Pagerank Estimation with Deep Graph Networks

Timo Denk, Samed Güner

June 2019

1 Notation

This section introduces the mathematical notations used throughout this document. It is taken from the book *Deep Learning* by Goodfellow, Bengio, and Courville (2016).

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \| Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes we use a function f whose argument is a scalar but apply it to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathbf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathbf{C} = \sigma(\mathbf{X})$, then $C_{i,j,k} = \sigma(X_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\text{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}^{(i)}$	The i-th example (input) from a dataset
$y^{(i)}$ or $\boldsymbol{y}^{(i)}$	The target associated with $\boldsymbol{x}^{(i)}$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}^{(i)}$ in row $\boldsymbol{X}_{i,:}$

2 Introduction

The world wide web has evolved to become a common source of information and entertainment for people from all around the globe. The number of active websites exceeds 100 million by far. People commonly access these websites through the internet using a webbrowser on their computer or hand-held devices and consume the content in graphical or auditory format.

Some website attract more visitors than others. As of January 2018, the Alexa page ranking[1] lists `google.com`, `youtube.com`, `facebook.com`, `baidu.com`, and `wikipedia.org` as the top five websites. Ordering websites by their popularity can be done in different ways: One might think of the total number of links pointing to a website, the visitor count, or visitor count combined with the average time people spend.

Newly published websites will initially be assigned a low rank, until their popularity grows and statistics start reflecting that. An experienced user, however, might be able to have a rough feeling for whether a website they see has potential to become popular or not, solely based on its content and look. To the best of our knowledge, there is a shortcoming in tools, which estimate the expected popularity of a website, thereby replicating the behavior of a human visitor.

In this work we model a pagerank estimator using methods of computer vision. More specifically, we train a graph network with convolutional building blocks to *look* at screenshots taken from web page of a domain and try to predict the expected pagerank.

The world wide web has been designed with hyperlinks. They allow users to navigate between web pages. A website typically consists of several web pages, which (often, but not necessarily) point to each other with hyperlinks. Such a website can be interpreted as a graph, where nodes correspond to web pages and edges to hyperlinks connecting two of them. We try to exploit the information contained in the graph structure by applying graph networks to the ranking problem.

The advantages over statistical approaches (link counting, visitor counters with toolbars, etc.) are the following: Once trained, our model can generalize to new, unseen websites, without need to spend time crawling the web. It is purely vision-based and has therefore access to (most of) the information that website visitors consume. On the other hand, the ground truth used to train

[1] The top 500 sites on the web: `https://www.alexa.com/topsites`

our model is the result of the aforementioned statistical analyses. Consequently, its predictions cannot be better on known website.

Traversing the web page graph of the internet or a single website is referred to as crawling. In order to create a screenshot dataset that contains the meta information we want, we have developed a webcrawler. It has visited 100,000 web pages and took screenshots of them. The crawler is described in this document as well.

Our main contributions are:

- Creation and release of a pagerank dataset with two versions: (v1) A single screenshot for each of the top 100k websites and (v2) 100k graphs, where each graph represents one of the websites with screenshots and meta data.
- Application of graph networks to the problem of pagerank prediction.
- Answering of the question to what degree purely vision-based information from websites is sufficient to estimate their rank.

This work is a student research project carried out by two students of Applied Computer Science from Coorporative State University Baden-Wuerttemberg, Karlsruhe.

The remainder of this document is structured as follows:

Write one sentence per section.

The mathematical notation used throughout this document follows the one defined in the chapter *Notation* of the Deep Learning book by [GBC16].

3 Background

!!! name the other website ranking paper from 2006 somewhere

3.1 Learning to Rank

Learning to rank is the application of machine learning to the creation of models that can rank. Ranking, in its most general form, is the ordering of a collection of documents. The book *Learning to Rank for Information Retrieval* by [Liu+09] enumerates three distinguishable approaches to phrasing and solving ranking problems. They are referred to as pointwise, pairwise, and listwise, and shall

be explained briefly in the following. The second part of this section deals with ranking metrics.

The **pointwise approach** deals with models that determine the rank of a single sample. It can be subdivided into the three subcategories (1) regression based, (2) classification based, and (3) ordinal regression based. For (1), the problem is considered a regression problem, where the rank of a sample is interpreted as a point on the real number line which the model seeks to predict correctly. With (2), the ranks are discrete and considered classes, for a given input, the model predicts a class which can be mapped to a rank. Ranks can also be grouped into bins, such that an entire range of adjacent ranks corresponds to a single class. While the classes in (2) do not have an ordering that is necessarily apparent to the model, the third subcategory (3) takes the ordinal relationship of ranks into account. The goal is to find a scoring functions such that thresholds can be found to distinguish the outputs into the different, ordered categories.

A possible loss function for pointwise training is to interpret the scaled and rounded model output y as the predicted rank of an input. A loss function could simply penalize divergence from the target output $\hat{y}$, e.g. in a quadratic fashion, $(y - \hat{y})^2$.

Opposed to the pointwise approach, the **pairwise approach** does not seek to predict the rank of a single sample. Instead the focus is on predicting the relative order between two given samples, i.e. whether the first sample x_i has a greater or a lower rank than the second one x_j. This problem can be considered a binary classification task on a tuple input (x_i, x_j), where the first class means $x_i \triangleright x_j$ (x_i has a higher rank than x_j) and the second class means $x_i \triangleleft x_j$.

[Bur+05] propose a probabilistic cost function for pairwise training, that is elements are being ranked relative to each other. They represent the ground truth as a matrix $\overline{\boldsymbol{P}} \in [0,1]^{m \times m}$, where $\overline{P}_{i,j}$ is the probability of sample x_i to have a higher rank than x_j. The model itself is a function $f : \mathbb{R}^d \to \mathbb{R}$ mapping a sample x_i to a scalar. The loss for two given samples x_i and x_j is defined as

$$-\overline{P}_{i,j} \left(f(x_i) - f(x_j)\right) + \log\left(1 + e^{f(x_i) - f(x_j)}\right) \tag{1}$$

This formulation has some favorable properties: It allows for uncertainty in the relative ordering of items in that the $\overline{\boldsymbol{P}}$ is a probability and could be e.g. $\overline{P}_{i,j} = \frac{1}{2}$ if the relative ranking of two items was unknown. Furthermore, the loss asymptotes to a linear functions. According to the authors, this is likely to be more robus with noisy labels than a quadratic cost.

The **listwise approach** deals with models that take an entire list of samples as their input with the goal of ordering them. This can be either done by mapping them to real values and penalizing wrong ordering, or mapping the samples discretely to a permutation.

In real world applications, the training of ranking models can be much more complicated. For example, updating a ranking model based on what users click leads to wrong updates because the feedback from users is biased. [JSS16] deal with that problem and propose *unbiased learning to rank*. This is, however, beyond the scope of this work because we do not deal with direct user feedback.

Once a model has produced a ranking, it must be evaluated. Since the magnitude of a loss function is ofter rather less meaningful, other measures, so called **ranking metrics**, have been defined. [Pas+18] name four, namely *Mean Reciprocal Rank (MRR)*, average of positions of examples weighted by their relevance values (*ARP*), *Discounted Cumulative Gain (DCG)*, and *Normalized DCG (NDCG)*. They also make the important point that "in ranking, it is preferable to have fewer errors at higher ranked positions than at the lower ranked positions, which is reflected in many metrics".

3.2 Graph Networks

The most basic type of neural network, consisting solely of fully connected layers, converts a vector of fixed size into another vector of fixed size. In contast, convolutional neural networks (CNNs) can convert variably-sized input into variably-sized output. In image classification or segmentation tasks, the inputs are often scaled to a fixed size and the convolutional layers serve as feature extractors for a fully-connected classification head. Recurrent neural networks can convert a sequence of vectors into another sequence.

While some of those neural network types *can* handle variably-sized inputs, they do not deal with sets very well: Consider for instance an (unordered) set of images and the task to assign a single label to the entire set. All architectures from above could be applied to that task by simply concatenating all the images into one large vector. However, they would implicitly try to assign a semantic meaning to the ordering in which the samples are being presented to them (relational inductive bias). One could augment the dataset by shuffling the samples of a set before concatenating them but that does not solve the underlying problem.

Graph networks are designed to handle graphs and sets naturally. They can

deal with any size of graph and do not regard the ordering of nodes. [Bat+18] have brought several different types of graph networks under a common hood. We follow their notation and will replicate the relevant parts in the following paragraphs. That is the formal definition of a directed, attributed multi-graph with a global attribute, and of graph networks.

Let a graph G be a 3-tuple $G = (\boldsymbol{u}, \mathbb{V}, \mathbb{E})$, where $\boldsymbol{u}$ is a vector of global attributes. $\mathbb{V} = \{\boldsymbol{v}_i\}_{i=1}^{N^{(v)}}$ is a set of $N^{(v)}$ nodes (also referred to as vertices) consisting solely of an attribute vector. $\mathbb{E} = \{(\boldsymbol{e}_k, r_k, s_k)\}_{k=1}^{N^{(e)}}$ is a set of $N^{(e)}$ edges, each of which connects a sender node $\boldsymbol{v}_{s_k}$ to a receiver node $\boldsymbol{v}_{r_k}$, and contains an attribute vector $\boldsymbol{e}_k$.

The attribute vectors ($\boldsymbol{u}$, $\boldsymbol{v}_i$, and $\boldsymbol{e}_k$) can contain any kind of information. In fact they could even be graphs themselves.

Graph networks (GN) are neural networks that contain at least one GN block. [Bat+18] describe it as follows: A GN block contains three update functions, ϕ, and three aggregation function, ρ,

$$\boldsymbol{e}'_k = \phi^e\left(\boldsymbol{e}_k, \boldsymbol{v}_{r_k}, \boldsymbol{v}_{s_k}, \boldsymbol{u}\right) \tag{2}$$
$$\overline{\boldsymbol{e}}'_i = \rho^{e\to v}\left(\mathbb{E}'_i\right) \tag{3}$$
$$\boldsymbol{v}'_i = \phi^v\left(\overline{\boldsymbol{e}}'_i, \boldsymbol{v}_i, \boldsymbol{u}\right) \tag{4}$$
$$\overline{\boldsymbol{e}}' = \rho^{e\to u}\left(\mathbb{E}'\right) \tag{5}$$
$$\overline{\boldsymbol{v}}' = \rho^{v\to u}\left(\mathbb{V}'\right) \tag{6}$$
$$\boldsymbol{u}' = \phi^u\left(\overline{\boldsymbol{e}}', \overline{\boldsymbol{v}}, \boldsymbol{u}\right), \tag{7}$$

where $\mathbb{E}'_i = \{(\boldsymbol{e}'_k, r_k, s_k)\}_{r_k=i,k=1}^{N^{(e)}}$ is the set of all edges pointing to the ith node, $\mathbb{V}' = \{\boldsymbol{v}'_i\}_{i=1}^{N^{(v)}}$, and $\mathbb{E}' = \bigcup_i \mathbb{E}'_i$.

1. ϕ^e is the **edge update function** and applied to every edge in the graph. For the kth edge it computes a new attribute vector $\boldsymbol{e}'_k$ based on the previous state $\boldsymbol{e}_k$, the attributes of the two nodes that this edge connected (i.e. $\boldsymbol{v}_{r_k}$ and $\boldsymbol{v}_{s_k}$), and the global state $\boldsymbol{u}$.

2. $\rho^{e\to v}$ is the **edge aggregation function** and applied once for every node in the graph. It computes an aggregation of all edges pointing to the ith node.

3. ϕ^v is the **node update function**. It is applied to every node in the graph and computes the new attribute vector $\boldsymbol{v}'_i$ based on the aggregation of all

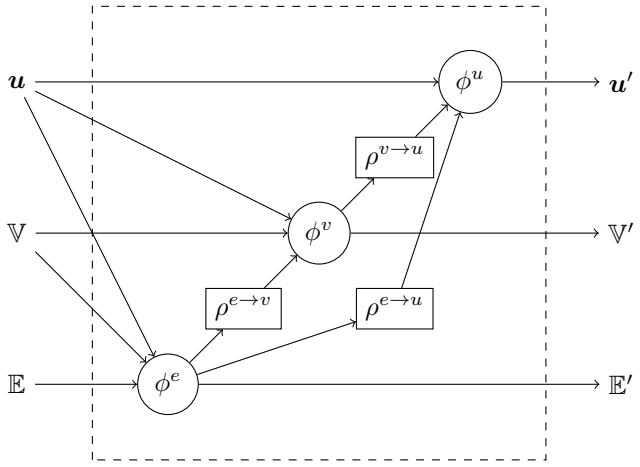


Figure 1: Visualization of the evaluation order in a **full GN block**. Arrows indicate inputs as well as temporal dependency. The three inputs to a GN block are on the left, the three outputs (transformations of the inputs) are on the right-hand side. Several variation of the depicted version exist. [Bat+18] show the full block, as well as the variations independent recurrent block, message-passing neural network, non-local neural network, relation network, and deep set in Figure 4.

edges pointing to the node, the previous state of the node, and the global state.

4. $\rho^{e\to u}$ is the **global edge aggregation function** which computes an aggregation $\overline{\boldsymbol{e}}'$ of all edges in the graph.

5. $\rho^{v\to u}$ is the **node aggregation function** which compuates an aggregation $\overline{\boldsymbol{v}}'$ of all nodes in the graph.

6. ϕ^u is the **global attribute update function**. It takes both node and edge aggregation as well as the previous global attribute $\boldsymbol{u}$ and computes a new global attribute $\boldsymbol{u}'$.

The functions are being evaluated in a specific sequence, visualized in Figure 1. GN blocks do not alter the structure of the data, i.e. they convert a graph into another graph. In order to extract a fixed size representation from a GN block, the global attribute can be used.

Graph networks in related forms have been applied to a variety of different problems from different domains:

- **Social sciences**. [HYL17] apply graph networks to post data from the social network Reddit. Like we do, [KW16] use convolutional graph networks. They run experiments on citation networks and on a knowledge graph dataset.
- **Physics**. [San+18] and [Bat+16] train graph networks that focus on the modelling of physical relationships between objects.
- **Medicine**. [Fou+17] consider the prediction of interfaces between proteins, a challenging problem with important applications in drug discovery and design.
- **Algorithms**. See e.g. [Sel+18] and [Dai+17].

Graph networks can be trained with stochastic gradient descent (SGD), since they are fully differentiable, if aggregation and attribute update functions are differentiable. [Sel+18] train a graph network on a boolean satisfiability task (SAT-solver) with a single bit of supervision, namely whether or not the statement is satisfiable. They thereby show that graph networks can be successfully trained on NP-complete tasks with very little supervision, namely just a single bit.

They extract the fixed-size information (true or false) from the graph network by computing the mean of all nodes. This fits into [Bat+18] Deep Mind's graph networks framework, since the output bit can be interpreted as a global state. Depending on the application, a model's output might be a graph itself, in such cases, the last layer can be a GN block. !!! extend with set2vec and explain the problem on a more general level

3.3 Screenshot Processing

Convolutional neural networks (CNNs) were originally inspired by the visual cortex of humans, see [LeC+98], and had their most notable breakthrough with the success of AlexNet ([KSH12]) in the domain of image classification. The convolutional layers are sliding multiple kernels over two dimensions of the input while looking at all channels simultaneously. This concept allows for two-dimensional translation invariant feature extraction and has proven to be both effective and efficient.

Most commonly, CNNs are applied to natual images, for instance pictures of animals taken in nature or car traffic scenes from a driver's point of view. Screen-

shots, i.e. images that show the content of a monitor, differ from natural images: Often they do not contain smooth transitions, do not feature a large color variety, and contain many equally colored areas such as white background. CNNs have been shown to have an inductive bias towards natural images, see e.g. [UVL17] who exploit this bias in denoising, super-resolution, and inpainting problems. The question arises whether CNNs are also applicable to screenshots, which have different characteristics. There is little work that uses CNNs for screenshots processing, which can either be attributed to them handling screenshots just as fine as natural images or few applications.

The closest to us is the work of [Bel17], who feeds screenshots into a CNN in order to extract structual information from them. He tries to convert screenshots of a user interface into code that describes that exact layout. In his setup, a CNN is combined with an LSTM: The former is responsible for the feature extraction, whereas the latter converts the extracted features into a sequence of layout descriptions.

More specifically, he does not perform any pre-processing and feeds images of size 256×256 into the model. The model consists of three convolutional blocks, each of which contains two convolutional layers with kernels of size 3×3 and stride 1, followed by 2×2 max-pooling and dropout with a probability of $p = 0.25$. The third convolutional block is followed by two fully connected layers with 1024 units each. Both regularized with $p = 0.3$ dropout. The filter count for the convolutions is 32, 64, and 128 for the three blocks, respectively.

We choose to use CNNs as feature extractors for the website screenshots as well. Our baseline architecture is a slightly adjusted version of [Bel17], with more agressive pooling to make it work for screenshots with larger resolution.

3.4 Ground Truth

3.4.1 Open PageRank

Open PageRank [Dom19] is an initiative, which has the goal to enable webmasters to determine the pagerank for their domains and easily compare with other domains. The initiative was constituted after Google had shut down the *PageRank*-toolbar, leaving a void in the industry. The Open PageRank initiative provides freely available data that was built on top of *Common Crawl* [do/19], which provides high quality crawl data of webpages since 2013.

Open PageRank uses the number of backlinks of a domain found in *Common Crawl* to calculate the pagerank. The greater the number of links, different domains use to refer to the given domain, the better the pagerank of the given domain according to Open PageRank.

The list offered by Open PageRank contains about 10 million domains ordered by their page rank. Each entry in the list consists of following comma-separated attributes:

- rank of the domain
- domain name
- calculated score by Open PageRank

The list can be downloaded at `https://www.domcop.com/openpagerank/`.

3.4.2 Google Chrome and Chromium

3.4.3 Puppeeter

3.4.4 Chromium Embedded Framework

3.4.5 Fully Qualified Domain Names

3.5 Kubernetes

This section is taken from the project work *Evaluation and Extension of Current Automated Infrastructure Deployment Solutions for Container-based Clusters* by Güner (2019).

Kubernetes (K8s) is an open source container orchestration solution that was originally designed by *Google* in 2014. It aims to provide a platform where containers and containerized applications can be deployed, scaled, and managed [con18]. K8s orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This means applications are automatically scaled depending on incoming workload.

K8s is much of the simplicity of Platform as a Service with the flexibility of Infrastructure as a Service [Kub18]. K8s offers platform abstraction at container level. Therefore, developers are responsible for building their own application

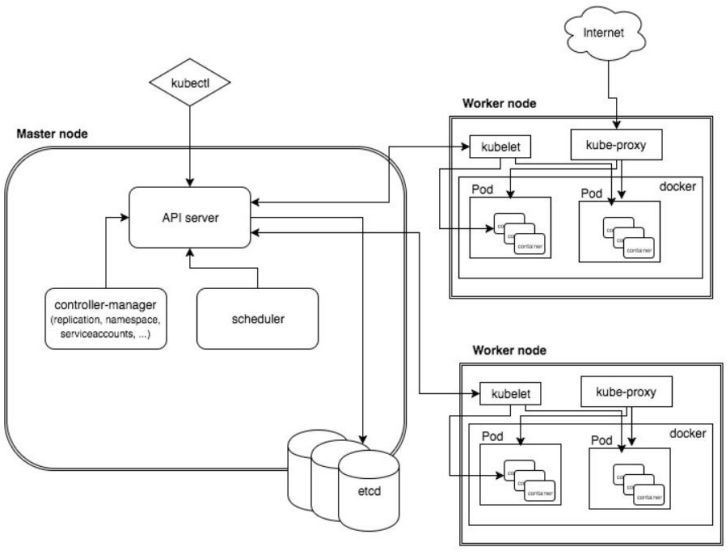


Figure 2: Architecture Diagram of K8s [tea18]

containers rather than only the application itself. This is rewarded with more freedom in application development, but comes at the cost of lower productivity.

3.5.1 Kubernetes Components and Concepts

The most popular container runtime in the industry is *Docker*. Container runtimes are responsible for serving the communication to the OS and the container environment, where applications can be configured and ran. K8s consist of three main components *kubectl*, *master node*, and the *worker nodes*. Kubectl represents the client interface to communicate with the master node, by sending requests against the API server. The master node represents the administrative entrypoint and is responsible for managing the whole K8s cluster [tea18]. It consists of the *API server*, *etcd storage*, *scheduler*, and *controller manager*:

- *API server* represents the entry points for all incoming REST commands. It processes and executes them according to their business logic.
- *etcd storage* is a simple key-value store, where information about the *pods* and the cluster is saved.
- *scheduler* is responsible for container distribution among the worker nodes by using information about the worker nodes.

- *controller-manager* consists of several controllers such as the replication controller, which is responsible for monitoring pods and recreating crashed pods.

Beside the master node, a K8s cluster consists of one or more worker nodes, which are controller by the controllers of the master node. Each worker node consists of *kubelet*, *kubeproxy*, and *pods*:

- *kubelet* communicates with the API server, ensuring the requested containers are up and running.
- *kube-proxy* acts as a proxy and load balancer on the worker node. It is responsible for the communication with the outside world.
- *pods* is the smallest unit in K8s. They encapsulate one or many other containers in the same shared context, which means that containers within the same pods share the same IP, storage, and memory.

4 Method

Our contributions, novel aspects, great detail

5 Datasets

In our approach we propose a supervised machine learning algorithm to estimate the page rank using deep graph networks. For this purpose we require labeled data to develop, train, and validate our model. Each dataset version was created by our Datacrawler (7) to answer a specific question raised during our research.

All dataset versions have OpenPage Rank (3.4.1) as ground truth. This means that we use the ordered list of domains from OpenPage Rank in the datacrawler to enrich them with more information thus to generate our datasets.

The following sections will profoundly discuss each dataset and the questions addressed with them.

5.1 Dataset Version 1

The goal of the dataset version 1 was to answer the question on how well CNNs perform on estimating a page rank of a website. The provided data was used to train and evaluate the performance of a CNN on page rank estimation.

The dataset version 1 consists of 100,000 samples in total. Every sample X corresponds to a domain x contained in the ground truth. We denote the r^{th} sample as $x^{(r)}$ in the ground truth and $X^{(r)}$ in the dataset. A single sample in the ground truth can be expressed as a tuple consisting of two values:

$$x^{(r)} = (r, u)$$

- r The global rank of the domain, according to Open PageRank (3.4.1), within $[1, 100000]$.
- u The URL of the domain. For example a string like `github.com`.

A single sample in the dataset can be expressed as a tuple consisting of three values:

$$X^{(r)} = (\boldsymbol{I}, r, u)$$

- $\boldsymbol{I}$ The screenshot of the domain in image format PNG and in the resolution 1920×1080. The screenshot is generated by the datacrawler for the given domain corresponding to the ground truth. For ensuring repeatability,

only the upper visible area of the website has been taken into account, which is provided during the initial loading of the website.

r The global rank of the domain, according to Open PageRank, within $[1, 100000]$.

u The URL of the domain mapped by the Open PageRank list. For example a string like `github.com`.

5.2 Dataset Version 2

5.2.1 Overview

The goal of the dataset version 2 is to enable the evaluation of deep graph networks for page rank estimation. The provided data was used to train and afterwards to evaluate deep graph networks in experiments. It was generated by our Datacrawler (7).

There are in total 100,000 samples in dataset version 2. Like dataset version 1 (5.1), every sample X corresponds to a domain x contained in the ground truth and is generated by the Datacrawler (7). Moreover we denote the r^{th} sample as $x^{(r)}$ in the ground truth and $X^{(r)}$ in the dataset as well. We define a single sample in the dataset version 2 as a tuple consisting of three values:

$$X^{(r)} = (G, u, r)$$

G The directed graph of the domain characterized by a set of nodes $\mathbb{V}$ and edges $\mathbb{E}$: $G = (\mathbb{V}, \mathbb{A})$. The directed graph G (5.2.2) is generated by the datacrawler for the given domain corresponding to our ground truth.

u The URL of the domain mapped by the Open PageRank list.

r The global rank of the domain, according to Open PageRank, within $[1, 100000]$.

5.2.2 The Directed Graph G

The directed graph provides a topological view on the given website by representing all associated web pages within the same domain as nodes and connections as arrows. Nodes and arrows of the graph are enriched with local

information about the web pages. Figure 3 exemplary illustrates the partial directed graph for the domain `timodenk.com` with three webpages represented as nodes with corresponding attributes and edges.

The directed graph G is characterized by a set of nodes $\mathbb{V}$ and edges $\mathbb{A}$. We limit the number of nodes in a graph to 8, since we are not able to express a website in its entirety as a graph. Each node $v \in \mathbb{V}$ represents a web page of the given domain associated with the graph. A node is characterized by the following tuple:

$$v = (u, \boldsymbol{I}, \boldsymbol{M}, t, l, s, h)$$

- u The exact URL of the web page. Example: `https://github.com/help`
- $\boldsymbol{I}$ The screenshot of web page in image format JPEG and in the resolution 1920×1080.
- $\boldsymbol{M}$ The screenshot of web page in image format JPEG and in resolution 375×667, which represents a screenshot taken from a mobile devices.
- t The title of web page as can be seen in the tabs bar of a common browser.
- l The time it takes to download all data of web page from the web server in milliseconds.
- s Number of kilobytes downloaded in total for web page (size).
- h Indicator for whether or not the website uses the protocol HTTPS, the value is $\in \{\text{true}, \text{false}\}$.

Each edge a represents a possible navigation from web page v_1 to v_2 in the graph G. It can be understood as a hyperlink or button found on web page v_1 (source), which points to v_2 (target) and has the same Fully Qualified Domain Name (3.4.5).

$$a \in (v_1, v_2, t)$$

- $v_1 \in \mathbb{V}$ The source node (web page).
- $v_2 \in \mathbb{V}$ The target node that v_1 points to.
- t If existent, the text that links from source to target web page.

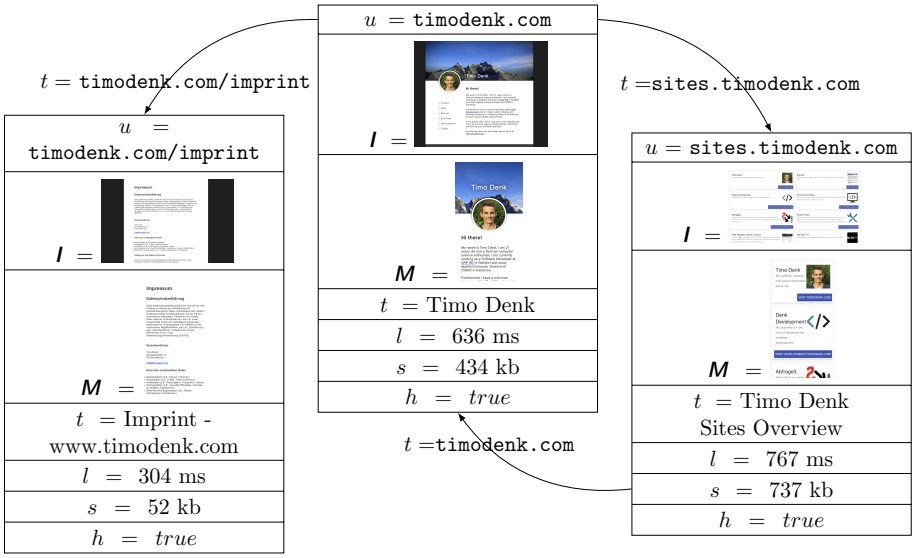


Figure 3: The figure illustrates the partial directed graph $G = (\mathbb{V}, \mathbb{A})$ of the domain `timodenk.com` with three nodes representing the web pages `timodenk.com`, `timodenk.com/imprint`, and `sites.timodenk.com`. Each webpage is a node $v \in \mathbb{V}$ represented as a box containing the tuple of attributes $v = (u, \boldsymbol{I}, \boldsymbol{M}, t, l, s, h)$ as specified in 5.2.2. Each attribute is represented as an entry in the box and contains respective values of the webpage. For example the webpage `timodenk.com` contains a screenshot $\boldsymbol{I}$ and screenshot in mobile format $\boldsymbol{M}$. Furthermore it has the title t "Timo Denk", a loading time l of 636ms, total download size s of 434kb and uses the HTTPS protocol as the value *true* indicates for h. Nodes are connected with directed edges $a \in \mathbb{A}$ represented by the arrows in the figure. Moreover, the arrows are annotated exemplary representing the text t that links from the source node $v_1 \in \mathbb{V}$ to the target node $v_2 \in \mathbb{V}$ as specified in 5.2.2. For example the node of the webpage `timodenk.com` has an edge $a \in \mathbb{A}$, which points to the node $v_2 \in \mathbb{V}$ of the webpage`sites.timodenk.com` and has the text t `sites.timodenk.com`.

6 Implementation Details

Technical aspects of our methods, things we have tried that might have failed, code snippets Technical description of the dataset creation process

7 Datacrawler

The main goal of the datacrawler is to deliver the required datasets as specified in chapter 5. The input to the datacrawler is simply a domain, which is provided by the ground truth. Depending on the configuration the output of the datacrawler can vary from a single screenshot (5.1) to a complete graph of the given domain (5.2).

The following section (7.1) will profoundly discuss the requirements in functionality for the datacrawler, which will serve as a basis for discussion for the decision between the frameworks `Puppeeter` and `Chromium Embedded Framework` (7.2). Afterwards we will give an overview of the architecture (7.3) and highlight the idea of *DataModules* in the crawler (7.3.2). Following that, we will discuss profoundly all developed *DataModules* with their role in the datacrawler (7.3.3, 7.3.4, 7.3.5).

In the last section (7.5) we will discuss how we successfully designed a sophisticated system to scale-out the datacrawler reducing the total dataset creation time on Google Cloud Platform.

7.1 Requirements

The requirements in functionality for the datacrawler arise from the dataset specifications (5.1, 5.2). The next section will derive the requirements browser emulation (7.1.1), information accessibility (7.1.2), modularity (7.1.3) and scalability (7.1.4) from the dataset specifications and will discuss them profoundly.

7.1.1 Browser Emulation

One of the major requirements for the datacrawler is to find a convenient way of emulating a web browser. According to the dataset specifications, for every given website a screenshot $\boldsymbol{I}$ has to be taken. In addition to that, every screenshot must represent the website as the user would see in a common browser. The latter and other attributes such as the loading time l of a website like in a common browser make the emulation of a browser an inevitable requirement.

7.1.2 Information Accessibility

Accessing *low-level* information such as the *HTTP-Request* to change the *user-agent* to generate a mobile screenshot $\boldsymbol{M}$ (7.3.4) or the *Document Object Model (DOM)* (7.3.5) to generate edges a in the graph is inevitable. The datacrawler has to be able to manipulate internal data structures and even able to inject own *JavaScript-code* (7.3.3) on the website.

7.1.3 Modularity

The datacrawler has to be as modular for allowing us to extend the datacrawler easily and decide which attribute should be calculated. This requirement in flexibility is raised from the fact that our dataset specifications might change over the time or new ones might be added.

This flexibility led to a sophisticated architecture such as the DataModule-System (7.3.2), which makes it possible to pass the dataset specification directly into the datacrawler.

7.1.4 Scalability

Both dataset versions require the analysis of at least 100,000 websites. Therefore, datacrawler has to be horizontally scalable to allow the analysis of multiple domains at once, and efficient in terms analysis time and start-up time. An inefficient and not scalable datacrawler would lead to multiple days of required analysis time and reflect also in high infrastructure costs.

7.2 Framework

The previous section has shown some of the intricate requirements for the datacrawler. To answer those requirements in the given time frame, we investigated in frameworks, which would do most of the heavy-lifting for us such as networking, I/O or rendering of the website.

In our research we focused on the evaluation of frameworks, which base on the most common browser `Google Chrome` (3.4.2) [w3s19]. This section will introduce two frameworks `Puppeteer` and `Chromium Embedded Framework` (`CEF`), compare both and discuss our decision for `CEF`. Both frameworks are providing an API to instrument the well-known browser `Chrome`.

7.2.1 Puppeteer

`Puppeteer` is a open-source library for the JavaScript runtime `Node.js`, which provides a high-level API to control instances of the browsers `Chrome` or `Chromium`. One of the main goals of `Puppeteer` is to grow adoption in automated browser testing [Dev19]. For this purpose, it wraps the `Chrome DevTools Protocol` in JavaScript, which allows web developers to instrument, inspect and debug instances of `Chrome`. Technically, the protocol is HTTP-based and exposed as a RESTful API at the port for debugging by `Chrome`.

The protocol offers a varies number of functionalities, which are grouped into domains [Chr19a]. Some of the interesting domains for our use-case are:

Page : API to load and take a screenshot of the given website.

DOM : API to read and manipulate the DOM of the given website.

Network : API to intercept network requests, track downloaded data and network issues.

Emulation : API to emulate different geolocation, network bandwith and mobile devices.

During start up `Puppeteer` starts an instance of `Chrome`, attaches to the instances using the port for debugging and afterwards custom code of the user will be executed against the instance. The instance will be started in `headless`-mode meaning that no UI will be visible.

`Puppeteer` fulfills all requirements (7.1) raised from the dataset specifications: The requirement of **Browser Emulation** (7.1.1), since it is using an instance of `Chrome` to render websites. The requirement of **Information Accessibility** (7.1.2), due to functionalities in the domains DOM and Network. Furthermore **Modularity** (7.1.3) is given per se by using JavaScript as the programming language, which makes the use of polymorphism possible as used in DataModule-System (7.3.2). The usage of `Chrome` ensures efficiency in terms of loading, rendering websites and retrieval of website information. Moreover, the combination of `Chrome` and `Puppeteer` can be scaled-out easily fulfilling the last requirement **Scalability** (7.1.4) by using our system described in 7.5.

7.2.2 Chromium Embedded Framework

Chromium Embedded Framework (CEF) is an open-source framework for embedding the Chromium browser into other applications. The main goal of CEF is to enable developers in adding web browsing functionality such as using HTML, CSS and JavaScript to create application UI. Well-known applications such as Adobe Acrobat, Spotify Desktop and MATLAB are using CEF.

CEF was designed ground-up with performance and ease-use in mind. The framework exposes C++ interfaces with default implementation for all features requiring little or no integration work. Furthermore, the community added wrappers for the base implementation to support a wide-range of operating systems and programming languages.

CEF has in total three versions, whereas CEF 2 was abandoned due to the appearance of the Chromium Content API which will be discussed later. CEF 1 is a single process implementation and based on the old Chromium WebKit API, due to deprecation of the Chromium WebKit API it is no longer supported and developed. CEF 3 is a multi process implementation and based on the current Chromium Content API.

As the Chromium code base has grown, it was inevitable to avoid opaque dependencies and features in the wrong places. Therefore the *content module* was introduced, which contains the core functionality of Chromium. The Chromium Content API wraps the content module and offers isolation for developers from the core functionalities [Chr19b]. CEF 3 insulates the user from the underlying complexity of Chromium by using the Chromium Content API and offering production-quality stable APIs [Fra19]. The offered and consumed APIs as well as the multi-process architecture will be profoundly discussed in **Datacrawler Architecture** (7.3).

Through the use of Chromium Content API, CEF 3 provides a close integration between the browser and the host application including support for custom JavaScript objects and JavaScript extensions. Moreover, the host application is able to control resource loading, intercept the network, navigation and many more, while taking advantage of the same performance and technologies available in the Google Chrome Web browser [Fra19].

CEF 3 clearly fulfills the requirement of **Browser Emulation** (7.1.1) and **Information Accessibility** (7.1.2). Moreover, **Modularity** (7.1.3) and **Scalability** (7.1.4) is also given by using C++ and the core functionalities of Chromium.

Requirement	CEF	Puppeeter
Browser Emulation	✓	✓
Information Accesibility	✓	✓
Modularity	✓	✓
Scalability	✓	✓

Table 1: The table shows that both Puppeeter and CEF satisfy the framework requirements in browser emulation, information accessibility, modularity and scalability.

7.2.3 Framework Decision

Both Puppeteer and CEF fulfill the requirements raised by the dataset specifications as illustrated in table 1. At first glance it seemed a free choice, but after close investigation we found additional points speaking for CEF.

In the end we decided to use CEF 3 for the datacrawler, due to following additional points:

- Our team has an outstanding expertise in C++, which is the programming language used in both CEF 3 and Chromium.
- While as Puppeteer is limited to the Chrome DevTools Protocol, CEF 3 provides low-level access to the core of Chromium. This ultimately gives us more freedom of control.
- CEF 3 is integrated into Chromium leading to a faster start-up and execution time Whereas Puppeteer has to start up an instance of Chrome and communicate via a RESTful HTTP API, which represents an addition overhead.

7.3 Datacrawler Architecture

One of the major points in the design and implementation of the Datacrawler was to meet the **Modularity** requirement (7.1.3). This led to the development of the sophisticated *DataModule*-System and introduced the concept of *Datamodules*. The Datamodule-System represents the architecture of the Datacrawler.

Before diving into our implementation details by introducing the *DataModule*-System, we will discuss how applications using CEF 3 are build and work

in general (7.3.1). Following that, we will introduce the *DataModule*-System (7.3.2) in great detail giving the required information to understand how *Datamodules* work. Afterwards we will discuss how we take a screenshot of a website and tackle the challenge of detecting when a website has finished loading in the *Screenshot*-Datamodule (7.3.3). Beyond that we will discuss how we take screenshots by emulating a mobile device (7.3.4) and collect URLs by traversing the DOM (7.3.5). In the end we will show how we generate and output the graph (7.3.6) and wrap up by taking a look into the end-to-end workflow of the Datacrawler (**??**).

7.3.1 CEF 3 Application Structure

`CEF 3` is based on multiple processes. It can mainly be divided into the *browser*-process and *render*-process.

- The browser-process represents the host process of the application, which handles window creation, painting and network access. Furthermore, most of the application logic will run in the browser-process.
- Multiple render-processes are responsible for rendering websites, executing JavaScript and running some application logic. Following the process model of `Chrome`, `CEF 3` spawns for every unique origin a render-process ensuring resource isolation, parallelism and better failure management.

The separate spawned processes communicate using *Inter-Process Communication* (IPC). Application logic implemented in browser- and render-process can communicate by sending asynchronous messages as **URL-Datamodule** (7.3.5) will show. Other processes are spawned when needed such as the *plugin*-process for handling of plugins like *Flash*.

In general `CEF 3` application consists of the class `CefApp` and `CefClient`. `CefApp` is responsible for process-specific callbacks such as returning handler for a custom render-process implementation (7.3.5). Whereas `CefClient` is responsible for handling browser-instance-specific callbacks such as returning handler for a customer website render implementation (7.3.3). It contains most of the application logic and is being used to create browser instances. As mentioned before, it can be used to control the browser-instance with custom implementation.

The following will briefly describe the start-up of an application using `CEF 3`:

1. The `CefExecuteProcess`-method is used with custom implementation of `CefApp` to start separate processes. The application executable will be started multiple times representing new processes.

2. `CreateBrowser` is used with custom implementation of `CefClient` to create a browser-instance. The browser instance is created in the browser-process, which means that there is only one browser at a time. Futhermore, an initial URL is passed to the browser-instance.

3. In the implementation of `CefClient` `CefMessageLoop`-method is executed, which starts the event loop of the browser-instance. This means that browser-instance will start loading the given URL, handling network, rendering and many more. `CEF` takes care of spawning and using of existing render-processes.

4. Once `CefQuitMessageLoop`-method is called, the message loop will be quit.

7.3.2 DataModule-System

The naming of the DataModule-System derives from the purpose to bind, execute instances of Datamodules, collect results and output the dataset. The system consists of several components, which enable developers to implement and execute Datamodules independently increasing flexibility and reducing potential failures. In a plugin-and-play-manner developers can easily develop and add Datamodules to the Datacrawler.

The main idea of a Datamodule is to represents a single attribute in the dataset, which is calculated and added independently from other attributes to the dataset. In addition, each Datamodule can be individually configured and turned on or off by the user.

We are heavily utilizing polymorphism language feature of `C++` to enable previous mentioned features. Thus, a Datamodule consists of the implementation of the following interfaces:

`DataModuleBaseConfiguration` This interface has to be implemented by each Datamodule. It is responsible for the creation and configuration of instances of a given Datamodule.

`DataModuleBase` The DatamoduleBase-interface represents the core of each Datamodule. Most of application logic of the Datamodule will be implemented here.

`DataBase` This has to be implemented if the result of the Datamodule should be stored in the graph. It represents the results of the implemented Datamodule.

Figure 5 represents a simplified UML-Diagram of the Datacrawler with the most important classes, interfaces and methods for understandig the DataModule-system. The figure does not show the implementation details of `CEF 3`, which are isolated and located in each Datamodule.

The class `Datacrawler` represents the entrypoint of the Datacrawler. It holds an instance of the class `DatacrawlerConfiguration`, which is responsible to load the user-defined Datamodules configuration using the method `loadConfigFromJSON()`. Upon successful loading, the Datacrawler creates instances of Datamodules using the method `createInstance()`. The interface method is implemented individually by each Datamodule reflecting custom Datamodule configuration.

Once the `process(url)`-method has been called with an URL in `Datacrawler`, we create `NodeElement` representing the starting node and add it into our graph. Our graph is implemented as a key-value list, whereas the key represents the URL and the value our `NodeElement`-object. For dataset version 1 (5.1) we assume to have a single node with a single screenshot. Afterwards we go through the our instantiated Datamodules and execute custom implementation of the method `process(url)`. At this point an isolated browser instance is created using `CEF 3`, which executes Datamodule-specific browser implementation. As soon as a Datamodule returns, we add the results to the current `NodeElement` and add `NodeElement` to our graph.

Upon the completion of all Datamodules for a given URL, we add all generated edges of the Datamodule `URLDatamodule` into a FIFO queue in `Datacrawler`. In section 7.3.5 we will discuss how and which URLs we select as edges of a node. Afterwards we repeat the process of creating a new object of `NodeElement`, executing of Datamodules, adding the results into new `NodeElement` and URLs to our FIFO queue for the oldest entry. We skip a given URL if there is a key-value pair in our list or if we reach our maximal number of nodes in the graph.

The use of the FIFO queue leads to breadth-first search in our graph in opposition to a deep search, which is implemented using a LIFO queue. The breadth-first search means that we expand all nodes reachable from the edges of our current node and selecting the oldest entry as new node, rather than using the first edge, expanding the reachable node and selecting it as the current

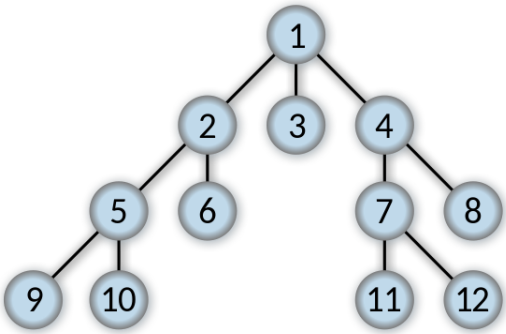


Figure 4: This figure illustrates the breadth-first search. The numbers in the nodes show the order in which nodes are expanded in breadth-first search. It shows that all nodes reachable from a selected nodes are expanded, before selecting the first node encountered and expanding the nodes reachable from it. E.g the selected node is **1** and the nodes **2**, **3** **4** are expanded, before we select **2** and expand the node **5** and **6**.

node. By using the breadth-first search, we maximize the number of directly reachable nodes from a given node. This leads to a wide and dense rather than deep and sparse graph. We perform a breadth-first search because we prefer to have connections between those nodes rather than having long strings originating from a root node. That way we capture at least some of the page-internal linking structure. Figure 4 illustrates the breadth search exemplary.

7.3.3 Screenshot-Datamodule

The Screenshot-Datamodule is responsible for visiting a website by a given URL and taking a screenshot upon successful loading as specified in section 5.1. The entrypoint of the Datamodule is `process(url)`-method in `ScreenshotDatamodule`-class. Upon completion, the Datamodule returns an instance of `Screenshot`, which contains a screenshot and information about width and height of the screenshot.

In general the Screenshot-Datamodule addresses three difficult problems with the help of `CEF 3`:

1. **Any** website given by URL has to be loaded without exceptions.
2. **Every** website has to be rendered in a format to take a screenshot from.

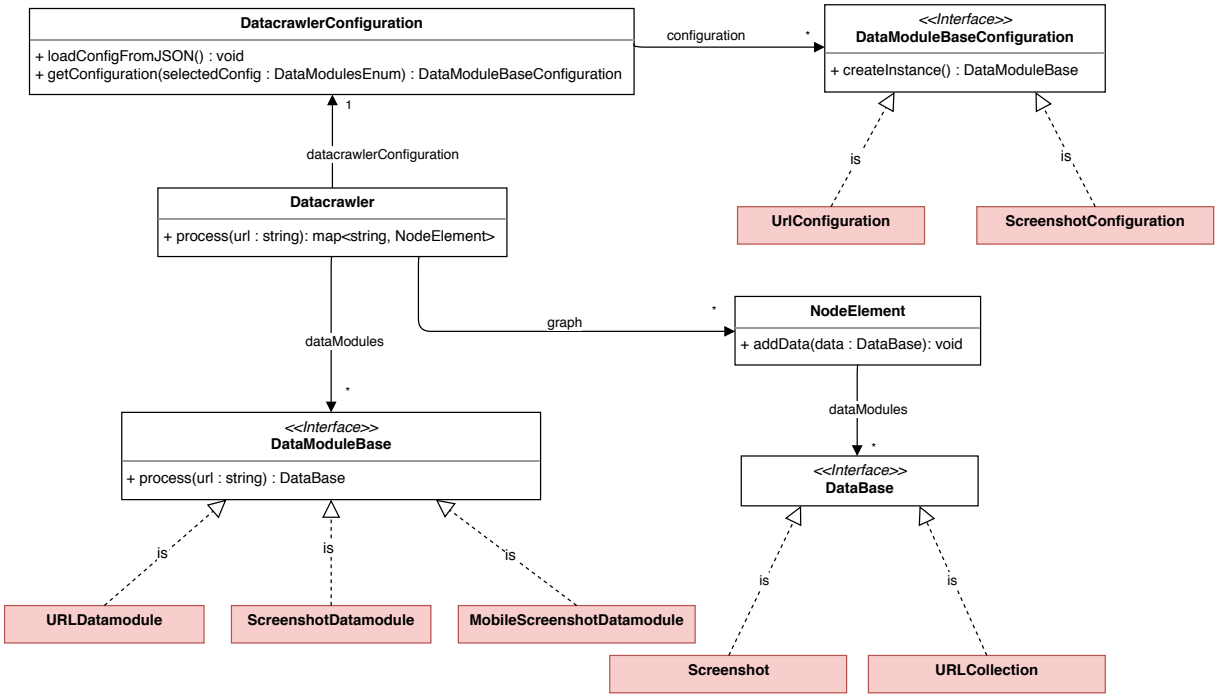


Figure 5: The figure shows the simplified UML-diagram of the datacrawler with the most important classes and methods. Classes in the white boxes represent essential components of the DataModule-System such as the classes `Datacrawler`, `DatacrawlerConfiguration`, `NodeElement` and the interfaces `DataModuleBaseConfiguration`, `DataModulBase` and `DataBase`. Classes in the red boxes represent implementation of DataModules such as the `URLDatamodule` (7.3.5), `ScreenshotDatamodule` (7.3.3) and `MobileScreenshotDatamodule` (7.3.4).

3. The Datamodule has to detect when a given website has finished loading to take a screenshot.

The following sections will profoundly discuss how the Screenshot-Datamodule successfully addresses previous mentioned problems with the use of `CEF 3`.

Loading The Screenshot-Datamodule consists of `ScreenshotDatamodule` and implementations of the interfaces `CefClient` and `CefRenderHandler`. In `process(url)`-method of `ScreenshotDatamodule` a synchronous browser instance is created with our implementation of `CefClient` and the given URL. As mentioned in section 7.3.1 `CefClient` is responsible for browser-instance-specific callbacks and hence the place to customize the browser for our needs. Therefore, we implement `CefClient` and overwrite the method `GetRenderHandler()` to return a custom implementation of `CefRenderHandler`. The browser instance will now use our implementation of `CefRenderHandler` to paint the website during loading.

The use of a synchronous browser instance allows us to iterate through the event loop of the browser instance by ourselves until we take a screenshot of the given website. This is mandatory for both threads, which are created directly after the browser instance and play a critical role in detecting if a website has finished loading.

Rendering In the world of `Chromium` *rendering* relates to the process of parsing, transforming of website data to generate DOM, executing JavaScript, applying CSS and painting of the website.

In Screenshot-Datamodule the interface `CefRenderHandler` is not responsible for rendering a website, but only for the painting to the end user device. In our implementation, we have implemented the `GetViewRect()` and `OnPaint()`-method of `CefRenderHandler`. Latter is the place to paint the website to the end user device, whereas we define in `GetViewRect()` the height and width of the output from the browser. In case of `CEF 3` the `OnPaint()`-method is invoked each time if the browser instance detects invalidation caused by changes in the DOM and CSS. These changes can be caused by a loading website, animations or dynamically changing content due to custom JavaScript code.

In each invocation of `OnPaint()` following information are passed as parameters:

- Reference to the current browser instance.

- Reference to the buffer, where image information are kept about the website.
- Information about the height and width of the given website.

The buffer is an array of respectively one byte encoded single characters, where each character is representing a channel of the color space `BGRA`. Consequently, a single pixel corresponds to four characters in the buffer. The array represents the pixels from the left most to the right most starting from the top of the viewport in repeating order of the color space. Thus, the buffer itself contains only information about visible area as defined in `GetViewRect()`, which reflects the viewport of the browser.

On each invocation we allocate total of $4 * \texttt{width} * \texttt{heights}$ bytes in the memory and copy the content of our buffer into it. Once we detect the website has finished loading, we can access the latest painting of `OnPaint()` from the memory.

Detection In our current implementation the browser instance calls `OnPaint()`-method to update the view port, whenever a change occurs on the website. In practice this means that our implementation might return screenshots of partially loaded websites, if we access to the buffer at a bad time. The simple solution to wait a given time t and take the most recent screenshot is not an option. This static approach would lead to high analysis time per website and violate the scalability requirement (7.1.4). We have to detect, when a given website has finished loading to reduce our analysis time per website.

In general, there are multiple states during the loading of a website, which can be interpreted that a given website has finished loading. Before we discuss our approaches to tackle this complex problem, we define our understanding when a website has finished loading:

A website has finished loading, if and only if all elements are painted on the view port and no new elements are being painted.

Our definition includes the painting and rendering of the given website in the browser and represents the *natural* behavior of a browser. The following definition does not include those aspects:

A website has finished loading, if and only if all the content of the website has been downloaded successfully from the internet.

Based on our assumption we have developed two algorithms for this problem:

- Detection based on calculation of *change* matrices.
- Detection based on `OnLoadingStateChange()`-method of `CEF 3` and hard coded waiting time.

Our algorithm based on calculation of change matrices, assumes that the number of changed pixels per screenshot will dramatically reduce over time. In other words, given the screenshots $I_t, I_{t+1}, I_{t+2} \in \{0, ..., 255\}^{width \times height \times 4}$ taken at times t, $t+1$, and $t+2$ respectively, the total number of changed pixels $|\Delta_{t,t+1}|$ between I_t and I_{t+1} will be less than $|\Delta_{t+1,t+2}|$. Furthermore, we assume that the $|\Delta|$ will converge for $t \to \infty$ meaning that the website has finished loading. The blue line in figure 6 illustrates our assumption exemplary.

In order to calculate the number of changed pixels between two screenshots I_t and I_{t+1}, we have introduced the change matrix $C_{t,t+1} \in \{0, 1\}^{width \times height}$. Each element $c_{i,j} \in C_{t,t+1}$ represents the change state between the pixels $\mathbf{s}^t_{i,j,} \in I_t$ and $\mathbf{s}^{t+1}_{i,j,} \in I_{t+1}$, which in turn are in $\{0, ..., 255\}^4$ representing the color space `BGRA`.

On every invocation of `OnPaint()` we calculate the change matrix $C_{t,t+1}$ on basis of the current screenshot I_{t+1} and the screenshot I_t of the previous invocation. The following applies for each element $c_{i,j} \in C_{t,t+1}$:

$$c_{i,j} = \begin{cases} 1, & \text{if } \mathbf{s}^t_{i,j,} \neq \mathbf{s}^{t+1}_{i,j,} \\ 0, & \text{otherwise} \end{cases}$$

We compute the total number of changes from I_t to I_{t+1} with matrix norm $||C_{t,t+1}||$ and save it in every invocation of `OnPaint()` for later use. In addition, we also calculate the maximum number changes per screenshot with $width \times height$ and call it C_{MAX}. The red line for `nationalgeographic.com` in figure 6 shows that the number of changes is indeed converging, but not stable as expected. We take this into account and calculate on every invocation the average C_{AVG} for the number of changes at $t-n, t-n+1, ..., t$, whereas n is defined in the Datamodule configuration and t represents our current screenshot.

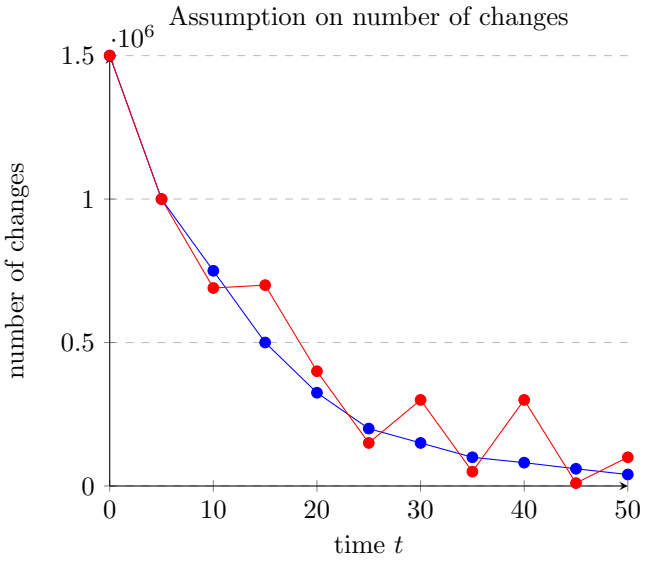


Figure 6: The figure illustrates exemplary in blue our assumption regarding the number of total changes $||C_{t,t+1}||$ from I_t to I_{t+1}. It shows that the number of total changes dramatically reduce and converge for $t \to \infty$. The red line illustrates the actual measured number of changes for `nationalgeographic.com` with the resolution 1920 in width and 1080 in height.

Putting everything together we detect if a given domain has finished and take a screenshot by following equation is true:

$$\frac{C_{\text{AVG}}}{C_{\text{MAX}}} \leq p_C; \quad \frac{C_{\text{AVG}}}{C_{\text{MAX}}}, p_{\text{C}} \in [0, 1]$$

The fraction $\frac{C_{\text{AVG}}}{C_{\text{MAX}}}$ describes the average in percentage of changes over the last n screenshots and p_C is the user-defined threshold.

Furthermore, we have observed that for websites such as `timodenk.com` the `OnPaint()`-method is only called for a limited number of times. This might not be enough to leverage our detection algorithm. Therefore we have implemented a thread, which starts upon every `OnPaint()`-invocation a timer. The timer has the goal to return a screenshot, if `OnPaint()`-method has not been invoked for the user-defined time.

Our second algorithm implements `OnLoadingStateChange()`-method of `CefLoadHandler`, which will be invoked upon a website has been loaded successfully. During invocation the parameter `isLoading` is passed, which is true if the given website

Algorithm based on	Partially Loaded	Fully Loaded	% of Partially Loaded	Total
change matrices	13	87	13 %	100
`OnLoadingStateChange()`	3	97	3 %	100

Table 2: The table illustrates the performance of our algorithms based on change matrices and `OnLoadingStateChange()` for the top 100 websites from `Open PageRank`. The websites are manually evaluated by us. A website is categorized as partially loaded, if it is obviously missing parts of the website, and fully loaded, if not. The approach using `OnLoadingStateChange()` performs with 3% of partially loaded websites better than our approach with change matrices with 13%.

has been loaded and false if not. However, this variable does not indicate that the given website has been painted successfully. It only indicates the browsers is ready to paint the given website. Therefore we have implemented a timer, which will start upon `isLoading` is true, run for a user-defined time and return the content of the buffer.

In addition both algorithms implement a thread, which starts a timer upon the loading of a website. This timer represents a hard timeout for the Screenshot-Datamodule and will run for a user-defined time. After timeout it will ultimately return a screenshort. Thus, guaranteeing us to return the buffer in any circumstances.

We have manually evaluated both approaches using the top 100 website in `Open PageRank` (3.4.1). Our results are described in table 2. They show that the algorithm based on `OnLoadingStateChange()` outperforms the algorithm based on change matrices. This ultimately leads to our decision to use `OnLoadStateChange()` as our preferred detection algorithm.

7.3.4 MobileScreenshot-Datamodule

The MobileScreenshot-Datamodule is responsible for visiting a website by a given URL and taking a screenshot upon successful loading. The `process(url)`-method is the entrypoint for the Datamodule. In contrast to the Screenshot-Datamodule, the visited website has to be in mobile resolution and mobile view leading to a screenshot taken from a mobile device (5.2).

In total there are two ways to request a mobile view for a website:

- Visit the website with mobile device resolution e.g 375 in width and 667

in height

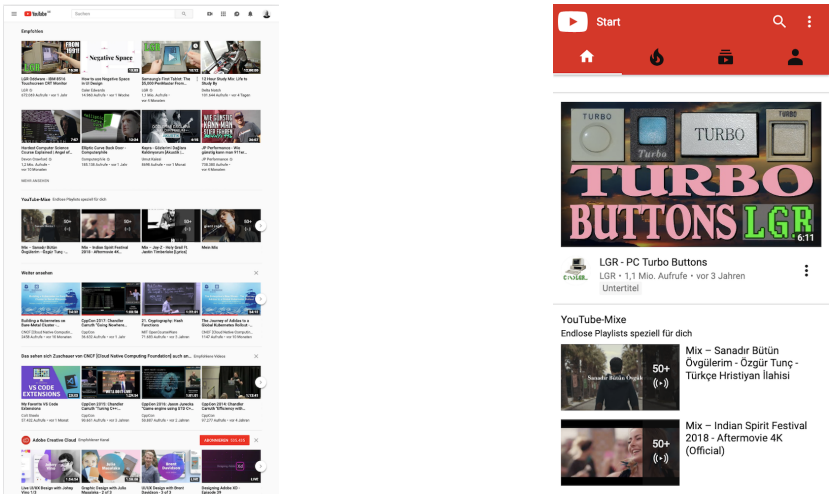
- Declare the *user-agent* as a mobile device e.g `Mozilla/5.0 (iPhone CPU)`

While the resolution represents the size of the view port in which the given website is being displayed, the user-agent represents the identification of the browser. It is a request header containing a characteristic string that allows the communication partner to identify application type, operating system, software vendor and version of the requesting browser instance. The request header is being send on every HTTP method such as `GET`. The `CEF 3` uses a desktop user-agent per default as figure 7a shows.

Our initial assumption, that requesting a website with mobile resolution would lead to a mobile view, was right for a majority of the websites. But figure 7 shows that there are exceptions such as `youtube.com`, where it is not enough to scale down the resolution. Hence, we consider in our implementation to use a mobile resolution and user-agent. While we reuse most of the implementation of the Screenshot-Datamodule (7.3.3) such as the loading, rendering and detection, we extend and change the implementation at specific points to support mobile view.

First we adjust the resolution of the browser instance in `GetViewRect()` to mobile resolution, which is given by the mobile screenshot requirement described in dataset version 2 (5.2). Second, we intercept all outgoing requests from the browser instance and add the required mobile user-agent header. This is done by providing a custom implementation of the interface `CefRequestHandler` having the primary goal to handle browser requests. In `CefRequestHandler` we implement the `OnBeforeResourceLoad()`, which is invoked every time before a request is send. During invocation the outgoing request `CefRequest` is passed as parameter, which allows us to modify the request for our needs. The request contains beside the body, target URL, HTTP method also a key-value store representing the header section of the request. Upon every invocation we modify the key-value store and overwrite the key `User-Agent` with the value `Mozilla/5.0 (iPhone [...]) Chrome/71 [...] Mobile [...]`. The latter represents the Chrome browser on the mobile device `Apple iPhone`.

The interception and modification led us to successfully emulate a mobile device from our Datacrawler, allowing us to take screenshots from a mobile device.



(a) (b)

Figure 7: This figure shows two screenshots taken from the website `youtube.com`. The figures (a) and (b) were taken with a browser instance having the resolution 375×667. In addition, we changed the user-agent of the figure (b) to represent a mobile device and use the default user-agent of *CEF 3*. The usage of the mobile user-agent clearly shows that the website is being displayed for mobile devices in figure (b), whereas in figure (a) not.

7.3.5 URL-Datamodule

The URL-Datamodule represents one of the most complex Datamodules in the Datacrawler. It is responsible for returning a set of valid URLs for a given website, server and client errors occurred during loading, title, loading time and download size of the website. While calculating and returning so many attributes in a single Datamodule is contrary to the principles of the Datamodule-System (7.3.2), we had to do the trade-off in order to (1) reduce overall development and (2) analysis time. The latter arises from the fact that for every single attribute we have to create a new browser instance and reload the entire website, which takes a tremendous amount of computing resources and time. Thus, complexity URL-Datamodule arises by calculating multiple attributes at once and the usage of different processes and their communication based on *IPC*.

In general, the URL-Datamodule is executed by invoking the `process(url)`-method. Upon successful completion it returns an instance of `URLCollection`, which is added to `NodeElement` and contains previous mentioned attributes. Similar to other Datamodules, we implement `CefClient` and other interfaces with our custom implementation and start a browser instance for the given URL. In the following, we will profoundly discuss the implementation by discussing each attribute calculation solely.

URLs The URL-Datamodule has to return valid URLs. This attribute is specified as following in the specification of dataset version 2 (5.2):

> *Each edge a represents a possible navigation from web page v_1 to v_2 [...] and has the same Fully Qualified Domain Name.*

According to the specification a given URL is only valid if it is a possible navigation and has the same FQDN. Possible navigation means that only URLs found as a `href`-attribute in `<a>`-tags in the `HTML`-code of the website can be considered as valid. This follows from the fact that in `HTML` `<a>`-tag is the only tag, which allows the user to navigate to other pages from the current viewed page. While possible navigation describes where to find valid URLs, equality in FQDN describes which URLs to choose from a set of URLs. For example the FQDN for the URL `http://web.samedguener.com/index.html` is `web.samedguener.com`. While the URL `http://web.samedguener.com/contact.html`

found on the website is a valid with `web.samedguener.com` as FQDN, `http://samedguener.com/` is not with `samedguener.com`.

In general, all browsers have a common process to render a website: They download the source code of the requested webpage, parse and build the DOM. The DOM is built and represented as a tree, where nodes are `HTML`-tags and children of a node represent nested `HTML`-tags. In `CEF 3`, we are able to directly access the DOM after a website has finished rendering.

While `isLoading`-parameter of `OnLoadingStateChange()`-method in `CefLoadHandler` (7.3.3) is used for detecting if a given website has finished rendering, the DOM itself can be accessed by implementing `Visit()`-method of `CefDOMVisitor`. Due to the nature of the DOM being built in the render-process, `CefDOMVisitor` has to be run in the render-process conversely to `CefLoadHandler` and all other implementations running in the browser-process. Therefore, we implement `OnMessageReceived()` of `CefRenderProcessHandler` respectively running the the render-process to run an instance of `CefDOMVisitor`. The `OnMessageReceived()`-method is invoked if the render-process receives any IPC-message from any other process. `CefRenderProcessHandler` is responsible for handling render-process-specific implementations. The user can change the default behaviour by implementing this interface. Moreover, we implement also `OnMessageReceived()`-method in `CefClient` to receive IPC-messages from the render-process.

Having all important components together, this leads to the following workflow of the URL-Datamodule:

1. Custom implementation of `CefRenderProcessHandler` is passed to `CefApp` during `CEF 3` initialization.
2. In URL-Datamodule a browser instance is created and started with custom `CefClient` implementation returning the `CefLoadHandler` implementation.
3. Upon successful `OnLoadingStateChange()` in `CefLoadHandler`, send an IPC-message to the render process ultimately invoking `OnMessageReceived()` in `CefRenderHandler`
4. Create instance of custom implemented `CefDOMVisitor` and execute application logic.
5. Upon successful URL gathering, send an IPC-message with collected URLs to browser-process invoking ultimately `OnMessageReceived()` in `CefClient`.

Then, return the results as Datamodule to the Datacrawler.

During the execution of `OnMessageReceived()` in `CefRenderProcessHandler` we use the passed browser instance to execute our implementation of `CefDOMLoader` against it (4). The main application logic is then executed in `Visit(CefDOMDocument)`-method of `CefDOMVisitor`. The passed argument `CefDOMDocument` represents the DOM of the current visited website. Since the DOM is organized as a tree, we use the breadth-first search (4) for tree traversal, check each if given node is a `<a>`-tag and retrieve the URL from the `href`-attribute and the URL text if given. Upon retrieval we filter the URLs having the same FQDN using several regular expressions and create respectively objects of the class `URL`. In addition, we use regular expression to detect whether the given URL has `HTTPS` enabled and enrich the `URL` object with this information. In the end, we return a list of `URL` objects to the URL-Datamodule.

Loading Time `OnLoadingStateChange()` in `CefLoadHandler` is invoked if the website transitions between loading states such as from not being loaded into being loaded or from being loaded into finished loading. To track the loading time of the given website, we just stop the time between both transitions and calculate the delta. We return the loading time back to the URL-Datamodule upon `CefDOMVisitor` has returned the valid URLs.

Client and Server Errors During the loading of a website, the Datacrawler can face client- and server-side errors. On the one hand client-side errors are e.g connection timeout, connection interruption, endless redirection or an invalid TLS certificate. They occur if no successful connection to the website could be established at all. On the other hand server-side errors are returned by the server and redirected through the browser instance to the end-user. They occur if a connection to the server is successful, but the given resource is e.g not accessible. Server-side errors are `HTTP` status codes such as `404` for `resource not found`.

In general client-side errors can be obtained by implementing `OnLoadError()`-method in `CefLoadHandler`. It is invoked if no connection could be established to the server at all. Server-sided errors can be obtained by implementing `OnLoadEnd()`-method in `CefLoadHandler`. Since only client-side error or server-side error can happen at once, only one of the methods is invoked. After invocation we save the client-server or the server-side error and return it to the

URL-Datamodule.

Title The title of the website can be accessed by implementing `OnTitleChange()`-method of `CefDisplayHandler`, which is mainly responsible in handling the display state of the browser e.g browser switching to fullscreen mode. The custom implementation of `CefDisplayHandler` is then passed to `CefClient`. Due to the fact that `OnTitleChange()`-method is invoked everytime the title of a website changes, we only use value of the initial invocation and pass it directly to the URL-Datamodule.

Download Size As specified in the dataset version 2 (5.2) the download size of a website represents the total amount of data downloaded during the loading of a website. In general, the process of loading a website consists of sending requests to a webserver and receiving responses over the internet. In our case, the total number of bytes received in the responses represents the download size of a website.

`CEF` allows us to intercept the raw received responses before they are processed in the render-process of the browser. By implementing `filter()`-method of `CefResponseFilter` we add a *filter* which receives the response beforehand, reads the number of bytes received and forwards the response to the browser-instance. In general, `CefResponseFilter` is returned by an instance of `CefRequestHandler`, which is passed to `CefClient`. Before we return the gathered valid URLs from the URL-Datamodule to the Datacrawler, we convert the download size to kilobytes and return it with the other attributes to the Datacrawler.

7.3.6 Generating the Graph

As specified in dataset version 2 (5.2), for each passed domain to the Datacrawler, we have to output a graph with nodes enriched by information from the Datamodules. Before we delve into the concrete implementation of graph generation process, we will profoundly discuss the output format of the graph, which is later consumed by our machine learning model.

Graph Data Format As section 7.5 will show, we run for each given domain an independent Datacrawler instance. This means that upon successful

completion of the Datacrawler, we will have only printed out exactly one graph belonging to the passed domain. Following this, we create an own folder for the computed domain and name the folder after the rank found in the Open PageRank (3.4.1). The graph itself is represented by a file in the `json`-format and named after the rank in Open PageRank. While most of the dataset information can be found in the `json`-file, we save our screenshots for the domain in the folder called `img`. We differ between normal and mobile screenshots by adding the suffix `_mobile` to the name of the mobile screenshots. Furthermore, we number consecutively each image according an internal ID specified for each node in the `json`-file.

The `json`-file consists of an array containing nodes represented as `json`-objects. Each `json`-object consists mainly of attributes calculated by the URL-Datamodule. Furthermore, each node has an array containing the URLs representing the edges of the given node. We have to emphasize that only URLs has been taken into account, which are also represented as nodes in the graph. Figure 8 exemplary shows the graph for `example.com`.

Process of Graph Generation The process of the graph generation is done in the `Datacrawler`-class and `GraphOutput`-class, which are responsible for the graph calculation and outputting of the graph to the local disk. In general the nodes of the graph are saved in a key-value data structure, where the key is the URL of the computed node and the value represents an object of type `NodeElement` (7.3). `NodeElement` consists of a simple list of the type `DataBase` representing attributes calculated by the Datamodules such as screenshots or URLs.

Upon the invocation of `process(url)`, the Datacrawler creates the initial `NodeElement`, runs all registered Datamodules against the passed domain and adds the results of the Datamodules to the node. Afterwards retrieved URLs of the current node are accessed and put into a working queue for further computation by the Datacrawler. Then, we recursively repeat the process for all elements in the queue until the graph has reached the limit of 8 nodes as specified in dataset version 2 (5.2). Before we hand over the graph to `GraphOutput` for outputting it to local disk, we have to remove edges from the node which direct to no nodes in the graph. This done by going through all the edges of a node and checking whether the given URL as key is mapped to a node. If it is not mapped to a key, we remove the edge.

```
[{
 "id": 1,
 "baseUrl": "https://example.com",
 "client_status" : null,
 "loading_time": 8822,
 "server_status": 200,
 "startNode": true,
 "size": 1222,
 "title" "example.com",
 "urls": [
  {
    "url": "https://example.com/2",
    "url_text": "Second Page"
  },
  ...
  ]
 },
 {
 "id": 2,
 "baseUrl": "https://example.com/2",
 "startNode" : false,
 ...
 }]
```

Figure 8: This figure exemplary illustrates the graph output for the domain `example.com`. The graph nodes are `json`-objects in the root array of the `json`-file. The `baseUrl`-attribute shows affiliation of the node to the given URL. Futhermore, each `json`-object consists of attributes mainly calculated in the URL-Datamodule such as client-sided and server-sided error (here: `client_status` and `server_status`), loading time, size and finally the valid URLs with the URL text. In general, all URLs found in the nodes are represented as nodes in the graph. Finally, each computed URL has an ID assigned, which represents respectively the name of the screenshots for the URL.

Upon the removal of arbitrary edges in the graph, we pass our key-value list to `GraphOutput`. Then for each element of the list, we calculate the `json`-object by going through each element of `Database` in `NodeElement` and generate the attribute entry. Afterwards we add the `json`-object to `json`-array and continue. For the subtype `Screenshot` of `Database`, we use `OpenCV` to output respective screenshots of the current node. We scale down mobile screenshots by factor two and desktop screenshots by factor four. Finally, the screenshots are outputted in `JPEG`-format with `CV_IMWRITE_JPEG_QUALITY` set to 75 out of 100, where the more represents a better quality of the screenshot.

7.4 Processing Time

For scaling the Datacralwer to process the tremendous amount of $100,000$ domains and generate the dataset version 2, we had to do track the average processing time and required disk space for each domain beforehand.

For this purpose we run the final version of the Datacrawler for the following 12 hand-selected domains: `sap.com`, `google.com`, `samedguener.com`, `timodenk.com`, `twitter.com`, `facebook.com`, `nationalgeographic.com`, `youtube.com`, `tagesschau.de`, `cnn.com.tr`, `nytimes.com` and `pornhub.com`

We run the experiment using one Datacrawler instance on a single CPU clocked at 2.3 Ghz on `Ubuntu 16.04`. Furthermore, we provide the Datacrawler with 500 Mb of RAM, enough disk-space for the output and a 100 Mbit internet connection from Karlsruhe.

The experiment has shown that the Datacrawler requires in average about 107 seconds for processing a single domain and in average 244 Kb for the graph. If we estimate those numbers for $100,000$ domains, the dataset version 2 will take about 2972 hours ($\sim$ 123 days) to compute and approx. 25 Gb in disk-space.

Those selected domains have a strong content delivery network, which that means that they can serve the website from any geolocation in a reasonable time. We are aware of this and also strongly assume that Open PageRank (3.4.1) contains domains, which do not have a strong content delivery network. Therefore we take this experiment as a rough estimate to scale the Datacrawler in section 7.5.

7.5 Running the Datacrawler at Scale

The experiment in section 7.4 shows that it is not sustainable to run the processing sequentially on a single Datacrawler instance. This section will profoundly discuss how we successfully managed to dramatically reduce the analysis time to 28 hours by scaling the Datacrawler using Kubernetes on Google Cloud Platform.

The next section (7.5.1) will discuss the general system design we created to scale the Datacrawler. We will focus on how we leveraged `redis` as working queue to distribute domains to be analyzed among the Datacrawler instances and used a network file system to collect the results from all Datacrawler instances. Afterwards in section 7.5.2 we will delve into the Datacrawler container and discuss how consume work items with the Datacrawler from our queue.

7.5.1 System Design

The tremendous processing time measured in section 7.4 comes from the fact that only a single instance of the Datacrawler was used for processing of domains. Fortunately, the problem, the computation of $100,000$ domains, is dividable into smaller independent work items, thus making the Datacrawler horizontally scalable. In our case we define a work item as a single domain, which has to be analyzed by the Datacrawler. From this follows that we run multiple instances of Datacrawlers, each independently analyzing domains and outputting the results to a central storage.

To reduce the total processing time from 2792 hours to 28 hours we have to run in total 100 Datacrawler instances at once. If we keep our Datacrawler resource requirements from section 7.4 as given, we require for this plan at least 50 CPUs, 25 Gb of RAM and 1 Gbit of network bandwidth (10 MBit per Datacrawler). Moreover, we have to take into account the overhead to schedule 100 Datacrawler instances, distribute work among them, collect the results and lifecycle-management. As a result, we decided to use Kubernetes as our workload scheduler, rather than building our own solution. Kubernetes is able to schedule workloads on multiple compute nodes, do lifecycle-management e.g restart crashed applications and automatically scale applications to handle large incoming workloads.

In general, we provision our Kubernetes cluster using Google Kubernetes Engine, which provides managed Kubernetes clusters. Thanks to Google Cloud

Platform our cluster has access to the internal Docker registry, where we provide and push images of our Datacrawler (7.5.2) and `redis`. Those, images are later used by Kubernetes to create containers and run our workload. Before we deploy any instance of Datacrawler, we prepare the cluster by mounting a permanent disk. The permanent disk will be the central place, where results of the Datacrawler are saved to. Since we will run multiple workloads distributed among a set of compute nodes, we are not able to mount our central disk to each of them. Therefore, we deploy a Network File System provisioner and mount our central disk to it. The NFS provisionier is responsible for dynamically provisioning new volumes for anybody who requests disk space from central disk. Afterwards we deploy `redis` as a key-value store from the official Docker registry. It will be used by us to push and consume work items, in our case the domain to be analyzed and respective rank. Before we deploy the Datacrawler instances, we create a queue in `redis` and fill it with the rank and domain name separated by comma. Finally, we deploy a set of Datacrawler from our custom Datacrawler image (7.5.2), which will request a new volume from the NFS provisioner and start to process work items from the queue until the queue is empty. Upon each successfully analysis, the Datacrawlers will output the results to his provisioned volume, thus to the central storage.

Upon successful computation of the specific dataset, the central storage can be mounted to a arbitrary virtual machine to access the dataset.

7.5.2 Datacrawler as a Container

As discussed in earlier sections, Kubernetes is only able to schedule containers, not directly applications such as our Datacrawler. Simply illustrated a container represents an isolated subsystem of the host system where it is run. In contrast to a virtual machine, a container shares the same hardware and kernel of the host system. The isolation consists of isolating processes from the host system to the container and vice versa. Within a container the user is able to run multiple applications as he would do usual on the host system.

The Datacrawler container is created during the deployment by the Kubernetes cluster from a custom Docker image, which can be found in our internal Docker registry in Google Cloud Platform. We locally build the Datacrawler for `Linux`, pack it to a `Linux` Docker image on our local development machine and push it our internal Docker registry on Google Cloud Platform.

Upon we request the deployment of 100 Datacrawler instances, Kubernetes

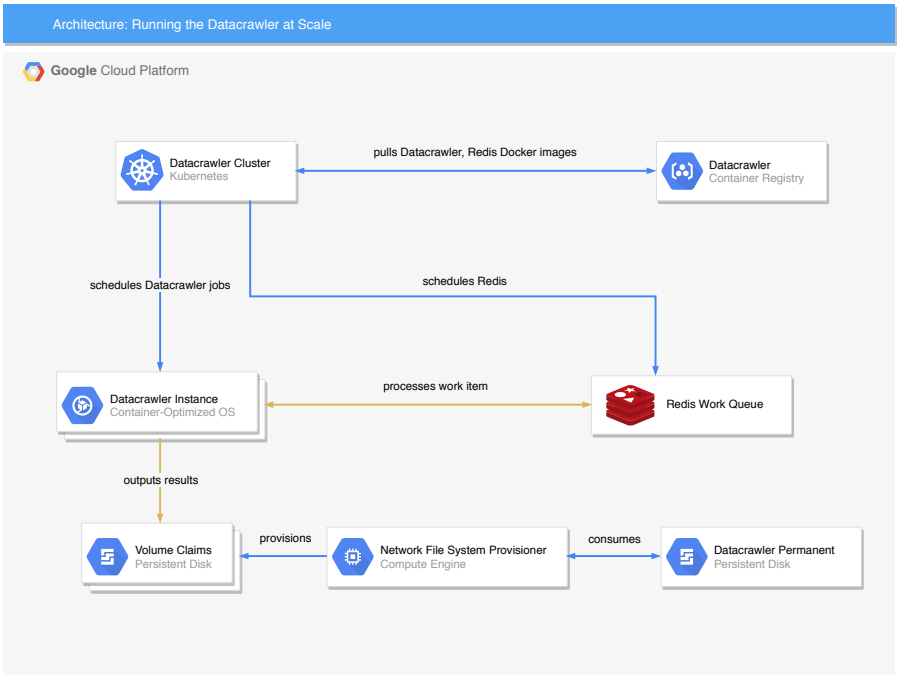


Figure 9: This figure illustrates the system design of the Datacrawler Kubernetes Cluster on Google Cloud Platform. It shows that the Datacrawler Kubernetes cluster consists mainly out of three applications: The `redis`-queue which represents a work queue consisting out of domains as work items to be analysed. The Network File System provisioner which is mainly responsible for providing volume claims for Datacrawler instances from the central storage. Finally, the Datacrawler which is deployed as a container from the internal Docker registry. The Datacrawler itself is executed wrapped by a `python`-application, which represents the tie between the `redis`-queue and the Datacrawler. It is resposible for retrieving of work items from the queue, starting the Datacrawler and confirming the work item upon successful analysis.

tries to allocate the resources required per Datacrawler. We define those requirements within the request. In our case if Kubernetes fails to allocate required resources, it automatically provisions new compute nodes to deploy the Datacrawler until the deployment is successful. Upon successful allocation, Kubernetes requests and mounts a volume from the NFS provisioner to `/opt/apt/datacrawler-data/`, which represents the location where results should be saved to. Afterwards the container is being started with the `Python`-file `worker.py` as entrypoint. `worker.py` is mainly responsible to be the tie between the `redis` work queue and the Datacrawler instances. It connects with the `redis`-instance in the cluster, accepts new working items, starts a Datacrawler instance for each work item and passes the rank and domain using environment variables to the Datacrawler. Upon the Datacrawler successfully exits, the work item is being confirmed in `worker.py`, which removes it from the working queue. This whole process repeats until the queue is empty.

8 Results

Presentation of our results, no opinion

9 Discussion

Discussion of the results, assumptions about why things are the way they are

10 Conclusion

Summary, outlook, future work

References

[Bat+16] Peter W. Battaglia et al. "Interaction Networks for Learning about Objects, Relations and Physics". In: *CoRR* abs/1612.00222 (2016). arXiv: 1612.00222. URL: http://arxiv.org/abs/1612.00222.

[Bat+18] Peter W. Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *CoRR* abs/1806.01261 (2018). arXiv: 1806.01261. URL: http://arxiv.org/abs/1806.01261.

[Bel17] Tony Beltramelli. "pix2code: Generating Code from a Graphical User Interface Screenshot". In: *CoRR* abs/1705.07962 (2017). arXiv: 1705.07962. URL: http://arxiv.org/abs/1705.07962.

[Bur+05] Chris Burges et al. "Learning to Rank Using Gradient Descent". In: *Proceedings of the 22Nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: ACM, 2005, pp. 89–96. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102363. URL: http://doi.acm.org/10.1145/1102351.1102363.

[Chr19a] Google Chrome. *Chrome DevTools*. Feb. 21, 2019. URL: https://chromedevtools.github.io/devtools-protocol/1-2.

[Chr19b] Chromium. *Content module*. Feb. 21, 2019. URL: https://www.chromium.org/developers/content-module.

[con18] Wikipedia contributors. *Kubernetes — Wikipedia The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=825655339 (visited on 02/21/2019).

[Dai+17] Hanjun Dai et al. "Learning Combinatorial Optimization Algorithms over Graphs". In: *CoRR* abs/1704.01665 (2017). arXiv: 1704.01665. URL: http://arxiv.org/abs/1704.01665.

[Dev19] Google Developers. *Puppeteer FAQ*. Feb. 21, 2019. URL: https://developers.google.com/web/tools/puppeteer/faq/.

[do/19] http://commoncrawl.org/big-picture/what-we do/. *Curious about what we do?* Jan. 5, 2019. URL: http://commoncrawl.org/big-picture/what-we-do/.

[Dom19] DomCop. *What is Open PageRank?* Jan. 5, 2019. URL: https://www.domcop.com/openpagerank/what-is-openpagerank.

[Fou+17] Alex Fout et al. "Protein Interface Prediction using Graph Convolutional Networks". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 6530–6539. URL: http://papers.nips.cc/paper/7231-protein-interface-prediction-using-graph-convolutional-networks.pdf.

[Fra19] Chromium Embedded Framework. *CEF : General Usage*. Feb. 22, 2019. URL: https://bitbucket.org/chromiumembedded/cef/wiki/GeneralUsage.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gü19] Samed Güner. *Evaluation and Extension of Current Automated Infrastructure Deployment Solutions for Container-based Clusters*. Mar. 23, 2019.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *CoRR* abs/1706.02216 (2017). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216.

[JSS16] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. "Unbiased Learning-to-Rank with Biased Feedback". In: *CoRR* abs/1608.04468 (2016). arXiv: 1608.04468. URL: http://arxiv.org/abs/1608.04468.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[Kub18] Kubernetes. *Cloud Foundry How applications are staged*. 2018. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 02/21/2019).

[KW16] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *CoRR* abs/1609.02907 (2016). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

[LeC+98] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[Liu+09] Tie-Yan Liu et al. "Learning to rank for information retrieval". In: *Foundations and Trends® in Information Retrieval* 3.3 (2009), pp. 225–331.

[Pas+18] Rama Kumar Pasumarthi et al. "TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank". In: *CoRR* abs/1812.00073 (2018). arXiv: 1812.00073. URL: http://arxiv.org/abs/1812.00073.

[San+18] Alvaro Sanchez-Gonzalez et al. "Graph networks as learnable physics engines for inference and control". In: *CoRR* abs/1806.01242 (2018). arXiv: 1806.01242. URL: http://arxiv.org/abs/1806.01242.

[Sel+18] Daniel Selsam et al. "Learning a SAT Solver from Single-Bit Supervision". In: *CoRR* abs/1802.03685 (2018). arXiv: 1802.03685. URL: http://arxiv.org/abs/1802.03685.

[tea18] x team.com. *Introduction Kubernetes Architecture*. 2018. URL: https://x-team.com/blog/introduction-kubernetes-architecture/ (visited on 02/22/2019).

[UVL17] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. "Deep Image Prior". In: *CoRR* abs/1711.10925 (2017). arXiv: 1711.10925. URL: http://arxiv.org/abs/1711.10925.

[w3s19] w3schools.com. *Browser Statistics*. Feb. 23, 2019. URL: https://www.w3schools.com/browsers/.