Annexe B

Module vectorisation : description des développements

Les méthodes développées pendant mon stage ont posé les bases d'un *workflow*, qui a très rapidement évolué. Ces méthodes ont ensuite été itérativement raffinées pour orienter la plateforme vers plus de modularité, notamment grâce à l'implémentation de l'instance *Treatment*, sur laquelle repose désormais le lancement des actions. Cette annexe présente le développement du *workflow* de lancement de la vectorisation les *Witnesses*, ainsi que son évolution.

B.1 Workflow

- Requête utilisateur.rice dans l'application AIKON ;
- Lancement d'un *Treatment* :
  - Identification des *Witnesses* ;
  - Parser les *Manifests* et ramener les URLs des régions d'images dans une liste ;
  - Création d'un fichier JSON
  - Envoie du JSON via requête HTTP POST à l'API *endpoint* ;
- Inférence du modèle dans Discover-Demo :
  - Vérification du JSON ;
  - Lancement d'une tâche ;
  - Enregistrement des images ;
  - Inférence avec le modèle : écriture des fichiers SVGs ;
  - Envoi d'un ZIP contenant les fichiers à l'application via requête HTTP POST ;
- Récupération du ZIP au *endpoint* de l'application ;
  - Dézip, lecture et écriture des SVGs en *backend* dans l'application AIKON ;

  - Vérification dans le répertoire de résultats (dossier dans `mediafiles`) que deux fichiers du même nom n'existent pas ;
  - S'il existe déjà un fichier du même nom, il est écrasé par le nouveau ;
- Apparition des visualisations dans l'interface ;

B.2 Settings

B.2.1 AIKON

Le fichier de configuration (`.env`) de l'application doit contenir l'URL de l'API à laquelle elle se connecte. Afin d'activer les fonctionnalités de vectorisation, il est aussi nécessaire de spécifier le module correspondant dans les paramètres de configuration.

```
# Computer vision apps to install
ADDITIONAL_MODULES=regions,similarity,vectorization
```

Listing B.1 – Extrait du fichier de configuration de l'application.

B.2.2 Discover-Demo

Dans le fichier `.env`, la variable `INSTALLED_APPS` contient la liste des modules à charger. Pour activer les fonctionnalités, notamment la vectorisation, il est nécessaire d'ajouter les modules correspondants à cette liste.

```
# apps (folder names) to be imported to the API
INSTALLED_APPS=dticlustering,watermarks,similarity,region,vectorization
```

Listing B.2 – Extrait du fichier de configuration de l'API.

B.3 Envoi d'un traitement de vectorisation

B.3.1 Initialisation de la requête

Le processus de vectorisation était initialement déclenché via l'interface d'administration Django, en utilisant le mécanisme des `@admin.action` (des fonctions appelées avec une liste d'objets sélectionnés depuis la page de liste pour modification, elles agissent donc au niveau du témoin). Cette approche, bien que fonctionnelle, présentait des limites : notamment concernant le suivi des traitements et leur application sur différents objets de la base.

Une refonte des processus a été entreprise par Jade Norindr, Ségolène Albouy et Fouad Aouinti. Cette évolution a conduit à la création d'un formulaire dédié au lancement

de tous les traitements, standardisant ainsi les modes de communication avec l'API. Ce formulaire, accessible depuis l'interface utilisateur.rice, permet d'appliquer le traitement sur un ensemble d'objets précédemment sélectionnés.

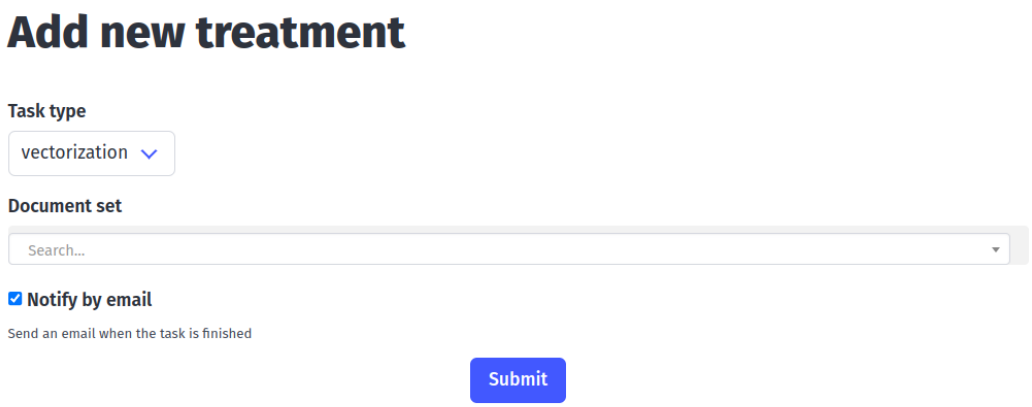

FIGURE B.1 – Capture d'écran du formulaire d'envoi d'un traitement dans AIKON.

Le lancement de la vectorisation se fait donc désormais au niveau de l'entité *Treatment* reliée à un ensemble de témoins. Le formulaire de lancement permet de créer une instance de cette entité, et par conséquent, tous les témoins associés seront soumis au processus de vectorisation.

B.3.2 Création du json

Initialement la fonction utilitaire `vectorization_request_for_one` était utilisée pour formater un fichier JSON à envoyer au *endpoint* de l'API. Afin de traiter par lots un ensemble de témoins, la fonction `vectorization_request` permettait d'itérer sur une liste de témoins, en appelant récursivement la fonction `vectorization_request_for_one` pour chacun d'entre eux.

```
try:
        response = requests.post(
        url=f"{CV_API_URL}/vectorization/start",
        json={
                "doc_id": regions.get_ref(),
                "model": f"{VECTO_MODEL_EPOCH}",
                "images": get_regions_urls(regions),
                "callback": f"{APP_URL}/{APP_NAME}/get-vectorization",
                },
        )
```

Listing B.3 – Extrait de la fonction gérant l'envoi de JSON à l'API dans `app/vectorization/utils.py`

Avec les récents développements, le JSON utilisé pour communiquer avec l'API est généré par la fonction `prepare_request`, présente dans le fichier `utils.py` de chaque module.

La boucle externe itère sur les *Witnesses* et vérifie qu'il n'y a pas déjà de vectorisation effectuée. La boucle interne itère sur les annotations des *Witnesses*. Le dictionnaire `regions_dic` est créé pour mapper les références des témoins annotés aux URLs des annotations. La fonction renvoie un dictionnaire contenant les données nécessaires pour la requête de vectorisation.

```
def prepare_request(witnesses, treatment_id):
    regions_list = []
    regions_dic = {}

    try:
        for witness in witnesses:
            if witness.has_vectorization():
                log(
                    f"[vectorization_request] Witness {witness.get_ref()} already
                        has vectorizations"
                )
                pass
            else:
                regions_list.extend(witness.get_regions())

        if regions_list:
            for regions in regions_list:
                regions_dic.update({regions.get_ref(): get_regions_urls(regions)
                    })

            return {
                "experiment_id": f"{treatment_id}",
                "documents": regions_dic,
                "model": f"{VECTO_MODEL_EPOCH}",
                "callback": f"{APP_URL}/{APP_NAME}/get-vectorization", # URL to
                    which the SVG zip file must be sent back
                "tracking_url": f"{APP_URL}/{APP_NAME}/api-progress",
            }

        else:
            return {
                "message": f"No regions to vectorize for all the selected {WIT}es
                    "
```

```
            if APP_LANG == "en"
            else f"Pas de regions à vectoriser pour tous les {WIT}s
                sélectionnés"
        }

    except Exception as e:
        log(
            f"[prepare_request] Failed to prepare data for vectorization request
                ",
            e,
        )
        raise Exception(
            f"[prepare_request] Failed to prepare data for vectorization request
                "
        )
```

Listing B.4 – Structurer les URLs de régions d'images.

La fonction reçoit une liste de *Witnesses*, générée par une tâche appelée par une méthode `post_save` de l'instance de *Treatment* :

```
# vhs/app/webapp/models/treatment.py
@receiver(post_save, sender=Treatment)
def treatment_post_save(sender, instance, created, **kwargs):
    if created:
        get_all_witnesses.delay(instance)
```

Listing B.5 – Méthode `post_save` du *Treatment*.

```
# vhs/app/webapp/tasks.py
@celery_app.task
def get_all_witnesses(treatment):
    try:
        witnesses = treatment.get_witnesses()
        treatment.start_task(witnesses)
    except Exception as e:
        treatment.on_task_error(
            {
                "error": f"Error when retrieving documents from set: {e}",
                "notify": treatment.notify_email,
            },
        )
```

Listing B.6 – Ramener tous les *Witnesses* à partir des entités reliées aux *Treatment* et lancer de la tâche.

B.4 Discover-Demo : module vectorization

B.4.1 Réception à l'*endpoint* de l'API :

L'*endpoint* (dans `api/app/vectorization/routes.py`) sert de passerelle pour lancer la tâche de vectorisation. Il valide la requête entrante, extrait les paramètres nécessaires et transfère la tâche au processus en arrière-plan pour son exécution.

```python
@blueprint.route("start", methods=["POST"])
@shared_routes.get_client_id
@shared_routes.error_wrapper
def start_vectorization(client_id):
    """
    A list of images to download + relevant data
    """
    if not request.is_json:
        return "No JSON in request: Vectorization task aborted!"

    json_param = request.get_json()
    console(json_param, color="cyan")

    experiment_id = json_param.get("experiment_id")
    documents = json_param.get("documents", {})
    model = json_param.get("model", None)

    notify_url = json_param.get("callback", None)
    tracking_url = json_param.get("tracking_url")

    return shared_routes.start_task(
        compute_vectorization,
        experiment_id,
        {
            "documents": documents,
            "model": model,
            "notify_url": notify_url,
            "tracking_url": tracking_url,
        },
    )
```

Listing B.7 – *Endpoint* `start_vectorization`.

B.4.2 Configuration de la tâche de fond

Dans le fichier `api/app/vectorization/tasks.py`, la fonction `compute_vectorization` définit un acteur Dramatiq qui exécute la tâche de vectorisation en *background*. Elle initialise une nouvelle instance de la classe `LoggedComputeVectorization`, en lui passant le logger et les paramètres fournis. Elle appelle la méthode `run_task` sur l'objet créé, qui va lancer le processus.

```python
@dramatiq.actor(time_limit=1000 * 60 * 60, max_retries=0, queue_name=VEC_QUEUE)
def compute_vectorization(
    experiment_id: str,
    documents: dict,
    model: str,
    notify_url: Optional[str] = None,
    tracking_url: Optional[str] = None,
    logger: TLogger = LoggerHelper,
):
    """
    Run vecto task on lists of URL
    """
    vectorization_task = LoggedComputeVectorization(
        logger,
        experiment_id=experiment_id,
        documents=documents,
        model=model,
        notify_url=notify_url,
        tracking_url=tracking_url
    )
    vectorization_task.run_task()
```

Listing B.8 – Tâche `dramatiq compute_vectorization`.

B.4.3 inférence avec le modèle

La classe `LoggedComputeVectorization` gère les processus de vectorisation en utilisant les paramètres fournis. La méthode `run_task` lance sur l'instance le téléchargement des données, l'inférence du modèle, le traitement des résultats (renvoi des SVGs sous forme de ZIP) et la notification, avec journalisation tout au long du processus.

Dans le fichier `api/app/vectorization/lib/vectorization.py` :

```python
class ComputeVectorization:
    def __init__(
        self,
```

```
        experiment_id: str,
        documents: dict,
        model: Optional[str] = None,
        notify_url: Optional[str] = None,
        tracking_url: Optional[str] = None,
    ):
        self.experiment_id = experiment_id
        self.documents = documents
        self.model = model
        self.notify_url = notify_url
        self.tracking_url = tracking_url
        self.client_id = "default"
        self.imgs = []

    def run_task(self):
        pass

    def check_dataset(self):
        if len(list(self.documents.keys())) == 0:
            return False
        return True

    def task_update(self, event, message=None):
        if self.tracking_url:
            send_update(self.experiment_id, self.tracking_url, event, message)
            return True
        else:
            return False


class LoggedComputeVectorization(LoggingTaskMixin, ComputeVectorization):
    def run_task(self):
        if not self.check_dataset():
            self.print_and_log_warning(f"[task.vectorization] No documents to
                download")
            self.task_update("ERROR", f"[API ERROR] Failed to download documents
                 for vectorization")
            return

        error_list = []
```

```
        try:
            for doc_id, document in self.documents.items():
                self.print_and_log(
                    f"[task.vectorization] Vectorization task triggered for {
                        doc_id} !"
                )
                self.task_update("STARTED")

                self.download_dataset(doc_id, document)
                self.process_inference(doc_id)
                self.send_zip(doc_id)

            self.task_update("SUCCESS", error_list if error_list else None)

        except Exception as e:
            self.print_and_log(f"Error when computing vectorizations", e=e)
            self.task_update("ERROR", "[API ERROR] Vectorization task failed")

    def download_dataset(self, doc_id, document):
        self.print_and_log(
            f"[task.vectorization] Dowloading images...", color="blue"
        )
        for image_id, url in document.items():
            try:
                if not is_downloaded(doc_id, image_id):
                    self.print_and_log(
                        f"[task.vectorization] Downloading image {image_id}"
                    )
                    download_img(url, doc_id, image_id)

            except Exception as e:
                self.print_and_log(
                    f"[task.vectorization] Unable to download image {image_id}", e
                )

    def process_inference(self, doc_id):
        model_folder = Path(MODEL_PATH)
        model_config_path = f"{model_folder}/config_cfg.py"
        epoch = DEFAULT_EPOCHS if self.model is None else self.model
        model_checkpoint_path = f"{model_folder}/checkpoint{epoch}.pth"
        args = SLConfig.fromfile(model_config_path)
```

```
        args.device = 'cuda'
        args.num_select = 200

        corpus_folder = Path(IMG_PATH)
        image_paths = glob.glob(str(corpus_folder / doc_id) + "/*.jpg")
        output_dir = VEC_RESULTS_PATH / doc_id
        os.makedirs(output_dir, exist_ok=True)

        model, criterion, postprocessors = build_model_main(args)
        checkpoint = torch.load(model_checkpoint_path, map_location='cpu')
        model.load_state_dict(checkpoint['model'])
        model.eval()

        args.dataset_file = 'synthetic'
        args.mode = "primitives"
        args.relative = False
        args.common_queries = True
        args.eval = True
        args.coco_path = "data/synthetic_processed"
        args.fix_size = False
        args.batch_size = 1
        args.boxes_only = False
        vslzr = COCOVisualizer()
        id2name = {0: 'line', 1: 'circle', 2: 'arc'}
        primitives_to_show = ['line', 'circle', 'arc']

        torch.cuda.empty_cache()
        transform = T.Compose([
            T.RandomResize([800], max_size=1333),
            T.ToTensor(),
            T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])

        with torch.no_grad():
            for image_path in image_paths:
                try:
                    self.print_and_log(
                        f"[task.vectorization] Processing {image_path}", color="
                            blue"
                    )
                    # Load and process image
```

```
                im_name = os.path.basename(image_path)[:-4]
                image = Image.open(image_path).convert("RGB")
                im_shape = image.size
                input_image, _ = transform(image, None)
                size = torch.Tensor([input_image.shape[1], input_image.shape
                    [2]])

                # Model inference
                output = model.cuda()(input_image[None].cuda())
                output = postprocessors['param'](output, torch.Tensor([[
                    im_shape[1], im_shape[0]]]).cuda(), to_xyxy=False)[0]

                threshold, arc_threshold = 0.3, 0.3
                scores = output['scores']
                labels = output['labels']
                boxes = output['parameters']
                select_mask = ((scores > threshold) & (labels != 2)) | ((
                    scores > arc_threshold) & (labels == 2))
                labels = labels[select_mask]
                boxes = boxes[select_mask]
                scores = scores[select_mask]
                pred_dict = {'parameters': boxes, 'labels': labels, 'scores':
                    scores}
                lines, line_scores, circles, circle_scores, arcs, arc_scores =
                     get_outputs_per_class(pred_dict)

                # Postprocess the outputs
                lines, line_scores = remove_duplicate_lines(lines, im_shape,
                    line_scores)
                lines, line_scores = remove_small_lines(lines, im_shape,
                    line_scores)
                circles, circle_scores = remove_duplicate_circles(circles,
                    im_shape, circle_scores)
                arcs, arc_scores = remove_duplicate_arcs(arcs, im_shape,
                    arc_scores)
                arcs, arc_scores = remove_arcs_on_top_of_circles(arcs, circles
                    , im_shape, arc_scores)
                arcs, arc_scores = remove_arcs_on_top_of_lines(arcs, lines,
                    im_shape, arc_scores)

                # Generate and save SVG
```

```
                self.print_and_log(f"[task.vectorization] Drawing {image_path}
                    ", color="blue")
                #shutil.copy2(image_path, output_dir)
                #décommenter cette ligne si on veut obtenir les images dans le
                     répertoire de sortie
                diagram_name = Path(image_path).stem
                image_name = os.path.basename(image_path)
                lines = lines.reshape(-1, 2, 2)
                arcs = arcs.reshape(-1, 3, 2)

                dwg = svgwrite.Drawing(str(output_dir / f"{diagram_name}.svg")
                    , profile="tiny", size=im_shape)
                dwg.add(dwg.image(href=image_name, insert=(0, 0), size=
                    im_shape))
                dwg = write_svg_dwg(dwg, lines, circles, arcs, show_image=
                    False, image=None)
                dwg.save(pretty=True)

                ET.register_namespace('', "http://www.w3.org/2000/svg")
                ET.register_namespace('xlink', "http://www.w3.org/1999/xlink")
                ET.register_namespace('sodipodi', "http://sodipodi.sourceforge
                    .net/DTD/sodipodi-0.dtd")
                ET.register_namespace('inkscape', "http://www.inkscape.org/
                    namespaces/inkscape")

                file_name = output_dir / f"{diagram_name}.svg"
                tree = ET.parse(file_name)
                root = tree.getroot()

                root.set('xmlns:inkscape', 'http://www.inkscape.org/namespaces
                    /inkscape')
                root.set('xmlns:sodipodi', 'http://sodipodi.sourceforge.net/
                    DTD/sodipodi-0.dtd')
                root.set('inkscape:version', '1.3 (0e150ed, 2023-07-21)')

                arc_regex = re.compile(r'[aA]')
                for path in root.findall('{http://www.w3.org/2000/svg}path'):
                    d = path.get('d', '')
                    if arc_regex.search(d):
                        path.set('sodipodi:type', 'arc')
                        path.set('sodipodi:arc-type', 'arc')
```

```
                        path_parsed = parse_path(d)
                        for e in path_parsed:
                            if isinstance(e, Line):
                                continue
                            elif isinstance(e, Arc):
                                center, radius, start_angle, end_angle, p0, p1 =
                                    get_arc_param([e])
                                path.set('sodipodi:cx', f'{center[0]}')
                                path.set('sodipodi:cy', f'{center[1]}')
                                path.set('sodipodi:rx', f'{radius}')
                                path.set('sodipodi:ry', f'{radius}')
                                path.set('sodipodi:start', f'{start_angle}')
                                path.set('sodipodi:end', f'{end_angle}')

                    tree.write(file_name, xml_declaration=True)

                    self.print_and_log(f"[task.vectorization] SVG for {image_path}
                        drawn", color="yellow")

                except Exception as e:
                    self.print_and_log(f"[task.vectorization] Failed to process {
                        image_path}", e)

            self.print_and_log(f"[task.vectorization] Task over", color="yellow"
                )

    def send_zip(self, doc_id):
        """
        Zip le répertoire et envoie ce répertoire via POST à l'URL spécifiée.
        """
        try:
            output_dir = VEC_RESULTS_PATH / doc_id
            zip_path = output_dir / f"{doc_id}.zip"
            self.print_and_log(f"[task.vectorization] Zipping directory {
                output_dir}", color="blue")

            zip_directory(output_dir, zip_path)
            self.print_and_log(f"[task.vectorization] Sending zip {zip_path} to
                {self.notify_url}", color="blue")

            with open(zip_path, 'rb') as zip_file:
```

```
            response = requests.post(
                url=self.notify_url,
                files={
                    "file": zip_file,
                },
                data={
                    "experiment_id": self.experiment_id,
                    "model": self.model,
                },
            )

        if response.status_code == 200:
            self.print_and_log(f"[task.vectorization] Zip sent successfully
                to {self.notify_url}", color="yellow")
        else:
            self.print_and_log(f"[task.vectorization] Failed to send zip to {
                self.notify_url}. Status code: {response.status_code}", color
                ="red")

    except Exception as e:
        self.print_and_log(f"[task.vectorization] Failed to zip and send
            directory {output_dir}", e)
```

Listing B.9 – Classe `ComputeVectorization`.

B.5 Réception des résultats dans AIKON

Pour que les vectorisations soient retournées de l'API à AIKON après l'inférence du modèle, un *endpoint* est créé, dont le routage avec la bonne URL est définie dans le fichier `urls.py` du module.

```
path(
    f"{APP_NAME}/get-vectorization",
    receive_vectorization,
    name="get-vectorization",
),
```

Listing B.10 – Routage de l'*endpoint* pour la réception de vectorisations.

Cet *endpoint* appelle la fonction `receive_vectorization` du fichier `views.py` du module `vectorization`, qui reçoit un fichier .ZIP via une POST request de l'API. Grâce à une fonction utilitaire du fichier `utils.py` (`save_svg_files`), l'archive est extraite, les

fichiers sont lus et écris dans le dossier `mediafiles`. Si les fichiers existent déjà, ils sont écrasés et réécrits.

```
@csrf_exempt
def receive_vectorization(request):
    """
    Endpoint to receive a ZIP file containing SVG files and save them to the
        media directory.
    """
    if "file" not in request.FILES:
        return JsonResponse({"error": "No file received"}, status=400)

    file = request.FILES["file"]
    # treatment_id = request.DATA["experiment_id"]

    if file.name == "":
        return JsonResponse({"error": "File name is empty"}, status=400)

    if file and file.name.endswith(".zip"):
        try:
            temp_zip_path = default_storage.save("temp.zip", file)
            temp_zip_file = default_storage.path(temp_zip_path)

            save_svg_files(temp_zip_file)
            default_storage.delete(temp_zip_path)

            return JsonResponse(
                {"message": "Files successfully uploaded and extracted"}, status
                    =200
            )
        except Exception as e:
            return JsonResponse({"error": str(e)}, status=500)
    else:
        return JsonResponse({"error": "Unsupported file type"}, status=400)
```

Listing B.11 – *Endpoint* `receive_vectorization` pour le retour des vectorisations dans l'application.

Fonction utilitaire pour traiter le contenu du ZIP :

```
def save_svg_files(zip_file):
    """
    Extracts SVG files from a ZIP file and saves them to the SVG_PATH directory
        .
```

```
    """
    # Vérifie si le répertoire SVG_PATH existe, sinon le crée
    if not os.path.exists(SVG_PATH):
        os.makedirs(SVG_PATH)

    try:
        with zipfile.ZipFile(zip_file, "r") as zip_ref:
            for file_info in zip_ref.infolist():
                # TODO do not save jpg file
                # Vérifie si le fichier est un fichier SVG
                if file_info.filename.endswith(".svg"):
                    file_path = os.path.join(
                        SVG_PATH, os.path.basename(file_info.filename)
                    )

                    # Supprime le fichier existant s'il y en a un
                    if os.path.exists(file_path):
                        os.remove(file_path)

                    # Extrait le fichier SVG et l'écrit dans le répertoire
                        spécifié
                    with zip_ref.open(file_info) as svg_file:
                        with open(file_path, "wb") as output_file:
                            output_file.write(svg_file.read())
    except Exception as e:
        log(f"[save_svg_files] Error when extracting SVG files from ZIP file", e
            )
        return False
    return True
```

Listing B.12 – Fonction utilitaire pour le traitement du contenu de l'archive.