

# Project Reconnaissance sémantique d'images et stockage RDF

**GitHub :** [github.com/Siloya/SemanticImageClassification-YOLO-RDF-ConceptNet-SPARQL](https://github.com/Siloya/SemanticImageClassification-YOLO-RDF-ConceptNet-SPARQL)

## Introduction

Ce projet vise à combiner des technologies de vision par ordinateur, de représentation de la connaissance et de visualisation de graphe pour créer un système intelligent d'identification sémantique d'objets dans des images. Nous avons utilisé **YOLOv8** pour détecter les objets dans les images, **ConceptNet** pour enrichir ces objets avec des relations sémantiques, et **Blazegraph** pour stocker ces relations sous forme de triplets RDF interrogeables via **SPARQL**. L'interface web permet ensuite de visualiser les graphes RDF générés.

## 1 Interface Web avec Flask

Pour orchestrer les différentes étapes du traitement ,de l'envoi d'images jusqu'à la visualisation des graphes RDF , nous avons développé une application web avec le framework Flask en Python. Cette API permet de recevoir les images en entrée, de déclencher la détection avec YOLO, d'enrichir les objets avec des relations issues de ConceptNet, de générer les triplets RDF, puis de les insérer automatiquement dans Blazegraph. Enfin, Flask renvoie les résultats vers l'interface web, qui affiche dynamiquement les objets détectés, les relations sémantiques et les graphes RDF associés.

## 2 Détection d'objets avec YOLOv8

### 2.1 Présentation de YOLO

**YOLO (You Only Look Once)** est un modèle de détection d'objets en temps réel développé par Ultralytics. Nous avons utilisé la version YOLOv8n , optimisée pour de bonnes performances avec une empreinte légère.

## 3 Détection d'objets avec YOLOv8

### 3.1 Présentation de YOLO

**YOLO (You Only Look Once)** est un modèle de détection d'objets en temps réel développé par Ultralytics. Nous avons utilisé la version YOLOv8n , optimisée pour de

bonnes performances avec une empreinte légère.

## 3.2 Chargement du modèle

Le modèle est initialisé une seule fois au lancement du serveur Flask à l'aide de la bibliothèque `ultralytics` :

```
from ultralytics import YOLO
model = YOLO('yolov8n.pt')
```

Le fichier du modèle `yolov8n.pt` est placé à la racine du projet. Ce modèle est ensuite utilisé dans la route `/detect` pour analyser les images reçues :

```
results = model(filepath)
detections = []
for result in results[0].boxes.data:
    class_id = int(result[5].item())
    confidence = float(result[4].item())
    class_name = class_names[class_id] if class_id < len(class_names) else "unknown"
    detections.append({"class_name": class_name, "confidence": confidence})
```

Chaque détection contient : le nom de la classe (par exemple `car`, `person`), et une valeur de confiance.

## 3.3 Chargement du fichier `COCO.names`

Pour faire correspondre les identifiants de classes numériques aux noms lisibles, nous avons utilisé le fichier `coco.names` contenant les 80 classes du dataset COCO. Ce fichier est chargé une fois au lancement :

```
with open("coco.names", "r") as f:
    class_names = [line.strip() for line in f.readlines()]
```

Par exemple, l'identifiant 0 correspond à `person`, 1 à `bicycle`, etc. Cela permet d'interpréter les résultats fournis par YOLO.

# 4 Enrichissement sémantique avec ConceptNet

## 4.1 Interrogation de ConceptNet

Pour chaque objet détecté par YOLOv8, nous avons utilisé l'API publique de **ConceptNet** afin de récupérer des relations sémantiques. L'URL de base utilisée pour les appels était de la forme :

```
http://api.conceptnet.io/c/en/<mot>
```

Le code suivant permet d'interroger ConceptNet pour un mot donné :

```
def query_conceptnet(word):
    url = f"http://api.conceptnet.io/c/en/{word}"
    response = requests.get(url)
    if response.status_code == 200:
        return response.json()
    else:
        return None
```

## 4.2 Extraction des relations RDF

Une fois les données JSON reçues, nous extrayons uniquement les relations sémantiques importantes telles que `IsA`, `UsedFor`, `CapableOf`, etc., via la fonction suivante :

```
def extract_relations(concept_data):
    if not concept_data:
        return []
    relations = []
    for edge in concept_data.get('edges', []):
        relation = edge['rel']['label']
        start = edge['start']['label']
        end = edge['end']['label']
        relations.append((start, relation, end))
    return relations
```

## 5 Génération de graphes RDF avec RDFLib

### 5.1 Création des triplets RDF

Nous avons défini une fonction `generate_rdf_triplets()` qui prend les détections YOLO et les relations ConceptNet, et les encode sous forme de triplets RDF avec la bibliothèque `rdflib` :

```
def generate_rdf_triplets(detections, conceptnet_results):
    g = Graph()
    for detection in detections:
        subject = URIRef(f"http://example.org/{detection['class_name']}.replace('
', '_')}")
        if detection['class_name'] in conceptnet_results:
            for rel in conceptnet_results[detection['class_name']]:
                predicate = URIRef(f"http://example.org/{rel[1]}")
                obj = Literal(rel[2])
                g.add((subject, predicate, obj))
    return g
```

Le fichier RDF est ensuite sauvegardé dans le dossier `output/`, avec un nom unique par image :

```
rdf_file = os.path.join("output", f"rdf_output_{file.filename}.rdf")
rdf_graph.serialize(rdf_file, format="turtle")
```

## 6 Résultat RDF généré à partir d'une image

Avant de générer et de sérialiser le graphe RDF à l'aide de `rdflib`, nous avons testé la structure des triplets RDF générés à partir des objets détectés par YOLOv8 et enrichis sémantiquement via ConceptNet.

Les triplets RDF suivants illustrent un extrait des fichiers générés automatiquement dans le dossier `output/`, pour les images `image1.jpeg` et `image2.jpeg`.

Ces fichiers montrent clairement la structure des triplets RDF :

```

output > 🦋 rdf_output_image2.jpeg.rdf
1  @prefix ns1: <http://example.org/> .
2
3  ns1:book ns1:AtLocation "a bookshelf",
4      "a classroom",
5      "the shelf" ;
6      ns1:CreatedBy "a writer" ;
7      ns1:HasA "knowledge" ;
8      ns1:IsA "a book" ;
9      ns1:MadeOf "Paper" ;
10     ns1:PartOf "a book",
11         "library" ;
12     ns1:RelatedTo "book",
13         "pages",
14         "reading" ;
15     ns1:UsedFor "learning" .
16
17  ns1:keyboard ns1:AtLocation "a keyboard",
18      "your desk" ;
19      ns1:DerivedFrom "keyboard" ;
20      ns1:HasA "a keyboard",
21          "keys" ;
22      ns1:IsA "a keyboard",
23          "keyboard" ;
24      ns1:PartOf "a computer",
25          "a piano",
26          "computer",
27          "typewriter" ;
28      ns1:Synonym "keyboard",
29          "tastatura" ;

```

FIGURE 1 – Extrait du fichier RDF généré

```
output > 📡 rdf_output_image1.jpeg.rdf
1  @prefix ns1: <http://example.org/> .
2
3  ns1:book ns1:AtLocation "a bookshelf",
4      "a classroom",
5      "the shelf" ;
6      ns1:CreatedBy "a writer" ;
7      ns1:HasA "knowledge" ;
8      ns1:IsA "a book" ;
9      ns1:MadeOf "Paper" ;
10     ns1:PartOf "a book",
11         "library" ;
12     ns1:RelatedTo "book",
13         "pages",
14         "reading" ;
15     ns1:UsedFor "learning" .
16
17 ns1:laptop ns1:RelatedTo "desktop",
18     "laptop" ;
19     ns1:Synonym "laptop",
20     "laptop computer" .
21
22
```

FIGURE 2 – Extrait du fichier RDF généré

- **Sujet** : l'objet détecté par YOLO (ex. book, keyboard, laptop).
- **Prédicat** : le type de relation sémantique (ex. IsA, HasA, UsedFor, RelatedTo).
- **Objet** : l'entité liée selon ConceptNet (ex. knowledge, pages, typing).

Chaque ressource RDF est construite avec un préfixe `ns1` pointant vers `http://example.org/`, et sérialisée au format Turtle.

## 7 Stockage des triplets RDF avec Blazegraph

### 7.1 Création dynamique de namespace

Avant insertion, nous avons défini une fonction qui s'assure que le namespace ciblé existe sur Blazegraph, sinon elle le crée :

```
def create_namespace_if_not_exists(namespace):
    endpoint_url = f"http://localhost:9999/blazegraph/namespace/{namespace}/sparql"
    response = requests.get(endpoint_url)
    if response.status_code == 200:
        return
    create_ns_url = "http://localhost:9999/blazegraph/namespace"
    xml_data = f"""<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="com.bigdata.rdf.sail.namespace">{namespace}</entry>
</properties>"""
    headers = {"Content-Type": "application/xml"}
    requests.post(create_ns_url, data=xml_data, headers=headers)
```

### 7.2 Insertion automatique dans Blazegraph

Une fois le graphe RDF généré, nous l'insérons dans le triplestore Blazegraph via un endpoint SPARQL spécifique, avec la méthode POST :

```
def insert_rdf_to_blazegraph(graph, namespace_name):
    endpoint_url = f"http://localhost:9999/blazegraph/namespace/{namespace_name}/sparql"
    sparql = SPARQLWrapper(endpoint_url)
    sparql.setMethod(POST)
    rdf_data = graph.serialize(format='nt')
    sparql.setQuery(f"INSERT DATA {{ {rdf_data} }}")
    sparql.query()
```

## 8 Interrogation et Visualisation des Données dans Blazegraph

### 8.1 Présentation de Blazegraph

**Blazegraph** est un moteur de triplestore RDF open-source, qui permet de stocker, interroger et manipuler des graphes de connaissances via le langage SPARQL. Il expose

une interface Web très intuitive et un endpoint SPARQL disponible par défaut à l'adresse suivante :

`http://localhost:9999/blazegraph`

Dans notre projet, Blazegraph joue le rôle de base de données RDF centrale. Chaque image traitée génère un fichier RDF contenant des triplets représentant les objets détectés et leurs relations sémantiques. Ces fichiers sont ensuite insérés automatiquement dans Blazegraph sous un namespace dédié.

## 8.2 Exécution de Requêtes SPARQL

Une fois les triplets RDF insérés, nous pouvons les interroger à l'aide de requêtes SPARQL. La Figure 3 montre une requête visant à obtenir toutes les relations (prédicats et objets) associées à l'entité `laptop` :

```
SELECT ?predicate ?object
WHERE {
  <http://example.org/laptop> ?predicate ?object
}
```

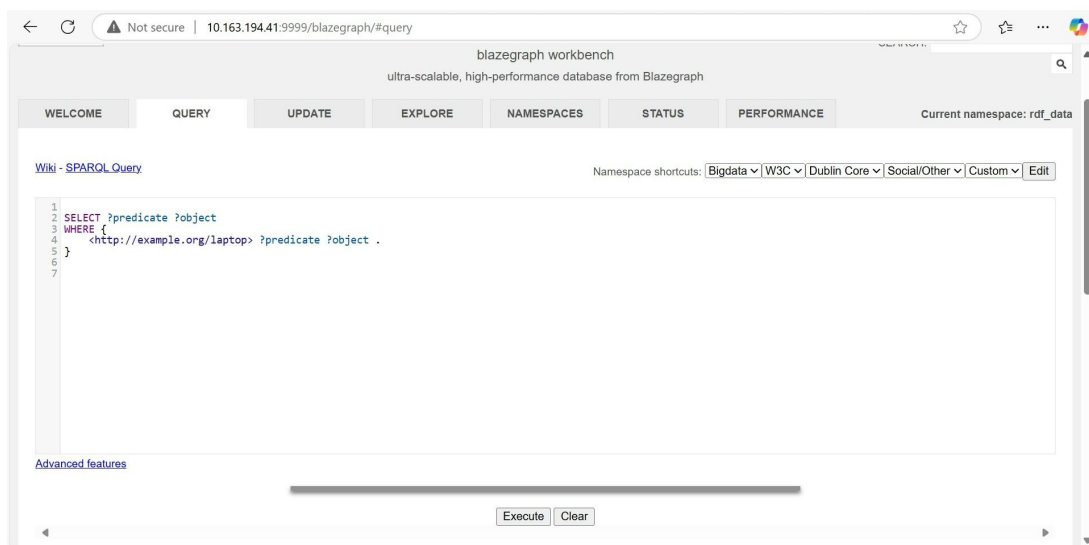
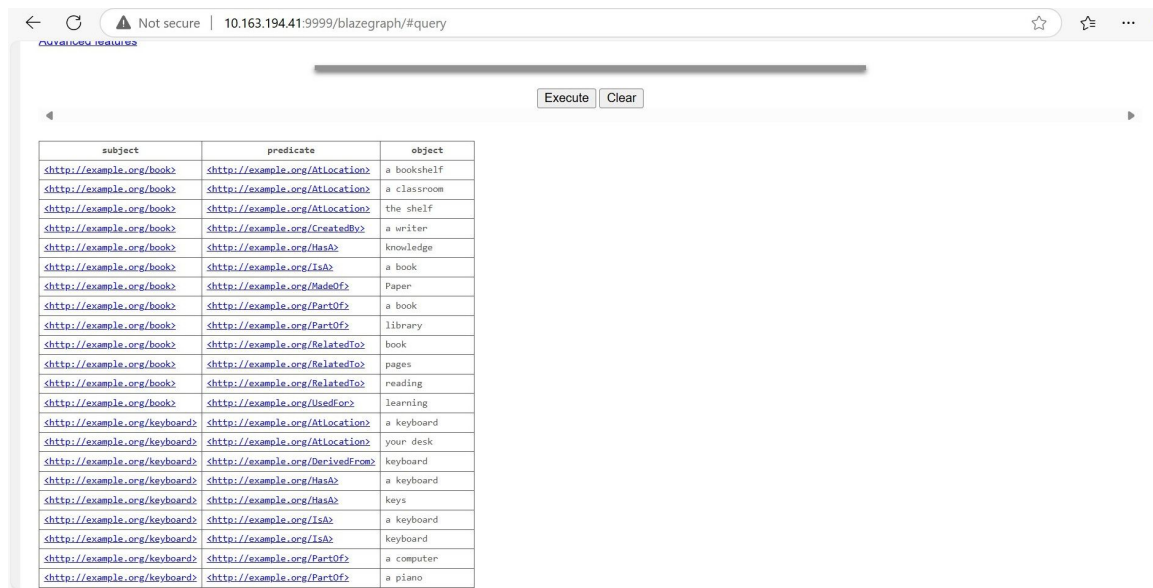


FIGURE 3 – Requête SPARQL exécutée sur l'interface Blazegraph

## 8.3 Résultat de la Requête

La Figure 4 montre le résultat de cette requête SPARQL sous forme tabulaire. On y retrouve plusieurs triplets RDF insérés précédemment, tels que :

- `<http://example.org/book> RelatedTo pages`
- `<http://example.org/keyboard> UsedFor typing`
- `<http://example.org/book> AtLocation a bookshelf`



subject	predicate	object
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/AtLocation">http://example.org/AtLocation</a>	a bookshelf
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/AtLocation">http://example.org/AtLocation</a>	a classroom
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/AtLocation">http://example.org/AtLocation</a>	the shelf
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/CreatedBy">http://example.org/CreatedBy</a>	a writer
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/HasA">http://example.org/HasA</a>	knowledge
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/IsA">http://example.org/IsA</a>	a book
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/MadeOf">http://example.org/MadeOf</a>	Paper
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/PartOf">http://example.org/PartOf</a>	a book
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/PartOf">http://example.org/PartOf</a>	library
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/RelatedTo">http://example.org/RelatedTo</a>	book
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/RelatedTo">http://example.org/RelatedTo</a>	pages
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/RelatedTo">http://example.org/RelatedTo</a>	reading
<a href="http://example.org/book">http://example.org/book</a>	<a href="http://example.org/UsedFor">http://example.org/UsedFor</a>	learning
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/AtLocation">http://example.org/AtLocation</a>	a keyboard
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/AtLocation">http://example.org/AtLocation</a>	your desk
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/DerivedFrom">http://example.org/DerivedFrom</a>	keyboard
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/HasA">http://example.org/HasA</a>	a keyboard
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/HasA">http://example.org/HasA</a>	keys
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/IsA">http://example.org/IsA</a>	a keyboard
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/IsA">http://example.org/IsA</a>	keyboard
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/PartOf">http://example.org/PartOf</a>	a computer
<a href="http://example.org/keyboard">http://example.org/keyboard</a>	<a href="http://example.org/PartOf">http://example.org/PartOf</a>	a piano

FIGURE 4 – Résultats RDF de la requête SPARQL affichés dans Blazegraph

## 8.4 Utilité pour l'Analyse Sémantique

Grâce à cette interface, l'utilisateur peut explorer dynamiquement les relations extraites de ConceptNet. Cela permet de valider la cohérence des graphes RDF générés à partir d'images, de rechercher des objets connexes, ou encore de vérifier des propriétés sémantiques entre objets (comme *IsA*, *AtLocation*, ou *UsedFor*).

## 9 Requête SPARQL via l'interface Web

L'application intègre une interface utilisateur simple permettant d'exécuter des requêtes **SPARQL** directement depuis le navigateur. L'utilisateur peut saisir librement ses requêtes dans un champ de texte dédié et interroger dynamiquement les triplets RDF stockés dans **Blazegraph**.

Par exemple, la requête suivante permet de récupérer les 20 premiers triplets RDF stockés :

```
SELECT ?subject ?predicate ?object WHERE {
  ?subject ?predicate ?object
} LIMIT 20
```

Les résultats sont ensuite affichés dans le site web. Cette approche facilite l'exploration manuelle des relations sémantiques extraites depuis ConceptNet et insérées dans Blazegraph.

## 10 Visualisation du graphe RDF avec D3.js

### 10.1 Génération et rendu du graphe

La bibliothèque **D3.js** a été utilisée pour représenter graphiquement les triplets RDF générés à partir des détections YOLO et enrichis par ConceptNet. Une requête SPARQL



est automatiquement envoyée au backend Flask, qui retourne les triplets RDF. Ceux-ci sont ensuite convertis en paires de nœuds et de liens.

## 10.2 Simulation physique avec D3.js

Pour disposer automatiquement les entités RDF de manière lisible et éviter les chevauchements, nous avons configuré une simulation D3.js combinant plusieurs forces :

```
const simulation = d3.forceSimulation(nodes)
  .force("link", d3.forceLink(links).id(d => d.id).distance(120))
  .force("charge", d3.forceManyBody().strength(-500))
  .force("center", d3.forceCenter(width / 2, height / 2))
  .force("collision", d3.forceCollide().radius(30));
```

## 10.3 Affichage des nœuds et des liens

Chaque lien RDF est représenté par un élément `<line>`, tandis que chaque entité (sujet ou objet) est représentée par un cercle et une étiquette textuelle :

```
node.append("circle")
  .attr("r", 10)
  .style("fill", "#4A90E2");

node.append("text")
  .attr("dx", 12)
  .attr("dy", 4)
  .style("font-size", "14px")
  .text(d => d.id);
```

# 11 Installation et mise en place du projet

Cette section décrit les étapes nécessaires pour cloner, configurer et exécuter le projet de reconnaissance sémantique d'images basé sur YOLOv8, ConceptNet et Blazegraph.

## 11.1 1. Clonage du dépôt GitHub

Le projet est disponible sur GitHub. Pour le récupérer localement :

```
git clone https://github.com/Siloya/SemanticImageClassification-YOLO-RDF-
  ConceptNet-SPARQL.git
cd SemanticImageClassification-YOLO-RDF-ConceptNet-SPARQL
```

## 11.2 2. Installation des dépendances Python

Une fois dans le répertoire du projet, les dépendances nécessaires sont installées via pip :

```
pip install -r requirements.txt
```

### 11.3 3. Lancement de Blazegraph

Avant de démarrer le backend, Blazegraph doit être lancé pour fournir un endpoint SPARQL local. Exécuter la commande suivante dans le répertoire contenant `blazegraph.jar` :

```
java -server -Xmx4g -jar blazegraph.jar
```

Cela démarre Blazegraph sur `http://localhost:9999/blazegraph`.

### 11.4 4. Démarrage du backend Flask

Le backend est développé en Flask. Une fois Blazegraph lancé, le serveur peut être démarré par :

```
python app.py
```

L'interface utilisateur sera accessible à l'adresse :

```
http://127.0.0.1:5000
```

## Conclusion

Ce projet démontre l'intégration de la détection d'objets, de l'enrichissement sémantique et de la représentation de connaissance RDF avec visualisation interactive. Il combine efficacement **YOLO**, **ConceptNet**, **SPARQL** et **D3.js** pour produire une interface pédagogique et performante.

## 12 Aperçu du projet

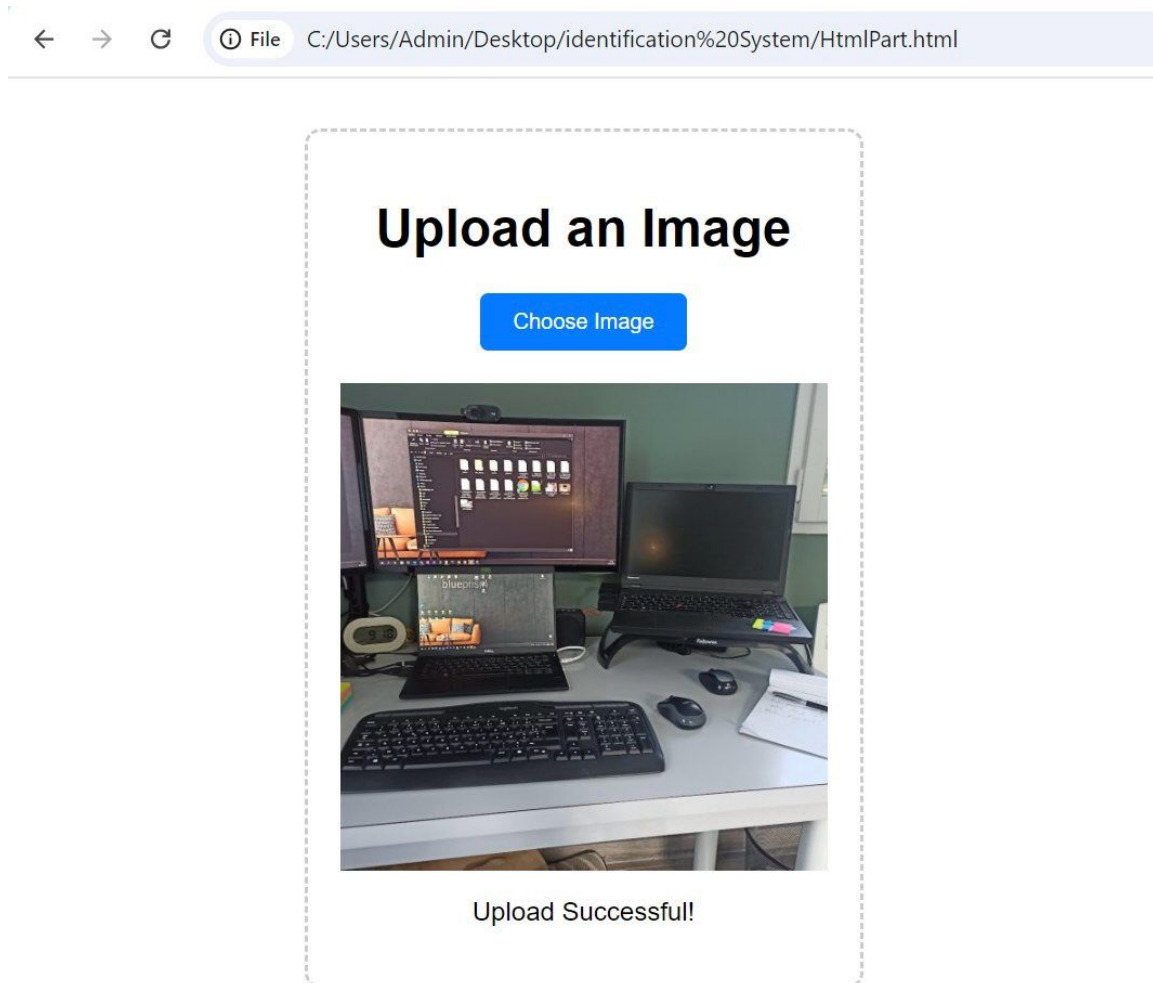


FIGURE 5 – télécharger des images

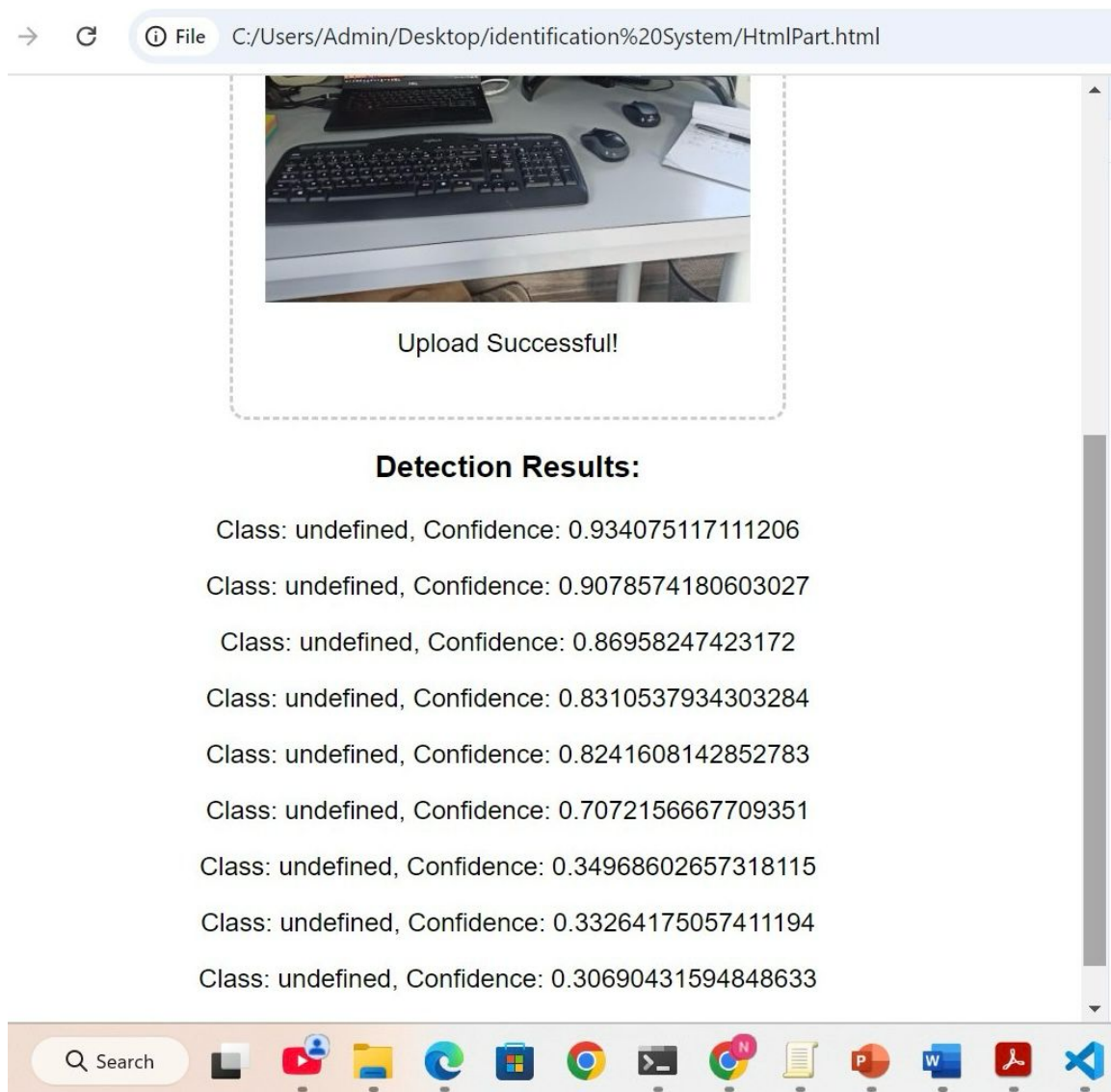


FIGURE 6 – Detection des objets



FIGURE 7 – Relations sémantiques extraites via ConceptNet

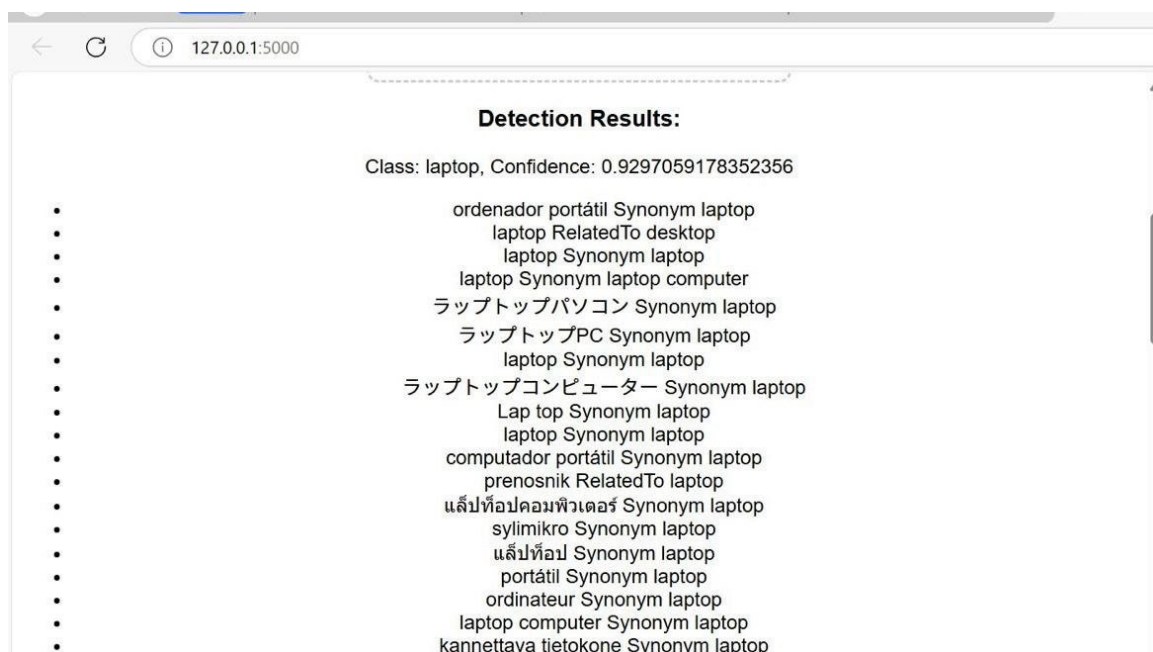


FIGURE 8 – Relations sémantiques extraites via ConceptNet

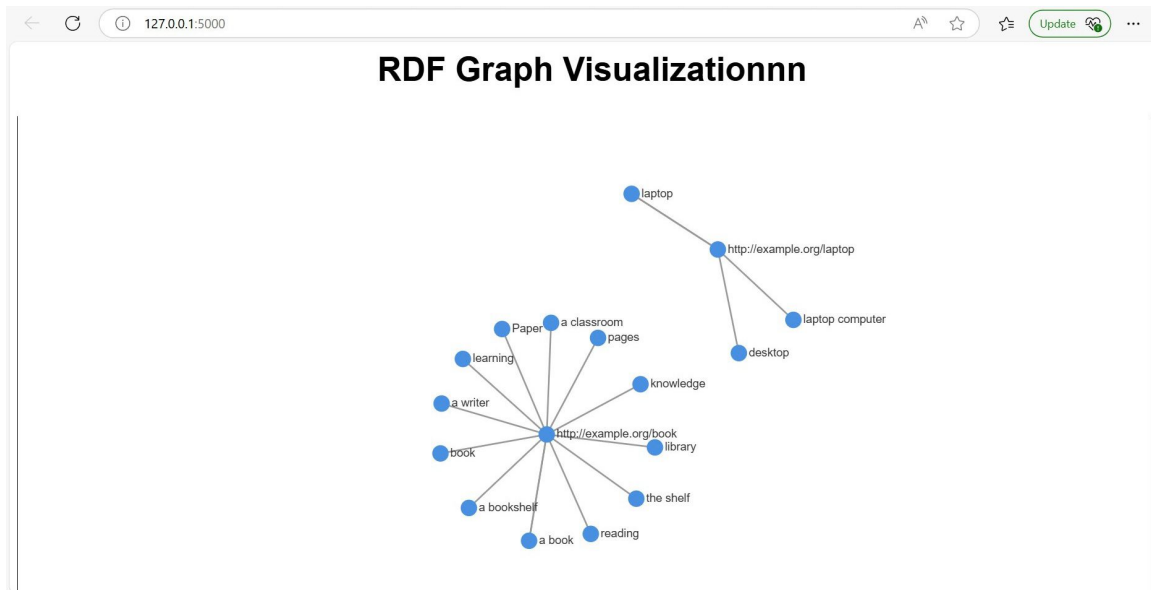


FIGURE 9 – Exemple de graphe RDF généré

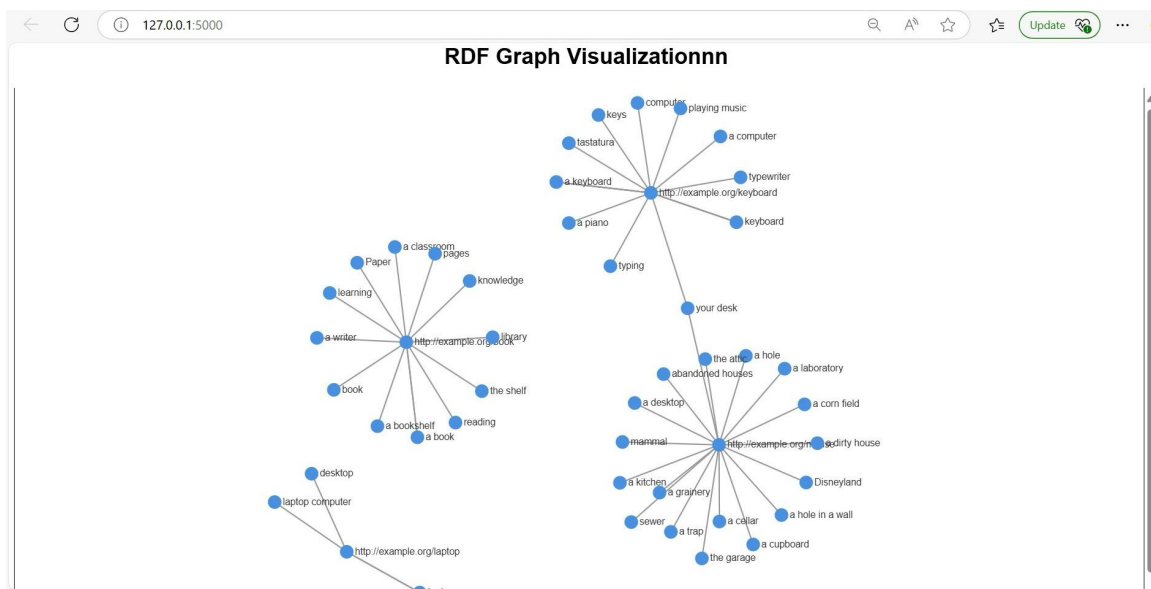


FIGURE 10 – RDF graphe visualization