URLs) for the same page content. This happens for various reasons, for example, to ease user navigation, many sites provide shortcut paths for the same page. Another frequent reason (especially on the deep web) is adding parameters to the URLs that do not change the page content. In addition, some sites use duplicate urls for load balancing and ensure fault tolerance. This is a problem for web crawlers as they collect a large number of duplicate content, wasting time and resources downloading the same content over and over again. Moreover, this DUST (Duplicate URLs with Similar Text) issue results in a bad user experience, as the crawler-retrieved documents contain a large volume of duplicate content showing the same results multiple times as incorrectly different ones. This project attempt to solve this issue using Machine Learning to provide on-line detection during crawling without the need to store the duplicate pages.

Problem Statement

The goal of this project can be stated as follows: While a crawler is visiting a URL in some site, determine which hyper-links it should follow next, discarding the identical ones to the current page. More specifically, given two URLs, predict if they direct to the same page or different ones. The end product is a ML middleware filter that is trained to classify the links the crawler is looking up, either a DUST link so it shouldn’t be revisited, or a unique one that should be added to the crawling queue, resulting in better resource utilization and overall enhanced information retrieval.

Several approaches have been proposed to mitigate this issue, with varying degrees of success. Broadly classified into two categories, detecting duplicate URLs by analyzing the URL addresses information, or by fetching and comparing the pages content using syntax and semantic analysis. The first approach tries extracting normalization and tokenization properties of the URLs to detect duplication rules. The second approach uses similarity techniques between documents and determine duplicate pages using a threshold.

The proposed solution here is to utilize a mixture of both these approached to create a convenient customized solution. The idea is to create a machine-learning model to discover these duplicate URLs rules by itself. The model will be given only the URLs and a score between each pair. This score will come from the content of the pages. Therefore, the data is going to be sampled from a target site only during training, whereas the deployed model will predict on the URL strings alone. Using machine-learning especially deep learning is powerful in its ability in manually extract features, learning a representation from the URLs sequence of characters. Using LSTM in URL analysis has already shown great promise in Phishing Detection, and domain-generated algorithms. And in fact, this [5] is the inspiration for me to apply the same technique on the DUST problem.

Evaluation Metrics

Prediction results are going to be evaluated on the cross entropy between the predicted values and ground truth ones. The ground truth is the similarity score between two HTML documents, whereas the predicted values are coming from these documents URLs only. If the model is able to infer the duplication rules, it should have high prediction accuracy.
The cross entropy is evaluated according to the following function:

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N}\sum_{i}^{N}[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)]$$

- y^i is the probability of ith object belonging class 1, as calculated by classifier.
- yi is the actual label of the ith object; could be either 0 or 1

The similarity between two pages is calculated using cosine distance:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}}$$

Using the dot product for two vectors where A and B are the components of the vector (features of the HTML document, or TF values for each word of the page).

Finally, the formula for accuracy is the following:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

II. Analysis

Data Exploration

The datasets are collected from different open sites, crawling a subset sample from each to train the algorithm. The following sites are the ones used:

- https://ubuntuforums.org
- https://digitalspy.com
- http://forums.mozillazine.org
- https://bodybuilding.com

Each site is dynamic, has large number of pages backed by millions of users. More importantly, each one of them exhibits the duplicate URLs issue. For example, the following urls are the same:
https://ubuntuforums.org/showthread.php?t=616672
https://ubuntuforums.org/showthread.php?t=616672&p=3796756

Another example:
https://forums.digitalspy.com/discussion/2294848/the-jeremy-vine-show-weekdays-channel-5
https://forums.digitalspy.com/discussion/comment/91010367#Comment_91010367
https://forums.digitalspy.com/discussion/comment/91009893#Comment_91009893
https://forums.digitalspy.com/discussion/comment/91076484#Comment_91076484
https://forums.digitalspy.com/discussion/comment/91108848#Comment_91108848
https://forums.digitalspy.com/discussion/comment/91108848#Comment_91108848

Plus 21 other of the comment links in the page.

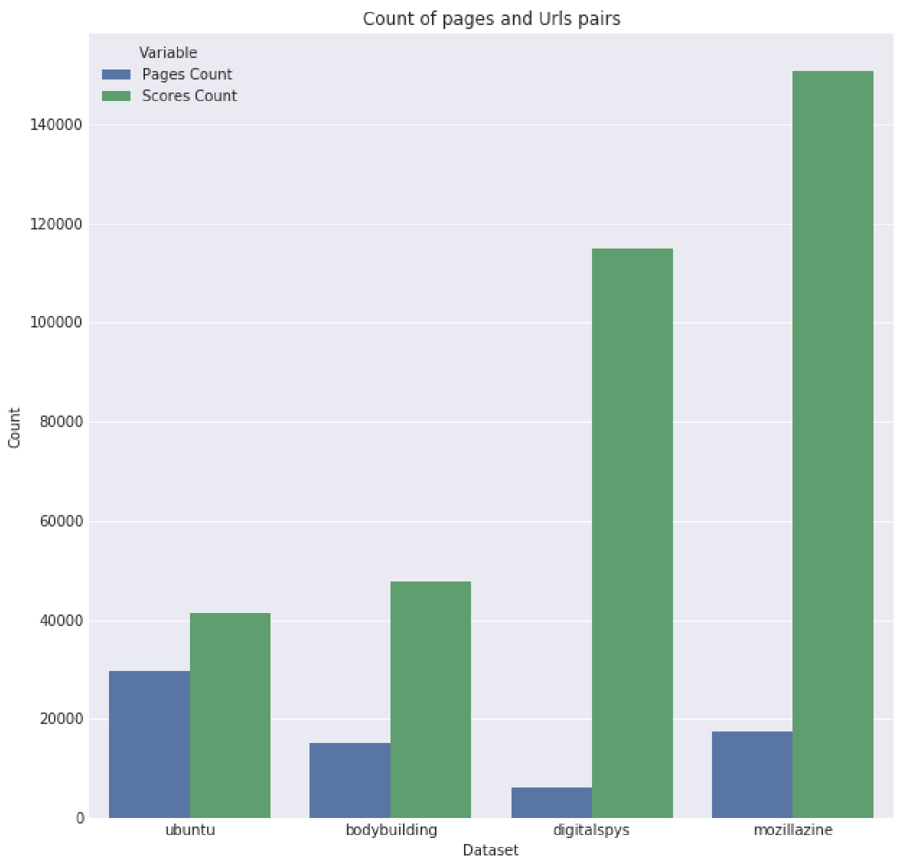
The created crawler is run over part of each site, collecting the training samples (while respecting the robots.txt rules of the site). The crawled data is saved as a SQLite database in a"*pages*" table with the following fields:

- **id** – The id of the training page
- **url** – The URL of the page
- **referrer** – The URL of the page the originated request
- **content** – The page HTML text.

Using these fields, another table "*scores*" is constructed showing the similarity between two URLs with the following fields:

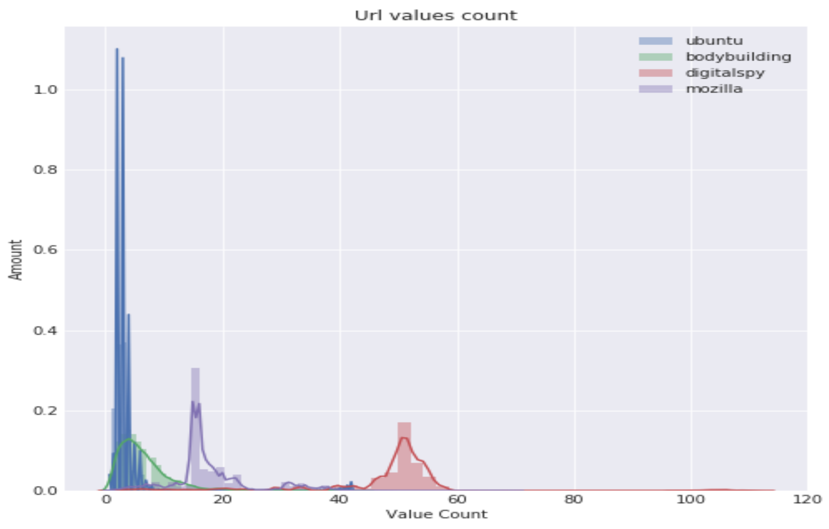
- **url1** – first URL to compare
- **url2** – second URL to compare
- **score** – similarity score between the two URL pages

In order to construct the "scores" table, each referrer URL is paired with all the URLs it is referring to, plus the combinations of these URLs. The following chart shows the number of crawled pages for each site and their number of URLs pairs.



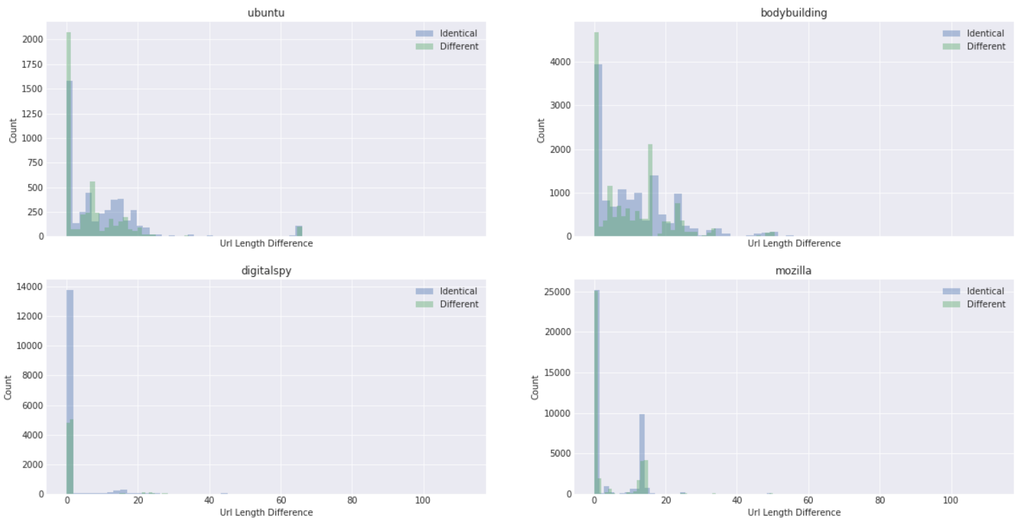
Exploratory Visualization

As it is shown on the chart above, there is a variance between the number of scores pairs for each site. This can be clearly shown we find how each site URL is mentioned in the "scores" table.

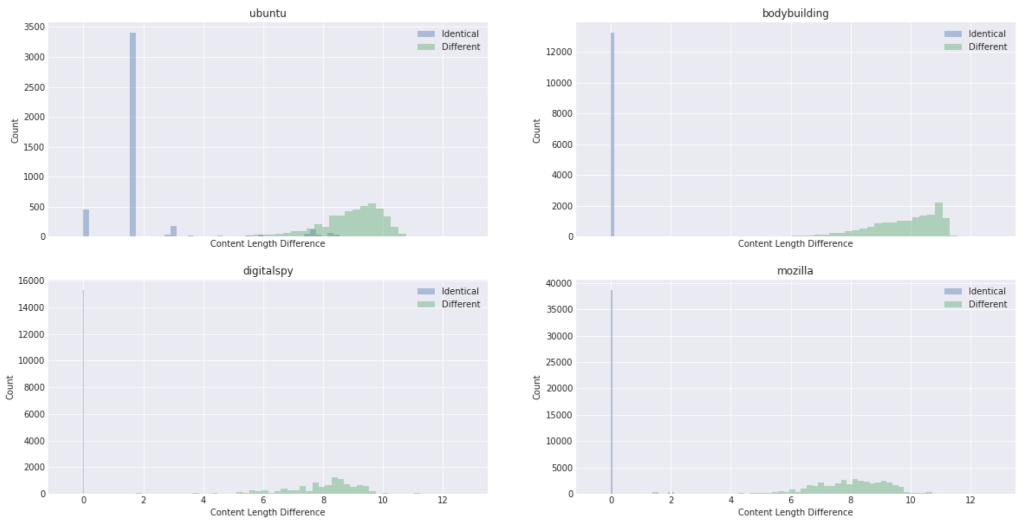


From here, we can see there is so much inter-relatedness for both *Digital Spy* and *Mozilla Zine*, while *Ubuntu forums* URLs appear in few records of the scores table, whereas *Bodybuilding* is in between.

Each URL is a string representation for a resource on the web, and while each URL follows a structured scheme, differentiating between identical and different URLs is an ambiguous task as it is specific to the dynamic rules for each site, that is why computing similarity between two URLs strings does not supply a direct measure for identical pages. This can be shown in the following chart, where the length difference between the URLs is plotted. As can be seen the variation in length between identical and different URL pairs is quite similar and difficult to tell apart.



On the other hand, the length difference of the string content of these URLs (the HTML sources) should be by definition the same for identical pages and different for unique ones. The following chart shows the log string length difference for the URLs content pairs.

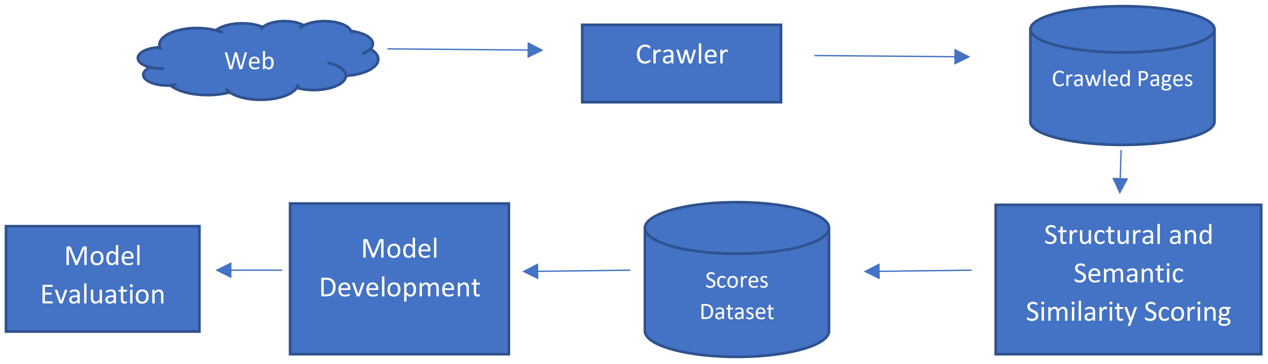


This is in contrast quite clear separation between identical and different URLs, and this would provide the ML model a training data to infer the URL rules for identical pages by itself.

It should also be noted that string length is a trivial separation score, and better and more nuanced measures we will discussed in the coming sections.

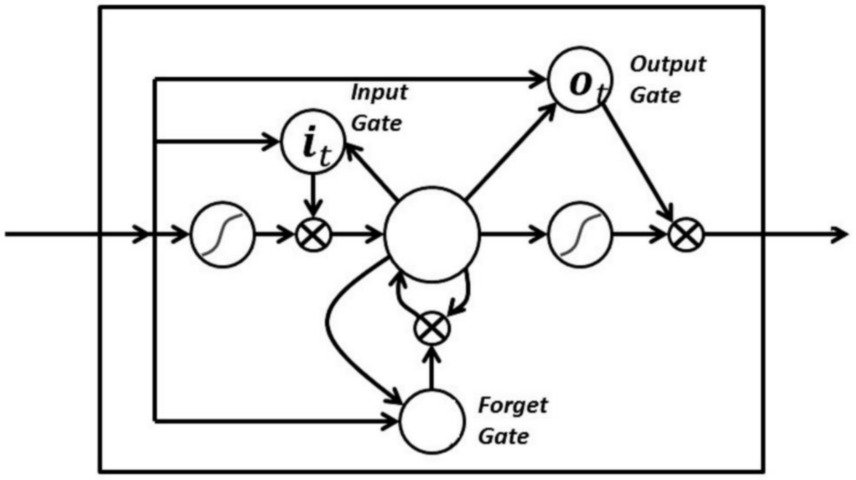
Algorithms and Techniques

The goal of the system is to create an automated system to train a recurrent neural networks model to detect duplicate URLs. Below is a diagram of the solution architecture.



LSTM algorithm is an extension for recurrent neural networks, to enhance their memory capacity. LSTM enable RNNs to remember their inputs over a long period. This is because LSTMs contain their information in a memory cell with several multiplicative gates, similar to a computer memory the LSTM unit can read, write and delete information from its memory.

The LSTM unit architecture is shown in the diagram, typically it has an input gate to control the input of information from the outside, a forget gate to decide whether to keep or delete the information in the internal state, and an output gate that allows or prevents the internal state to be seen from the outside.

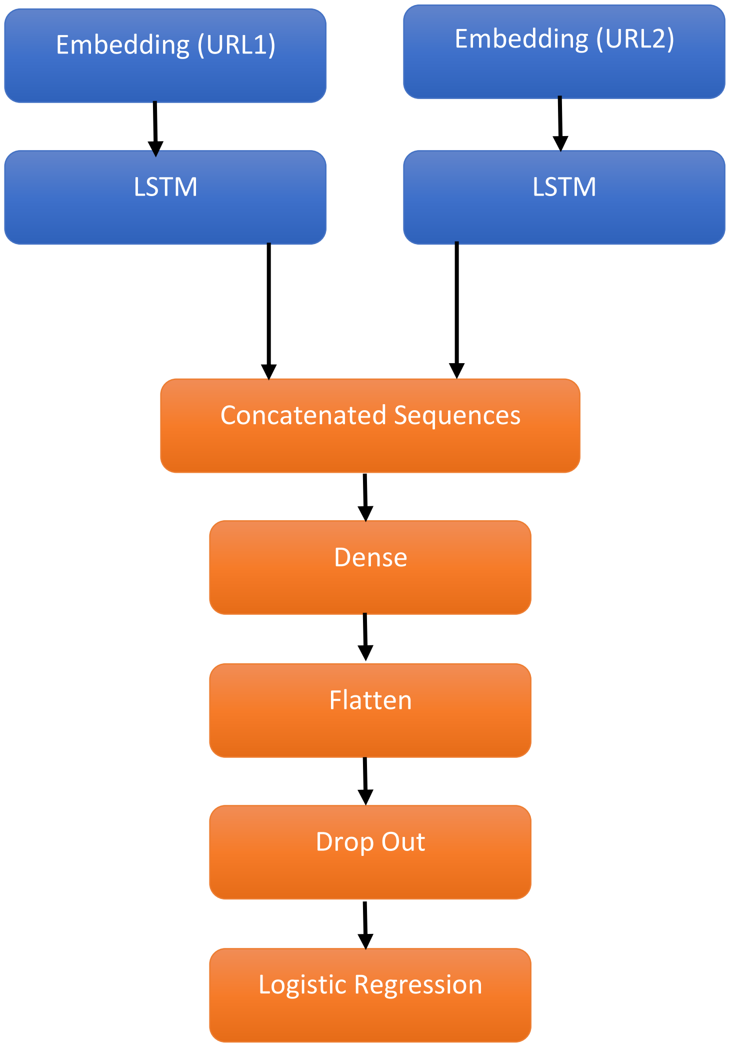


Having an internal memory makes Recurrent Networks to be one of the most powerful algorithms to model sequential patterns like text, speech, audio, video and many other time-dependent data, providing a deeper understanding of the data and its context. This is the motivation for using them in this problem. The LSTM units are used to build a model that receives two URL inputs as a sequence of characters and predicts whether these URLs are identical.

Each input URL character sequence is translated by an initial embedding layer, which are both fed into a LSTM layer, returning a dense layer representation of the sequence output of the LSTM.
After that, both outputs are concatenated, and then fed into a dense layer with half of the input size. Finally, the output of the last layer is flattened and the classification is performed using logistic regression.

The network is trained by backpropagation (through the Adam optimizer) using a cross entropy loss function and dropout in the last layer.

The architecture is shown in the following diagram.



Benchmark Model

Random Forest algorithm is used to provide a benchmark and sanity check for deciding whether the LSTM network is producing meaningful output.

III. Methodology

Data Preprocessing

Boilerplate Removal

In order to obtain an accurate similarity score between identical pages, comparing their HTML sources directly will be error prone due to boilerplate content. This is all the content that's not part of the main page content, so all navigation menus, headers, footers, ads, copyright notices..etc are all boilerplate. Removing this is essential since two identical pages can have different boiler-plate content when the URL is changed, even with same URL the pages can change (in case of ads for example).

A new column "content text" is added to the "pages" table and filled with the extracted content text of the HTML column; it is this processed column that is used for semantic similarity.

Similarity Scoring

As mentioned in prior sections, the "scores" table is constructed from the crawled URLs and their referrers. Each row of this table is filled with the semantic similarity between the pages content text, using cosine distance between their TF vectors. Because this is a score from zero to one and the problem is classification, an additional pre-processing step is needed to convert these multiple values into either one as identical and zero as not. After inspecting the values, a threshold of 0.99 is chosen to cut between duplicate and different URLs.

Implementation

The implementation of the solution consists of a set of Python files and notebooks, from collecting the data to the final model training and evaluation.
The following is a breakdown of each part.

Content Crawler

This section is a *Scrapy* crawler containing a single spider that accepts a main URL of a site to crawl it, while saving a dataset of the visited pages from the website domain.
The files under this project:

- **content_spider.py**: Main access for the crawler. Contains the callback parse function, which is run after a page is downloaded, returns a *Scrapy* page item.
- **items.py**: Contains the PageItem class, holding relevant information to be saved into the db.
- **pipelines.py**: Contains the DatabaseHandlerPipeline class, which adds a page item to the database if it does not exist.
- **middlewares.py**: Contains the DuplicateFilterMiddleware class, checks if the URL and its referrer are identical, raises an IgnoreRequest exception when they are.
- **dup_detect_filter.py**: This is the implemented model, to be used for duplicate detection after the model is trained and deployed.
- **settings.py**: Holds settings for the project, notably:
    - Download delay.
    - Required number of pages.
    - Database settings.
    - Which middlewares and pipelines are activated.

To run the spider on the required main URL use the following command:
scrapy crawl content -a url=<<MAIN_URL>>
This will crawl the website and the save the pages it visited in a *SQLite* database, and it stops when it reaches the required number of pages.

Similarity Scoring

Provides the boilerplate removal and similarity scoring calculation for the collected databases.

- **db_handler.py:** Contains the database handling functions, to read and update the tables.
- **urls_pairing.py:** Helper functions to make pairs of the URLS and their referrers.
- **semiliarity_scoring.py:** Contains the functions for boilerplate removal and similarity scoring.

To remove boilerplate content, several tools have been investigated and the best performance was for *jusText*[12], which is a multi-lingual heuristic based boilerplate removal tool. It is particularly successful at detecting non-grammatical sentences, leaving only the real content of the page.

For similarity scoring, term frequency TF vectors are made then their cosine distance is calculated. *Sklearn* is used for both tasks.

Exploratory Analysis and Data Preparation

This section has one *jupyter* notebook **exploratory_analysis.ipynb** contains the code for performing exploratory analysis on the databases, generating the visualization charts. In addition to converting the datasets into their final pre-training csv format. *Pandas* and *Seaborn* are used.

Model Training and Evaluation

This final section contains also one notebook **model_training.ipynb**, contains the code for creating and training the model. It performs the followings:
- Loading the datasets into Pandas data frames.
- Calculate some configuration prior to training (maximum URL length, number of unique characters/tokens, and char2idx and idx2char dicts for converting the URLs into numbers).
- Split the training data randomly into a training and validation set, with 20% of data used for validation.
- Benchmark the performance using random forest classifier.
- Define the model with the architecture shown above. Keras is used for this.
- Train and evaluate the performance on each dataset.

The biggest hurdle during implementation was acquiring the different datasets and preparing the scores table. This consumed a considerable time and resources to accurately compute the similarity between all URL pairs. Moreover, several similarity scores have been attempted before settling on cosine distance, notably HTML structural similarity provided convenient results but semantic similarity was much better in outlining identical pages.

Refinement

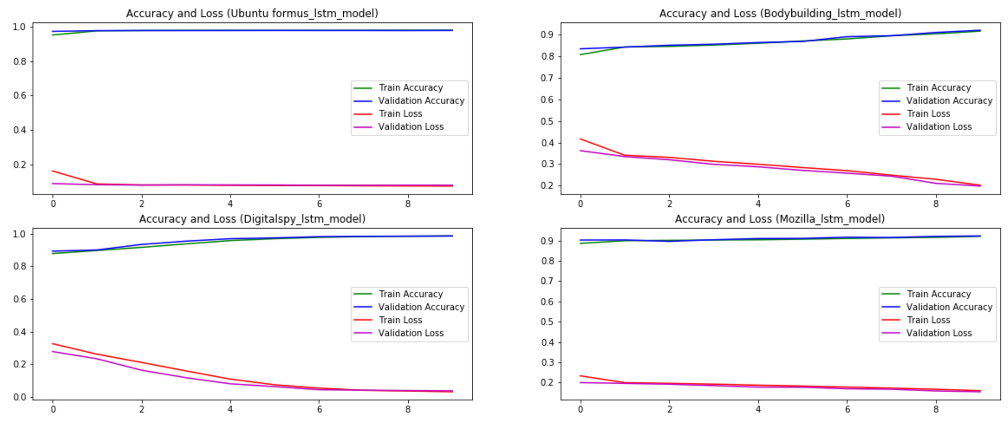
Several hyper-parameters of the LSTM algorithm were adjusted to achieve better results. The initial model achieved +90% for the 4 sites after +100 epochs. After various experimentations, this result is achieved after less than 10 epochs. These refinements include:

- Concatenate the two URLs after going through the LSTM return sequences.
- Add Dense layer after concatenation.
- Add Dropout to the LSTM layer and after the Dense layer.

IV. Results

Model Evaluation and Validation

The following chart shows the final model accuracy and loss results across the four training and validation sets.



After these results, I have crawled another set of pages from the site *Ubuntu forums*, then I ran the model on this not-seen-before dataset to predict the outcomes against their similarity scores. The accuracy score turned out to be .78 with the following confusion matrix:

	True	**False**
Identical	134	4
Different	2520	753

Justification

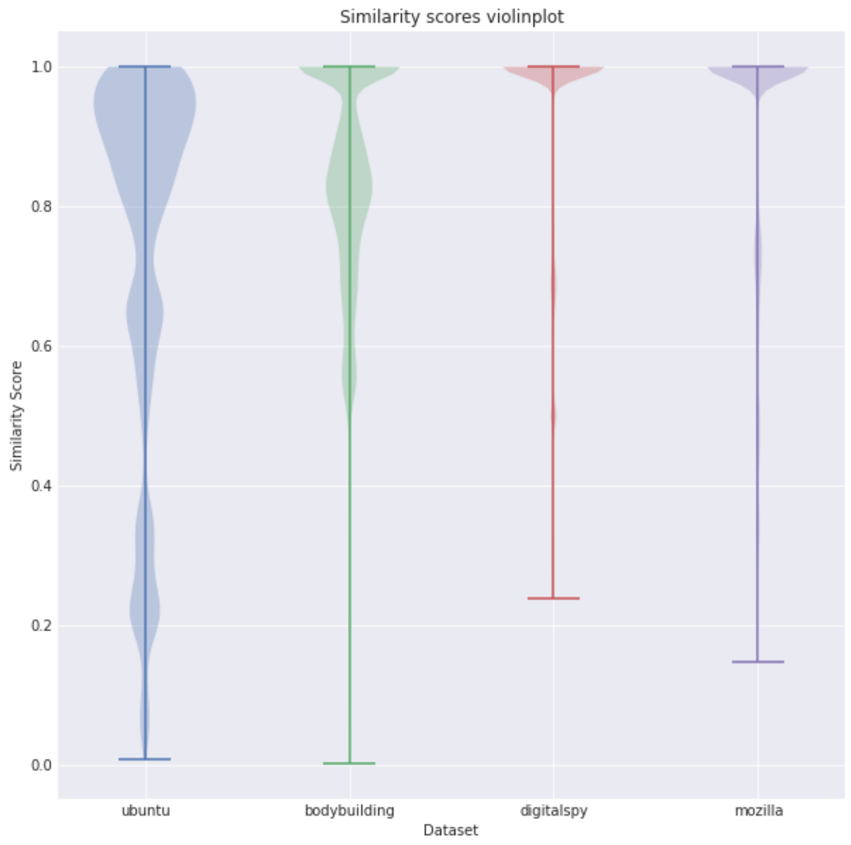
The final solution outperforms the Random Forest benchmark model as can be seen from the table.

	Random Forest	LSTM
Ubuntu Forums	0.95	0.97
Body Building	0.82	0.91
Digital Spy	0.88	0.98
Mozilla Zine	0.89	0.92

With these results, I believe the model can provide a suitable filtering for duplicate detection middleware. The results from the URLs that have not been included during training indicate the model can generalize.

V. Conclusion

Free-Form Visualization



This violin plot shows the similarity scores for each site. That is before categorizing them as identical or different using a threshold. It shows there is a lot of variance for the sites *Ubuntu forums* and

Bodybuilding than the two others. This is a direct effect of pairing the referrer URLS and theirs combination. The point of choosing this is to generalize the crawling strategy, but it is not the only way of pairing the URLS, and changing this would affect the resulting training set.
It would be interesting to try other pairing schemes together with controlling the threshold to train the model to filter other types of similarity not just identical pages. For example, all the sites chosen are discussion forums, which have two predominant page types: the topic pages showing the content, and the menu pages showing links to the topic pages. For this, the model can be trained to focus only on content pages.

Reflection

In this project, we explored how to detect duplicate URLs using deep recurrent networks. The end-to-end solution can be summarized as:

- Crawl a subset of target site(s) into a URLs-content database.
- Convert this dataset into URLs scores (using content similarity)
- Train a machine-learning model to infer the URL duplicate rules.
- Use this model on new pages while crawling to avoid duplicate ones.

I believe the most interesting part of this project is the lack of manually created training set. Since it can be automatically calculated from a predefined similarity score such as the cosine distance. This can save substantial resources used in manually selecting the duplicate URLS. In addition, using deep learning techniques such as LSTM showed promise in automatically detecting the duplicate detection rules without any use of feature extraction of the URL lexical or semantic properties. The LSTM took only the string representation of the URLs, making it better to generalize on any website regardless of its precise rules. In conclusion, I think the solution showed a good proof of concept for duplicate detection problem, and it can be used in real world crawling projects for detecting this issue and similar ones to it.

Improvement

There are several aspects of this POC that could be improved. First, each website was trained as a dataset on its own, these datasets can be combined together to try to generalize on various websites. As can be seen, the chosen sites are all forums, and adding them all together may be a good duplicate filter for other forum sites.

In addition, other similarity scoring methods can be used to detect different classes of the site URLs not just duplicate ones. Finally, it would also be interesting to try other deep learning techniques, notably convolution neural networks CNN, since it is also good for string sequences and it can be much faster than LSTM networks.

References

- [1] Detection of Distinct URL and Removing DUST Using Multiple Alignments of Sequences https://www.irjet.net/archives/V3/i1/IRJET-V3I1162.pdf
- [2] Detection and Removal of DUST: Duplicate URLs with Similar Text Using DUSTER http://www.ijircce.com/upload/2017/january/112_Jyoti%20Paper.pdf
- [3] A Graph Based Approach for Eliminating DUST Using Normalization Rules http://www.ijircce.com/upload/2016/april/181_A%20Graph.pdf
- [4] ELIMINATE DUPLICATE URLs USING MULTIPLE ALIGNMENT OF SEQUENCES http://www.mjret.in/V2I4/M59-2-4-10-2015.pdf
- [5] Classifying Phishing URLs Using Recurrent Neural Networks https://albahnsen.com/wp-content/uploads/2018/05/classifying-phishing-urls-using-recurrent-neural-networks_cameraready.pdf
- [6] Cosine Similarity for Vector Space Models http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/
- [7] Metrics for Evaluating Machine Learning Algorithms https://www.dezyre.com/data-science-in-python-tutorial/performance-metrics-for-machine-learning-algorithm
- [8] Recurrent Neural Networks and LSTM https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5
- [9] Overview of Text Similarity Metrics in Python https://towardsdatascience.com/overview-of-text-similarity-metrics-3397c4601f50
- [10] Keras Malicious URL detector https://github.com/chen0040/keras-malicious-url-detector
- [11] Analyzing Multiple Data Sets https://anenadic.github.io/2014-11-10-manchester/novice/python/03-loop.html
- [12] jusText: Heuristic based boilerplate removal https://github.com/miso-belica/jusText