Université de Montréal

Patentor: Information Extraction from Chemical Patents

par

Mahmoud Nassif

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

June 8, 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Patentor: Information Extraction from Chemical Patents

présenté par

Mahmoud Nassif

a été évalué par un jury composé des personnes suivantes :

Nom du président du jury
(président-rapporteur)

Nom du directeur de recherche
(directeur de recherche)

Nom du membre de jury
(membre du jury)

Abstract

InVivo AI is a drug discovery platform that empowers scientists and researchers with the latest innovations in Machine Learning. Using a centralized data model it unobtrusively integrates drug discovery processes to improve the odds of successful clinical trials at a low cost.

Founded by Daniel Cohen, Therence Bois, Sébastien Giguère and Prudencio Tossou in 2018, InVivo AI was the host of my Mitacs internship between May 2020 and November 2020. My mandate was to build Patentor; an AI powered patent extraction tool that gathers, extracts and compiles standardized datasets from the trove of data available in pharmaceutical patents. The generated datasets can be used to improve the accuracy of other in-house models involved in drug discovery at InVivo AI.

This report summarizes the work completed during the internship to create the first proof of concept of Patentor. It is split into two main parts. The first describes the software engineering portion which goes over architecture, design and implementation details. The second part addresses the Document Layout Analysis of patents which is a Machine Learning technique that locates elements of different categories within scientific documents. It is used in the first stage of the patent to dataset pipeline. The model can be seen on GitHub `github.com/mabounassif/invivo_ssd_publaynet`.

Contents

List of Tables

List of Figures

Liste des sigles et des abréviations

SSD	Single Shot MultiBox Detector
DLA	Document Layout Analysis
PDF	Portable Document Format
API	Application Programming Interface
CRD	Custom Resource Declaration
DAG	Directed Acyclic Graph
HITL	Humain In The Loop

Acknowledgment

First and foremost, I would like to thank InVivoAI for giving me the opportunity to develop my skills. The team has been a true source of inspiration and humility.

To the people of Canada, to whom I am forever in debt. Thank you for allowing me to pursue this wonderful challenge. You have been aptly represented by Mitacs.

Finally, to my wonderful wife and biggest supporter Maxine, I love you. To Malek, my sweet little baby boy, you make me happy when skies are gray.

Introduction

Patentor is a chemical patent extraction tool. It is a software that can process batches of patents stemming from the pharmaceutical industry and convert them into an in-house standardized dataset. The dataset can be used after to help improve the accuracy of the models deployed in drug discovery.

A typical patent comes in as a PDF file and it contains chemical information about a newly discovered chemical compound. The information of interest to us can be categorized roughly in 3 ways as listed below:

(1) A structural description of the compound and its variants are illustrated as molecular drawings and/or chemical formulas.
(2) Chemical reactions related to the compound are defined as drawings and/or chemical formulas
(3) Chemical activities and properties are usually included in the form of tables and/or text and/or graphs

Parsing through the patents, one can quickly identify five representations of the information above. They include:

(1) **Text:** author, author affiliation; paper information; copyright information; abstract; paragraph in main text, footnote, and appendix; figure and table caption; table footnote
(2) **Title:** article title, standalone (sub)section title, standalone figure and table label.
(3) **List**: list of items
(4) **Table**: main body of a table
(5) **Figure**: illustrations of various kind

It becomes obvious that Patentor must include a Document Layout Analysis (DLA) component that can locate texts, titles, lists, tables and figures in a PDF document. How we get there will be the topic of the report.

We first start with the software architecture of the system that will support the layout analysis component. Then we show how the DLA component itself was built. Finally, we end with closing remarks.

Please note that the work described in this report stops at the implementation of the DLA. The timeline and the complexity of the project did not allow for a fully featured tool. Only the foundation was built. Patentor will require further development.

Chapter 1

System Architecture

Patentor has been implemented as a containerized Django application powered by a container-based workflow engine. We dive deep into what that means in the following section starting with the workflow engine.

The containerization of scheduled jobs is quickly becoming a core design decision when it comes to building a production grade machine learning pipeline. This new operating system (OS) paradigm bundled with Kubernetes which is considered the leading open source container-orchestration system, has led to the inception of many creative workflow engines.

One of them is Argo that we shall explore in more details below. However, before we do so, it would be valuable to investigate what containers really are and how would they be useful in our context. It is a fundamental technology that can be easily overlooked.

1.0.1. Containers: OS-level virtualization

In the last decade alone, virtualization technologies have seen an unprecedented progress in their development. Their general purpose has been to allow the partitioning of a computer system into multiple isolated virtual environments. They fall into two main categories: container-based virtualization and hypervisor-based virtualization. The former provides a more lightweight and efficient virtual environment. It allows ten times more virtual environments to run on a physical server compared to hypervisor-based virtualization [**2**]. This allows for a higher density of virtual environments which is mostly due to the fact that a container does not require a full OS to run. Additionally, container-based virtualization provide better performance matching that of native applications [**10**]. Finally, containers take away repetitive, mundane configuration tasks and are used throughout the development lifecycle for fast, easy and portable application development - desktop and cloud. For these reasons, we selected for the containerization-based virtualization for the rest of this section.

A container-based virtualization works at the operating system level, thus allowing multiple applications to operate without redundantly running operating system kernels on the

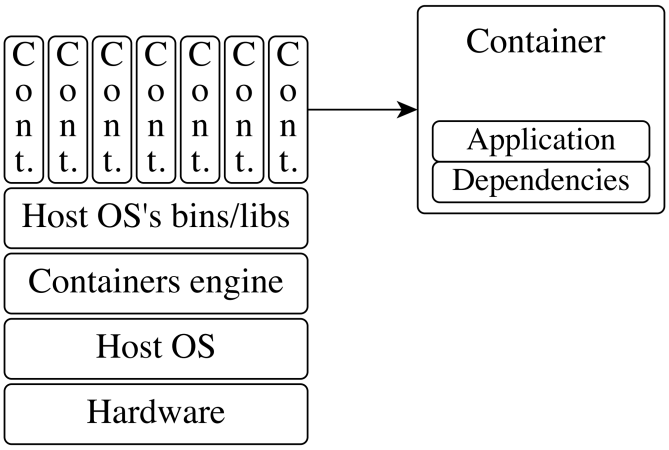

Figure 1.1. Architecture of Container-based Virtualization

host as can be seen in Fig. 1.1. Its containers look like normal processes from outside, which run on top of the kernel shared with the host machine. They provide isolated environments with necessary resources to execute applications. These resources can be either shared with the host or installed separately inside the container.

Even if container-based virtualization comes with security concerns [**4**] and shortcomings in supporting environments of different type as that of the host (i.e. Windows containers cannot run on Linux host), Docker dominates amongst technologies that deploy distributed container-based applications. In the following section, we inspect the architecture of Docker and identify the traits that makes it, and other similar CRI runtimes, ideal candidates for Kubernetes.

1.0.2. Docker

Docker is an open source container technology with the ability "to build, ship, and run distributed applications". It is very popular amongst the biggest tech companies for a reason.

Although emerging late in the world of container technologies, Docker brings a better user experience through its simple interface, setting itself apart from the competition. Not only does it provide the means to safely and effectively control containers, it also provides a robust cross-platform experience where a container can run almost anywhere whether that is locally or in a data center, allowing for a fast and consistent delivery of applications.

Docker leads in the number of containers it can run per hardware unit while maintaining a good cooperation with third-party tools [**10**]. Docker is backed by a powerful engine called the Docker engine.

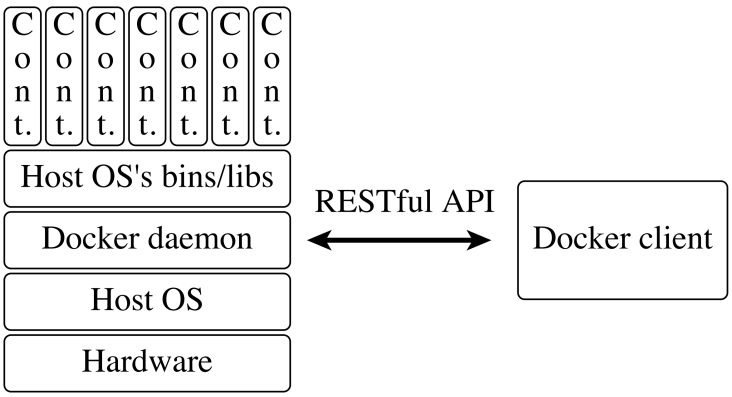

Figure 1.2. Architecture of Docker Engine

The Docker Engine is a lightweight and portable packaging tool which relies on container-based virtualization. We can see the overall architecture of the Docker engine in Fig. 1.2. The containers run on top of the *Docker daemon* that executes and manages them.

At its core, Docker leverages the libcontainer library that provides functionalities around cgroups, namespaces, capabilities, and filesystem access controls that are needed to create containers. The cgroups are a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. The namespaces, as the name implies, restrict a particular kernel resource dimension to its own isolated view. They include pid (process views), mnt (mounts), ipc (shared memory, message queues etc), UTS (host and domain names), user (uids and gids), and net (network view). For example, each net gets its own isolated network interfaces, loopback device, routing table, iptable chains and rules [**9**]. The capabilities are the units that processes privileges are broken down into, which can be independently enabled and disabled. Capabilities are a per-thread attribute. Most file systems include attributes of files and directories that control the ability of users to read, change, navigate, and execute the contents of the file system. The filesystem access controls are the attributes.

At this point, we can clearly see the richness of the Docket ecosystem that makes it an ideal candidate for a container orchestrator such as Kubernetes on which Argo, the backbone of the ML pipeline, was built. Kubernetes will be the topic of the next section.

1.0.3. Kubernetes

Kubernetes is an open-source container-orchestration system for automating computer application deployment, scaling and management [**3**]. Initially, Kubernetes used to interface with the Docker runtime via a Dockershim interface. They started rolling out a more native alternative, the Container Runtime Interface (CRI) compliant *containerd*.

We will go over the main components detailed in the architecture schematic illustrated in figure 1.3

1.0.3.1. Kubernetes Master. The Kubernetes Master is the main controlling unit of the cluster. It is usually recommended to run on its own dedicated server. This will protect it from performance strains that could arise with a heavy load. The control pane includes multiple components that each run in their own process on one or multiple machines in the case of high availability clusters. *etcd* is a persistent, lightweight, distributed, key-value data store developed by CoreOS that reliably stores the configuration data of the cluster, representing the overall state of the cluster at any given point of time. It favors consistency over availability in the event of a network partition. The consistency is crucial for correctly scheduling and operating services. The kubernetes API server listens to changes on *etcd* and tries to restore any divergences back to the initial configuration that was determined by the deployer.

The API server is a key component and serves the Kubernetes API using JSON over HTTP. It serves the internal mechanisms of the cluster and also provides the interface to external users of the cluster.

The Scheduler tracks resource use on each node and selects the node on which to run an unscheduled pod based on resource availability. It basically matches the resource "supply" to workload "demand".

The Controller manager is the main component that actively tries to pull the current state of the cluster closer to the desired state. It leverages the API server to create, delete and update the resources it manages. It also manages a set of core Kubernetes controllers that can influence the cluster.

1.0.3.2. Kubernetes Node. The Kubernetes Node is the workhorse of the cluster. It behaves like a Worker or a Minion, which represents a machine where the containers are deployed. Each node in the cluster must run a container runtime such as Docker, a Kubelet and a Kube-proxy. A container resides inside a Kubernetes Pod which is the basic scheduling unit in Kubernetes that we will explore more later. The Kubelet is responsible for the running state of the Node. It monitors cosntantly the health of the containers.

The Kube-proxy is an implementation of a network proxy and a load balancer. It is behind the service abstraction and other network operations. It directs traffic to containers based on ip addresses and port numbers.

Finally, the container runtime is the binary that runs the containers in this case that would be the Docker runtime that we have already covered before.

1.0.3.3. Kubernetes Extensions. Kubernetes is highly configurable and extensible. As a result, the core project code changes very rarely. When it comes to extending and customizing the basic functionalities of Kubernetes there are two flavors: *configuring* the cluster (which

only involves changing flags, local configuration files, or API resources) and *extensions*. We will focus more on the latter as it is the main way of implementing the Argo Worflows system, the topic of the next section.

The extensions are software components that extend and deeply integrate with Kubernetes. They adapt it to support new types and new kinds of hardware [**3**].

Client programs that reads and/or writes to the Kubernetes API can provide useful automation. Implementing them via the *Operator* pattern works well with Kubernetes. They typically read an object's *.spec*, possibly do things and then update the object's *.status*. Objects are records that are stored in the *etcd* data storage. They help construct the global state of the cluster guiding which actions are dispatched for different scenarios that are played out in the cluster.

An extension can also register and create custom resources through the Custom Resource Declaration (CRD). In general, a *resource* is an endpoint in the Kubernetes API that stores a collection of API objects. The custom *resource* extends the Kubernetes API and provides new types to work with that otherwise would not be available.

Combining CRD with a custom operator provides a true *declarative* API. A declarative API allows you to declare a desired state of your resource and tries to keep the current state of Kubernetes objects in sync with the desired state. You can use custom operators to encode domain knowledge for specific applications into an extension of the Kubernetes API [**3**]. This is exactly how Argo was implemented.

1.0.4. Argo Workflows

Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. Argo Workflows is implemented as a Kubernetes CRD (Custom Resource Defintion). It allows a user to:

- Define workflows where each step in the workflow is a container
- Model multi-step workflows as a sequence of tasks or capture the dependencies between tasks using a directed acyclic graph (DAG).
- Easily run compute intensive jobs for machine learning or data processing in a fraction of the time

Argo's architecture is illustrated in fig 1.4. Argo introduces four main components: Workflow Archive, Artifact Store, Workflow Controller and Argo Server.

The main component, the Argo Workflow Controller essentially listens to changes in pod states via the *Pod Informer* interface of the Kubernetes API. It then queues those objects in a rate limited work queue instead of processing them directly as soon as a change happens. This ensures that the same item is not processed twice and that only a limited amount of resources are processing those items.

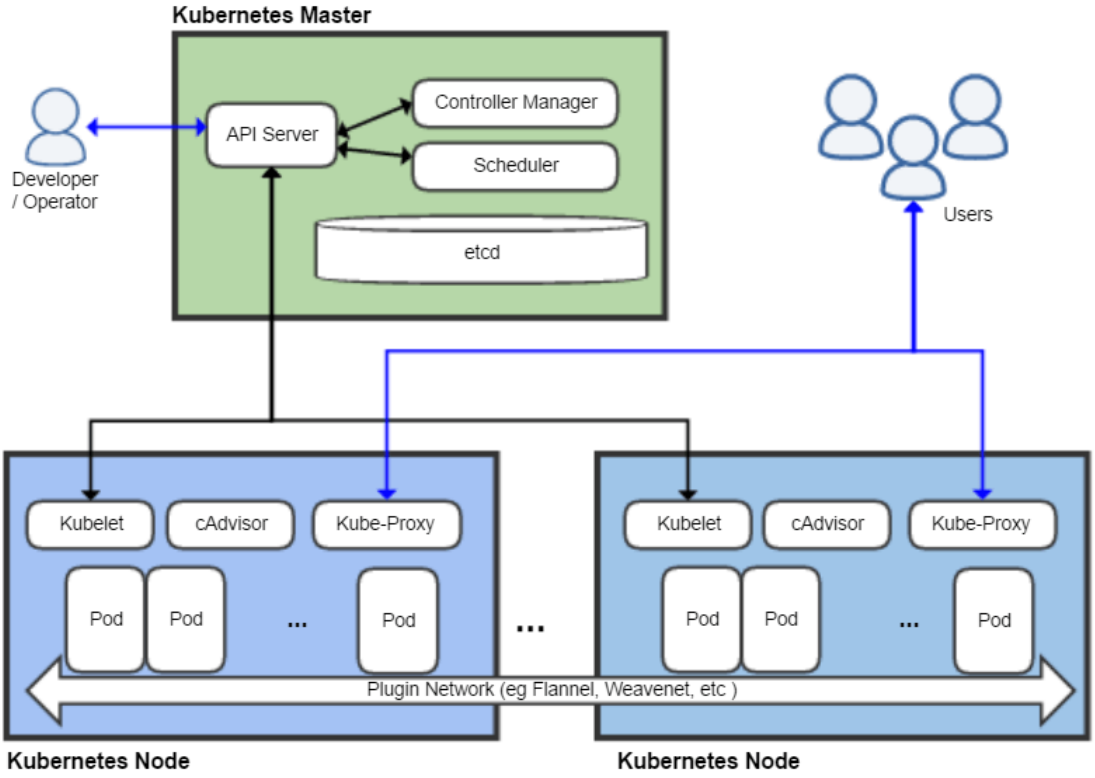

Figure 1.3. Architecture of Kubernetes

When a pod item is processed after it had been queued. The corresponding *workflow* key is extracted from a simple label cross referencing and added to the workflow queue as illustrated in fig 1.5.

Workflows keys are picked up by a group of workers. They build up a workflow execution context that will be needed in the subsequent stage. During that stage, they evaluate the current state of the workflow and its pods then decides how to proceed down the execution path. [**1**]

The Argo Server is a server that exposes an API and UI for workflows. The Workflow Archive is a persistent layer that keeps historical data. The Artifact Store is essential to store the input and output data of a job.

We can now project a clear mental picture on how such a powerful system would be an ideal platform for a machine learning (ML) pipeline. In the next section, we will look at the architecture of Patentor and the way it leveraged Argo to build a full end to end ML pipeline with a human in the loop (HITL).

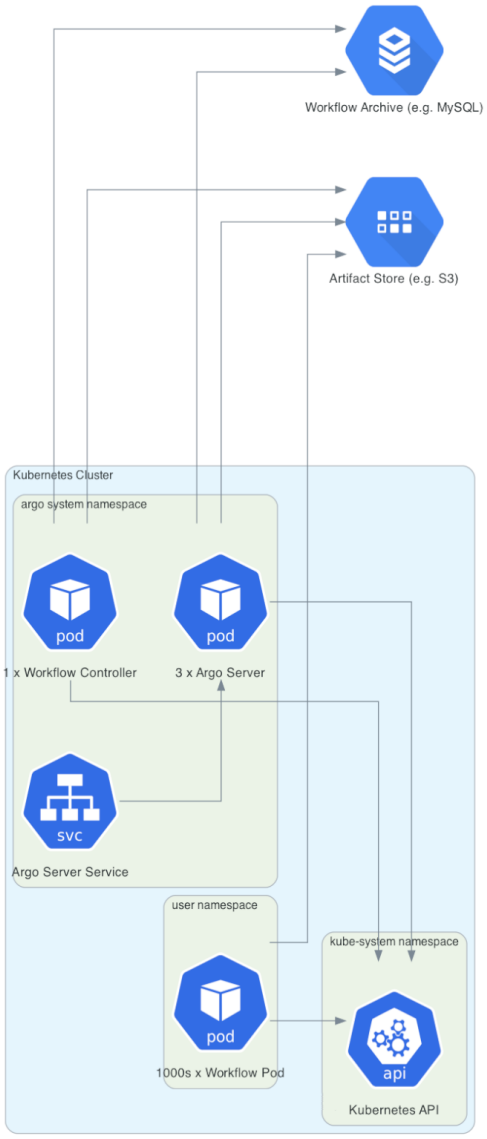

Figure 1.4. Architecture of Argo Workflows

1.0.5. Patentor Architecture

At the time of writing, InVivo has been developing a concept of an operating system that enables synthetic drug discovery. The main idea is to run many software applications on top of a consolidated data model. This makes for a powerful and extensible platform that effectively supports the wide range of user needs without compromises.

The Django project presented itself as a compatible framework for such a vision. It is a Python based open-source web framework that follows the model-template-view (MTV) architectural pattern. There is emphasis on components reusability and a plugin based

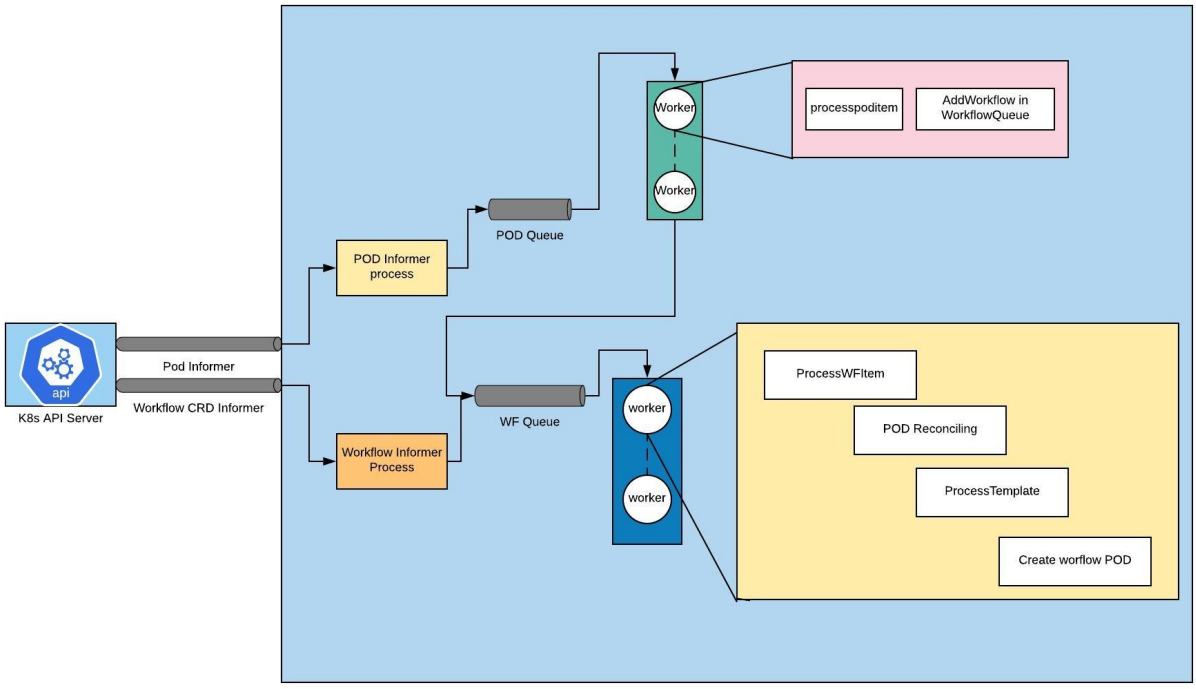

Figure 1.5. Argo Workflow Controller Design

integration. This allows for rapid development, less code, low coupling and embraces the principle of *Don't Repeat Yourself*. In addition, the complete adherence to Python fits well with the Machine Learning ecosystem of tools and libraries stemming from advanced research, the bedrock of the talent at InVivo.

In order to explore further the MTV framework, it is important to take a step back and visit the model-view-controller (MVC) pattern first. The MVC software design pattern commonly used for developing user interfaces divides the software stack into three interconnected elements as shown in figure 1.6. This dissociates the way information is internally represented from the way it is presented to the user. The components are the follow:

(1) **Model**: The model component manages the data, logic and rules of the application.
(2) **View**: The view component represents the information in a predetermined way. Multiple views can exist for the same data.
(3) **Controller**: The controller component accepts user input and converts it to commands for the model and the view.

Because the controller is handled by the framework itself and most of the work happens in models, templates and views, Django has been referred to as an MTV framework with its components listed below:

(1) **Model**: the data layer. It knows how to access it, how to validate it, which behaviors it has, and the relationships between the data

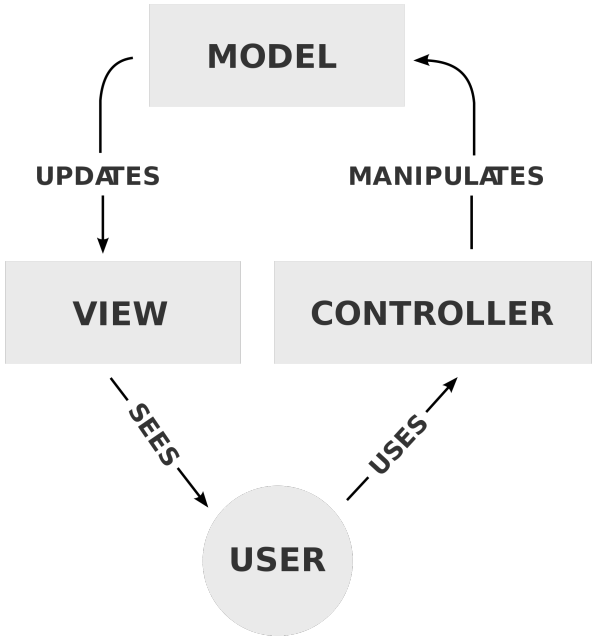

Figure 1.6. MVC Design Pattern

(2) **Template**: the presentation layer. It contains presentation-related decisions, for example, how something should be displayed on a website.
(3) **View**: the business logic layer. It contains the logic that access the model and defers to the appropriate template(S). It bridges the Model and Template.

The comparison between MVT and MVC can be ambiguous as it relates to the different interpretation of the MVC. In Django's interpretation, the view describes *which* data is being presented not just *how* it is presented. In contrast, other popular frameworks such as Rails more akin to the MVC pattern consider a view more like a template in Django and a controller is made more explicit instead of offloading it to the framework.

A main web application is represented as a project in Django. Within a project, there might be one or more apps. Each app owns its own models and is considered standalone as depicted in figure 1.7.

In our case, Patentor has been implemented as a Django application. It contains a *Patent* model that is uniquely identified. The record stores the S3 path of the PDF version of the patent to be annotated, it also stores the manual and automatically generated bounding box annotations in JSON format. Postgres was used as the persisting layer.

Patentor is a RESTful API which requires the templates within it to map the attributes of the model to a JSON format encapsulated within the body of the HTTP response. All the validations, security, hooks and business logic are contained within the views.

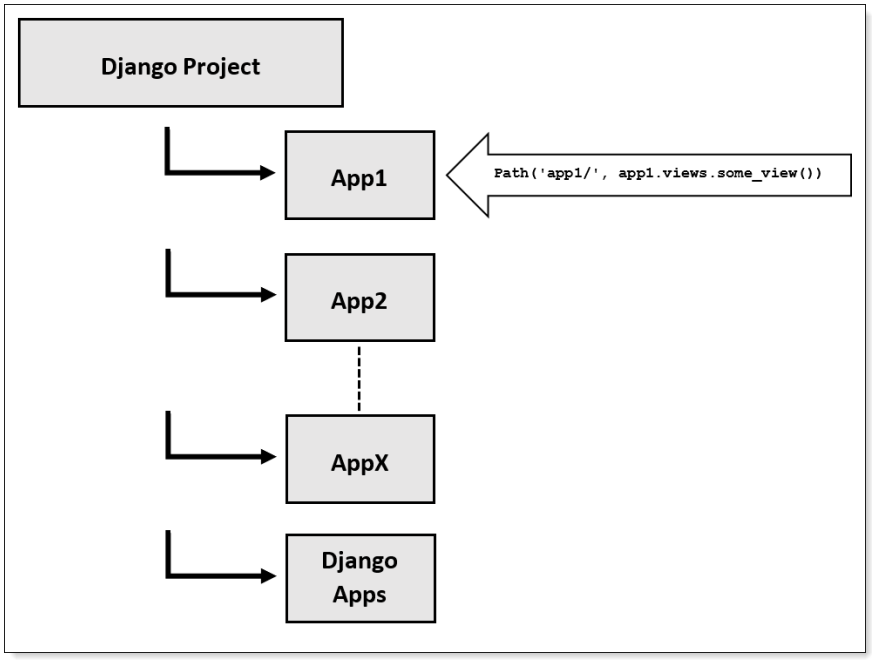

Figure 1.7. Django Project and Applications

Visually, a logged in user can drag and drop a patent in PDF format on the Patentor interface which triggers an upload of the file to the S3 bucket and also creates the above *Patent* model with empty annotations in the Postgres database. Once the upload and the record creation completed, Patentor kicks off a job on Argo. The job pulls in the latest trained DLA model from S3 and uses it to predict the bounding boxes of the data representations of interest listed in the introduction. This fills in the automatically generated annotations of the *Patent* record associated with the patent in question. In the meantime, the user can always select the patent and manually annotate the Patent using an intuitive bounding boxes annotator tool.

Independently, there is a scheduled background job, that kicks off the training of the DLA model. It pulls in all the manually generated annotations and retrains the model to improve accuracy by finetune. The data flow can be visualized in the figure 1.8.

Now that we covered the backend of Patentor and how everything works together to process patents, we move on to the magic behind the layout analysis in the next chapter.

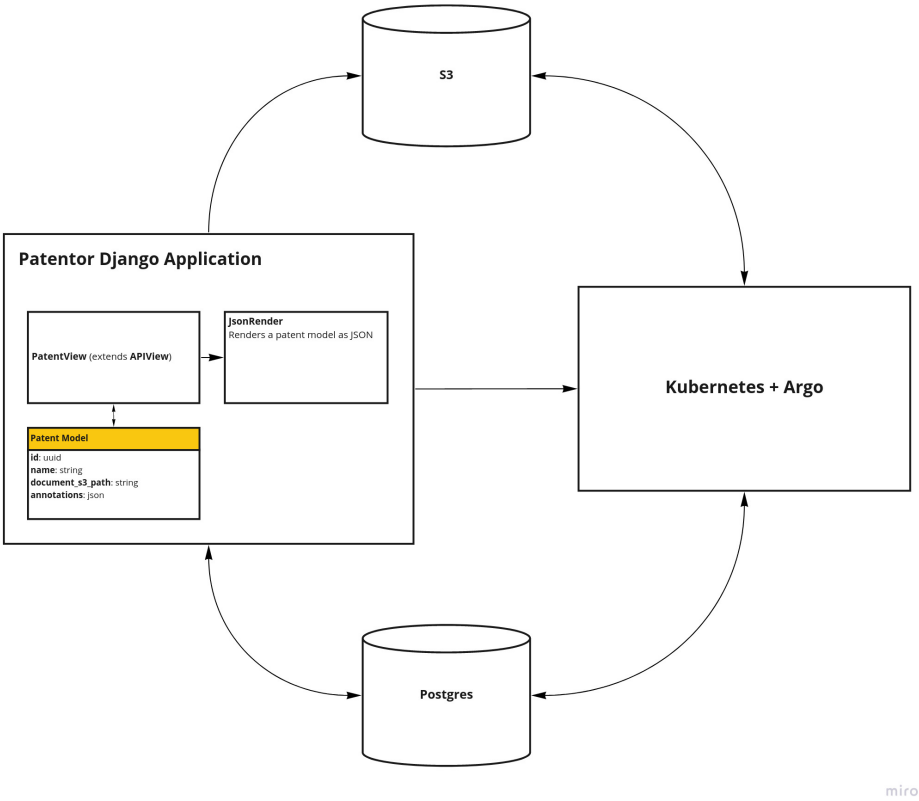

Figure 1.8. Patentor Architecture

Chapter 2

Machine Learning
Document Layout Analysis

Now that we have established the platform of Patentor, we move on to the Machine Learning part of the report which goes over the Document Layout Analysis component.

Recognizing the layout of a digital patent is an essential step in converting documents into a structured machine-readable format [**11**]. Attempts to complete the task started out with a combination of geometrical layout analysis and optical character recognition methods [**5**]. Until more recently, image analytics methods based on deep learning have become prevalent in the domain[**7**].

In the following section, deep learning methods are investigated more thoroughly. Emphasis have been put on their various computational performances such as running time and accuracy.

Two main models have been compared side by side, the Mask R-CNN [**7**] and the SSD [**8**]. Finding the balance between accuracy, speed and cost was the upmost priority and these models dominated different parts of the spectrum. Although, the R-CNN outperforms its predecessor the Fast R-CNN [**6**] in all three tracks of the COCO suite of challenges it still clocks out at 5 FPS. The SSD, on the other hand, can offer a top processing speed of 58 FPS on a Nvidia Titan X and for 500×500 input, while still outperforming the Fast R-CNN. However, the two models do not seem to have been compared in the context of document layout analysis. That would be the goal of the following section.

Firstly, the dataset used for training will be presented, followed by a quick overview of the architecture of each model and lastly the results of the comparison will be shared.

2.1. PubLayNet Dataset

The PubLayNet dataset is a dataset that was compiled using a large collection of documents from PubMed Central Open Access (PMCOA). It combined the PDF and XML

	Training	Development	Testing
Pages			
Plain pages	87,608	1,138	1,121
Title pages	46,480	2,059	2,021
Pages with lists	53,793	2,984	3,207
Pages with tables	86,950	3,772	3,950
Pages with figures	96,656	3,734	3,807
Total	340,391	11,858	11,983
Instances			
Text	2,376,702	93,528	95,780
Title	633,359	19,908	20,340
Lists	81,850	4,561	5,156
Tables	103,057	4,905	5,166
Figures	116,692	4,913	5,333
Total	3,311,660	127,815	131,775

Table 2.1. Statistics of training, development, and testing sets in PubLayNet

representations of the same document to locate the different layout components in the article. At the end, the dataset contains 360 thousand annotated document images. [**11**].

The layout components have been broken down into 5 main categories: titles, texts, figures, lists and tables. The distribution of those components in the dataset can be seen in the table 2.1.

The Mask R-CNN and the SDD models were both trained on the PubLayNet. We compare their architectures and performances in the following sections.

2.2. Mask R-CNN

Mask R-CNN extends the Faster R-CNN by adding a third element to the output of each candidate object. The output consists then of a class label, a bounding-box offset and the object mask.

2.2.0.1. Faster R-CNN. It would be valuable to briefly go over the two stages of the Faster R-CNN. On the one hand, there is a Region Proposal Network (RPN) proposing candidate object bounding boxes. On the other, a stage that extracts features using RoIPool from each candidate box and performs classification and bounding-box regression. During inference, performance gains are made if features of both stages are combined.

2.2.0.2. Mask R-CNN. Mast R-CNN is also assembled in two stages. The first stage is identical to that of the Faster R-CNN; the Region Proposal Network (RPN). In the second stage, while predicting the class and box offset the Mask R-CNN also predicts the binary mask for each RoI. This breaks from the traditional approaches where classification depends on the mask prediction.

Mask R-CNN has a detection speed at about 5 frames per second on average. Another model the Single Shot Detection (SSD) can run up to 58 frames per second. This will be the topic of the next section.

2.3. SSD

The Single Shot Multibox Detector divorces from the Faster R-CNN inspired object detector by removing components that resample pixels or features for bounding boxes hypotheses while still being competitive in accuracy. Although, these changes have been popular before, SSD is unique in the improvements in accuracy. They include a small convolutional filter to predict object categories and offsets in bounding box locations, using separate filters for different aspect ratio detections, and applying these filters to multiple feature maps from the later stages of a network.

2.4. Training

The M-RCNN was trained in the same way as [**11**] using the Detectron implementation. The model was trained for 180k iterations with a base learning rate of 0.01. The learning rate was reduced by a factor of 10 at the 120k iteration and the 160k iteration. 8 GPUs with one image per GPU were used. The backbone was the ResNeXt-101-64x4d loaded pre-trained on ImageNet. Luckily, the results were reproduceable.

The SSD was trained using the mmdetection implementation. The base learning rate was 0.002 with a momentum of 0.9 and weight decay of 0.0005. 4 GPUs with 8 images per GPU were used. The backbone was vgg16-caffee loaded pre-trained on ImageNet.

2.5. Results

The evaluation metric is the mean average precision (MAP)@ intersection over union (IOU) [0.50:0.95] of bounding boxes, which is used in the COCO competition [**11**]. Both models can generate accurate (MAP>0.89) document layout, where M-RCNN shows a small advantage over SSD. The SSD is faster at inference by a factor of 10 which makes it a better option for scalability from an engineering perspective. The results can be seen in the table 2.2.

Method	Precision	Recall	FPS
F-RCNN	0.972	0.964	5
M-RCNN	0.940	0.955	5
SDD	0.898	0.917	58

Table 2.2. Comparaison of the F-RCNN, M-RCNN and SSD on PubLayNet

Figure 2.1. Some bad SSD inferences

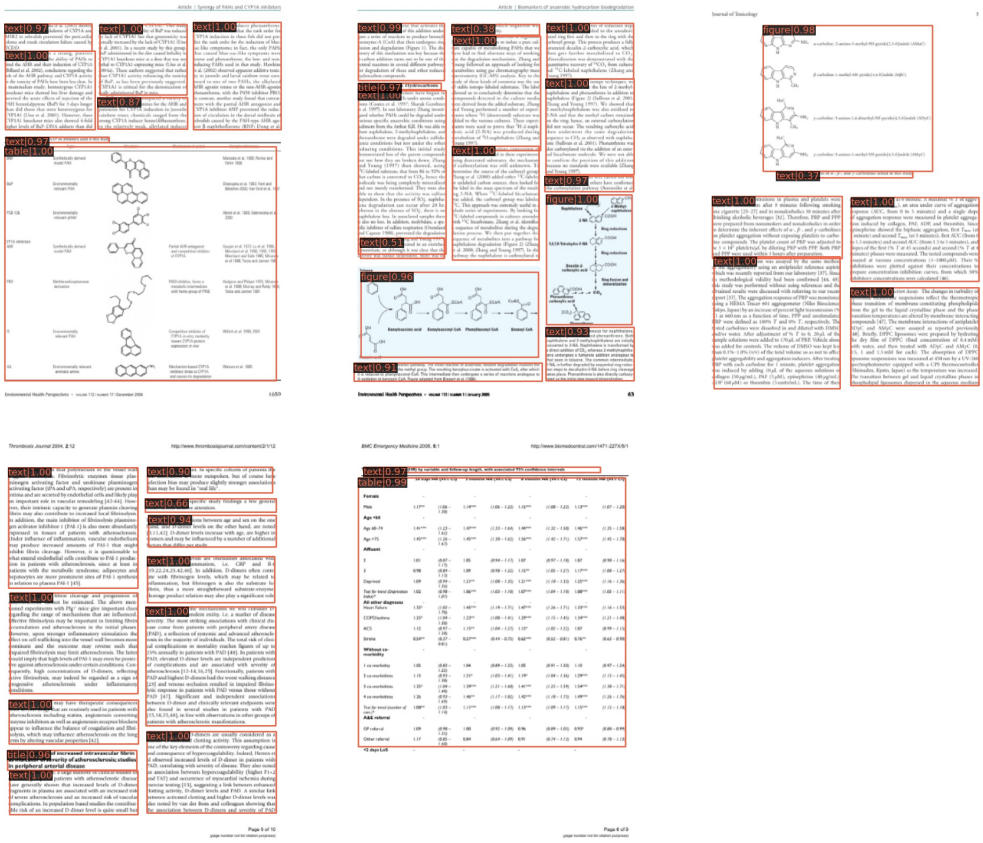

Figure 2.2. Some good SSD inferences

2.6. Conclusion

The choice between M-RCNN and SSD comes down to a compromise. On the one hand, M-RCNN excels at inference with an unmatched accuracy (MAP > 0.9). On the other hand, SSD has unmatched inference speed, clocking at a whopping 58 fps.

For 300 pages long patent, it would take about 5 seconds to identify the components with SSD versus 60 seconds with M-RCNN. It is about a 90 percent decrease in computing cost, which can be worth the 4 percent hit in precision.

Références bibliographiques

[1] Argo workflows official documentation.

[2] C. burniske. containers: The next generation of virtualization?

[3] Kubernetes official documentation.

[4] Thanh Bui : Analysis of docker security, 2015.

[5] Roldano Cattoni, T. Coianiz, Stefano Messelodi et Carla Modena : Geometric layout analysis techniques for document image understanding: a review. 11 2000.

[6] Ross Girshick : Fast r-cnn, 2015.

[7] Kaiming He, Georgia Gkioxari, Piotr Dollár et Ross Girshick : Mask r-cnn, 2018.

[8] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu et Alexander C. Berg : Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[9] Victor Marmol, Rohit Jnagal et Tim Hockin : Networking in containers and container clusters. *Proceedings of netdev 0.1, February*, 2015.

[10] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange et C. A. F. De Rose : Performance evaluation of container-based virtualization for high performance computing environments. *In 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, 2013.

[11] Xu Zhong, Jianbin Tang et Antonio Jimeno Yepes : Publaynet: largest dataset ever for document layout analysis. *In 2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1015–1022. IEEE, Sep. 2019.