Projet INGE 2 : Majeure IA
Computer Vision pour la lecture automatique de factures

Thomas KUSNIEREK, Benjamin SZUREK, Baptiste BEHR
Encadrant : Wajd MESKINI

21 mai 2025

Table des matières

Table des figures

Liste des tableaux

1 Remerciements

Nous tenons à exprimer notre profonde gratitude à M. Wajd Meskini et M. Mounir Lahlouh pour leur accompagnement et leurs conseils tout au long de ce projet.

Nous remercions également tous ceux qui ont pu participer à notre projet que ce soit de près ou de loin.

2 Introduction

2.1 Contexte

Dans un monde de plus en plus numérique, la virtualisation et la lecture de documents représentent un enjeu majeur pour de nombreuses entreprises. Faciliter les démarches administratives, comme la lecture automatisée de factures, s'inscrit pleinement dans cette dynamique. C'est dans cette optique que notre cabinet de conseil accompagne un client qui souhaite extraire automatiquement les frais totaux à partir de documents financiers.

Pour des raisons réglementaires, le client ne peut pas fournir de données issues des factures. En réponse à cette contrainte, nous exploiterons des images de reçus de paiement comme source de données. En effet, ces derniers présentent des structures proches à celles de factures. Ce choix nous permet de simuler un environnement représentatif tout en respectant les exigences de confidentialité imposées par le client.

2.2 Problématiques

Nous avons opté avec l'approche suivante : nous appuyer sur des images de reçus de paiement, dont la structure présente des similitudes avec celle de facture, afin de développer un système d'extraction. Ces reçus serviront de terrain d'expérimentation pour concevoir, entraîner et tester notre solution. Ce projet s'articule ainsi autour de plusieurs étapes clés :

- le traitement des images pour optimiser la reconnaissance de texte
- l'extraction ciblée du montant total
- l'application du système sur un jeu de données plus large

Dans ce contexte, une question centrale émerge :

Comment concevoir un système d'extraction automatique du montant total d'une facture, tout en garantissant sa fiabilité et reproductibilité ?

2.3 Nos Objectifs

Notre objectif est d'obtenir les informations essentielles d'une facture. Nous consacrerons une plus grande importance sur le montant total de celle-ci.
Le projet sera considéré comme réussi si notre extraction automatique arrive à détecter le montant total d'une facture avec une précision supérieure à 90%. Nous utiliserons 3 types de modèles : en premier, un modèle d'extraction via les bibliothèques python OpenCV et PyTesseract, puis un modèle pré-entraîné provenant de la plateforme Hugging Face et enfin un modèle utilisant le Large Language Model Ollama.

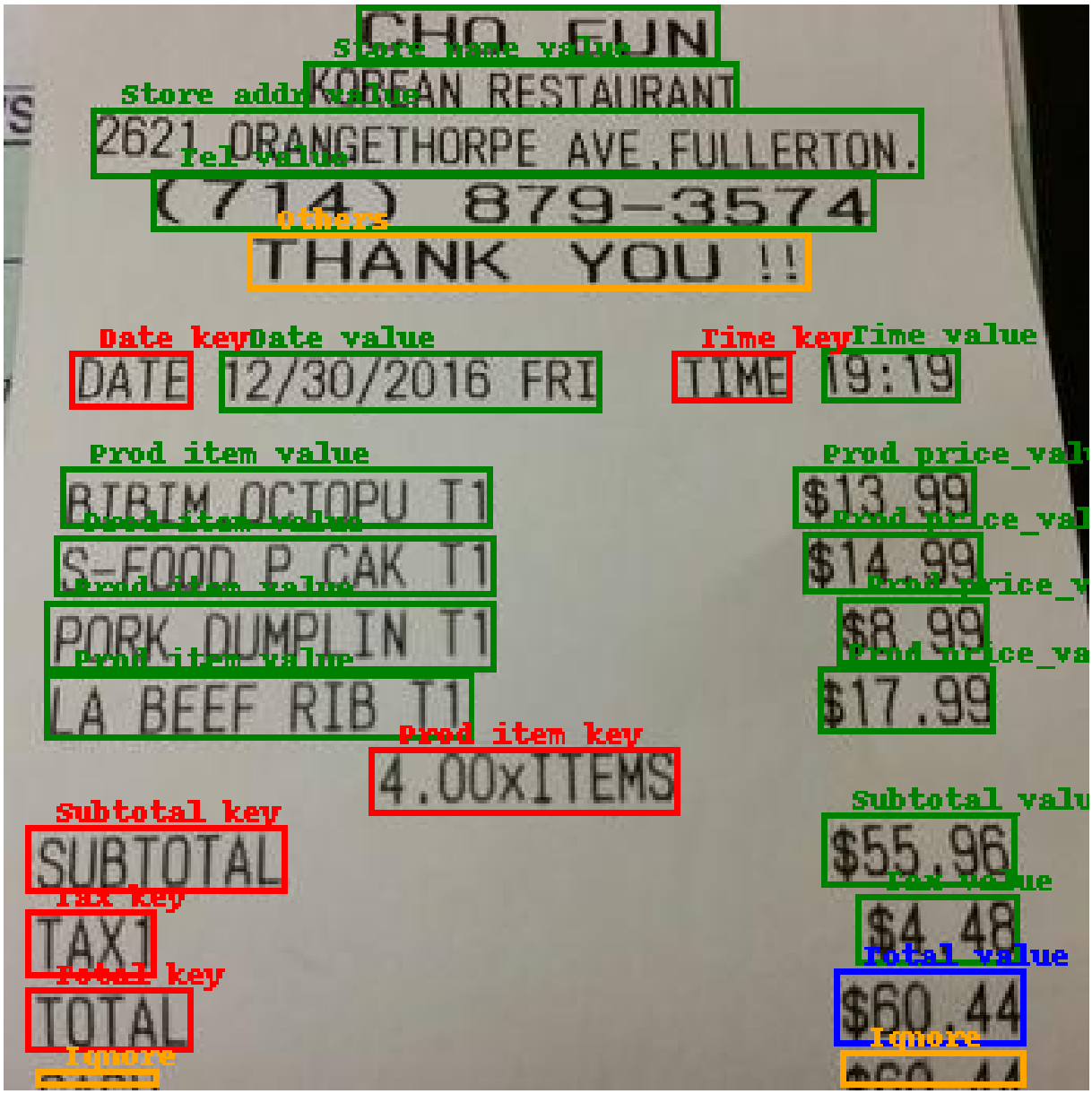

FIGURE 1 – Résultat du projet via le modèle Hugging Face

3 État de l'art

3.1 Bibliothèque classique de python

Les librairies classiques de python sont les suivantes :

— NumPy (Numerical Python) : permet d'effectuer des calculs numériques avec des objets "tableau" de plusieurs dimensions en Python (addition, multiplication de matrice,...). NumPy est la base dans d'autres bibliothèques tels que Pandas ou scikit-learn.

FIGURE 2 – Logo Numpy

— Pandas : permet d'analyser et manipuler les données plus facilement (filtrage et tri de dataframe). Cette bibliothèque utilise deux structures principales : la Series (vecteur unidimensionnel) et le DataFrame (tableau bidimensionnel avec axes étiquetés).

FIGURE 3 – Logo Pandas

— Scikit-image : permet le traitement d'image (importation et affichage d'image). Scikit-image fournit des outils de manipulation, d'analyse et de transformation d'images multi-dimensionnelles.

FIGURE 4 – Logo Scikit-learn

3.2 OpenCV

OpenCV permet le traitement d'image centrées sur la vision d'ordinateur (Computer Vision) en temps réel. (reconnaissance de formes, manipulation de l'image). Cette bibliothèque propose de nombreuses fonctionnalités tel que la lecture et l'écriture de fichiers multimédias, filtrage spatial, détection de contours, détection d'objets (visages, corps,...), suivi de mouvement, reconnaissance de formes, transformations géométriques, ... *OpenCV* est reconnu pour sa performance, sa robustesse et son intégration dans des systèmes complexes.
Nous avons utilisé énormément *OpenCV* pour toutes les visualisations de nos images, des filtres sur les images, de la reconnaissance de forme (de nos factures notamment qui sont rectangulaires), de la création de case de détection de texte. Il s'agit donc de notre bibliothèque principale.

FIGURE 5 – Logo OpenCV

3.3 PyTesseract

PyTesseract permet le traitement de texte imprimé ou d'images. Cette bibliothèque est un wrapper Python du moteur de reconnaissance optique Tesseract OCR . Un wrapper permet la compatibilité et l'interopérabilité entre des structures logicielles [1]. *PyTesseract* peut ainsi extraire du texte à partir d'image, ce qui facile l'automatisation de tâche comme la lecture de documents scannés, de plaques d'immatriculation, ou dans notre cas, de factures.

FIGURE 6 – Logo PyTesseract

3.4 Hugging Face et modèle de Hugging Face basé sur LayoutLM-v3

Hugging Face est une plateforme open source qui permet à ses utilisateurs de créer, utiliser, partager des modèles ou des projets de machine learning, d'accéder à de larges datasets et de créer des démos interactives [6]. L'entreprise fondée en 2016 est devenue incontournable et est célèbre pour sa bibliothèque *Transformers*.

Transformers est une bibliothèque très vaste pouvant analyser du texte, du sons, et bien sûr d'image. Le Vision Transformer (ViT) permet ainsi l'analyse d'une image par zone et de la découper en "patches" pour une analyse et compréhension plus profonde du contenu visuel. Cette approche facilite la détection d'objets, de reconnaissance de visages et d'analyse de scènes complexes [7].

Hugging Face prend en charge des frameworks populaires tel que *PyTorch* et *TensorFlow*. Un framework est ensemble d'outils ou logiciels facilitant le développement. L'intérêt principal d'un framework est de donner une structure de base à notre code pour éviter de partir de zéro (build from scratch). [8]

Nous avons cherché et trouvé un projet de reconnaissance facture sur la plateforme *Hugging Face*. Nous utiliserons ce modèle pour repérer plus précisément des informations sur nos factures, et nous le récupérerons avec le framework *PyTorch* pour avoir le processeur et le modèle. Ce dernier a été entrainé via un multimodale *Transformers* de Microsoft : "LayoutLMv3-base" et a pour but de prédire la position : du nom de la personne qui facture, de l'adresse de la personne qui facture, du code postal de la personne qui facture, du jour de facturation, de Taxe sur les Produits et Services (TPS), le numéro de facturation, le sous-total et le total [9]. Le modèle n'extrait pas les valeurs d'une facture. Il ne sert qu'à déterminer, ou prédire si le modèle n'arrive pas à trouver les valeurs, la position du prix total sur une facture. Cela sera à nous de l'extraire via *PyTesseract*.

FIGURE 7 – Logo plateforme Hugging Face

3.5 LayoutLM

Dans notre précédent paragraphe, nous avions repris un modèle pré-entraînés utilisant LayoutLMv3 sur la plateforme Hugging Face. Revenons un peu sur la construction d'un tel modèle. Le premier LayoutLM avait pour principal but de lier la lecture de texte et l'architecture de l'image (Layout). En effet, précédemment, les applications négligeaient la forme et le style d'information d'une image, ce que vient corriger le premier LayoutLM. [10]

La seconde version LayoutLMv2 renforce le premier modèle avec une nouveau architecture et l'alignement entre le texte et l'image pour faire correspondre les tâches de prétraitement. Il fait également attention à sa propre architecture Transformer, ce qui lui permet de comprendre les relations entre les différents blocs de textes. [11]

Enfin, la troisième version LayoutLMv3 simplifie la seconde version. La plupart des modèles utilisent un langage masqué ("causal mask") sur l'image pour apprendre les modalités du texte. Cependant, le pré-entraînement du langage masqué et celui de l'image est différent, ce qui complique l'apprentissage multimodale. Cette version unifie le texte et l'image. [12]

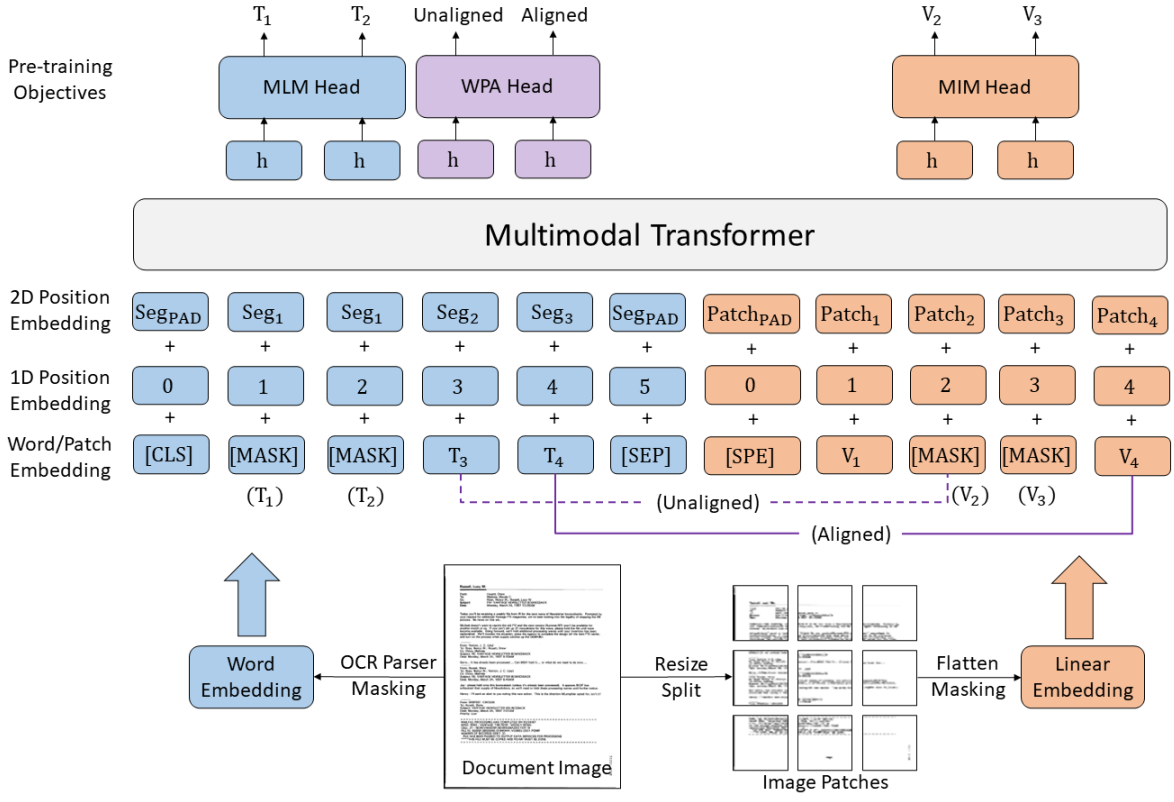

FIGURE 8 – Architecture LayoutLMv3

3.6 Large Languages Model (LLM) : Serveur Ollama et modèle Gemini

Avant de revenir sur ce qu'est un LLM, nous allons revenir sur ce qu'est un Natural Language Processing (NLP). Un NLP a pour but de traiter du langage pour comprendre chacun des mots et de comprendre le contexte associé à l'utilisation de ces mots. Un NLP peut ainsi classifier des phrases, des mots dans une phrase, générer du texte, extraire une réponse à partir d'un texte et générer de nouvelles phrases à partir d'un texte. Un NLP peut être spécialisé à des tâches spécifiques, possède une approche simple, utilise des données en petites quantités.
Un LLM est un type spécifique de modèle NLP qui utilise du Deep Learning avancé pour générer du langage naturel. Un LLM a pour but de générer du texte et possède des modèles complexes avec plusieurs millions de paramètres. Ils sont plus généraliste dans leur capacité à comprendre et générer du langage mais dépendent fortement des données d'entraînement qui doivent être massives et diverses pour obtenir des résultats qualitatifs. [13]

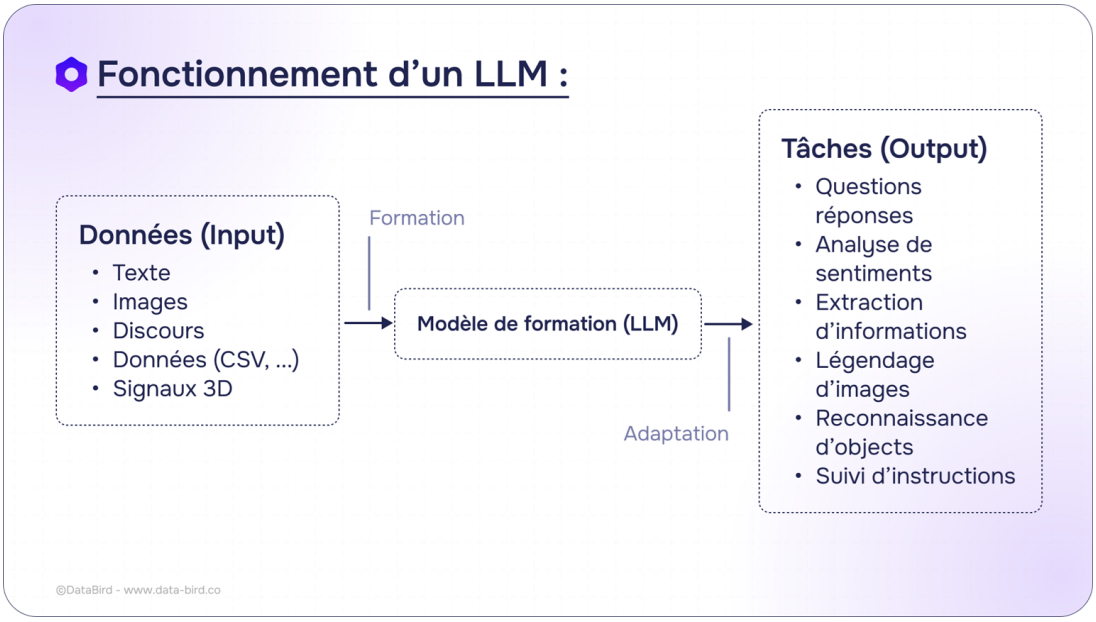

FIGURE 9 – Schéma de fonctionnement d'un LLM

Dans notre cas, nous allons utiliser le serveur Ollama en local et le LLM gemma3 :4b provenant de Gemini. Gemini est une famille de modèles d'IA développée par Google et est multimodale, c'est-à-dire, qu'elle est capable de traiter de l'audio, du code, des vidéos, et surtout dans notre cas des images [14]. Le modèle choisi possède ainsi environ 4 milliard de paramètres, 128 000 tokens et demande seulement 3,2 Go de mémoire [15]. Nous allons réaliser une requête via ce modèle pour qu'il nous ressorte le prix de notre facture.

FIGURE 10 – Logo de Gemini 3

FIGURE 11 – Logo de Ollama

3.7 LayoutLLM : DocLLM

Ce que nous faisons finalement avec l'utilisation du modèle Gemini sur le serveur Ollama se rapproche de ce que fait un LayoutLLM. En effet, un LayoutLLM utilise la puissance d'un LLM pour en extraire du texte, des formes, du textes,...

Le principe est d'abord d'encoder un visuel/layout pour lire l'image et extraire des tokens visuels ou des blocs, puis de fusionner les tokens avec du texte extrait pour finir par le lancement d'un LLM prédéfini. DocLLM en est un exemple qui est fondé sur l'auto-régression du transformer langage modèle suivi de d'une structure décodeur causal, c'est-à-dire, que chaque token (mot) ne peut voir que les tokens précédents. Il est composé de plusieurs blocs de transformers entassés et chacun de ces blocs contient une tête multi-couche et complètement connecté au partage internet. [16]

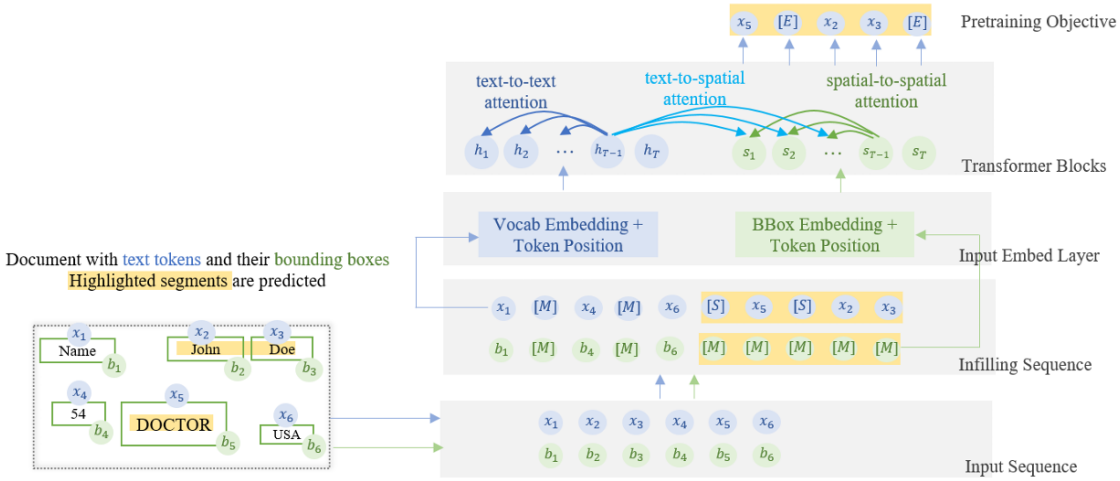

FIGURE 12 – DocLLM modèle architecture

À gauche, le document en entrée avec les tokens de texte xi et les encadrés. À droite, la séquence créée remplaçant les segments avec [M] et les préfixant avec [S].

4 Roadmap et outils collaboratifs

4.1 Roadmap

Nous avons réalisé notre projet sur une période d'environ trois mois à partir de février. Voici la Roadmap du projet.

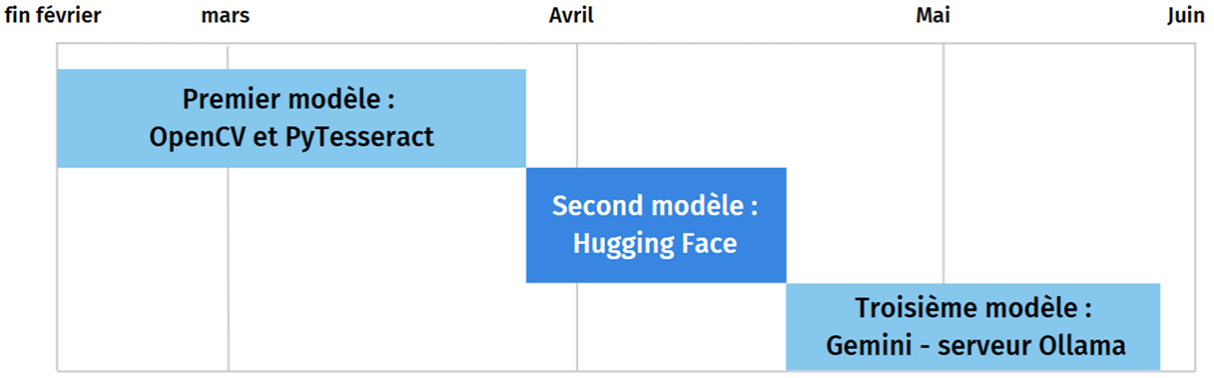

FIGURE 13 – Roadmap

4.2 Outils collaboratifs

Nous avons d'abord échangé nos codes en utilisant un Jupyter Notebook. Par la suite, nous nous sommes rendus compte qu'il était plus judicieux de mettre notre projet sur Github [2].

FIGURE 14 – Logo Jupyter Notebook

FIGURE 15 – Logo Github

5 Premier Modèle : Bibliothèque OpenCv et Pytesseract

5.1 Pré-traitement de l'image

Dans un premier temps, nous procédons à l'amélioration de la qualité des images en vue d'extraire précisément l'information cible, à savoir **le montant total** inscrit sur la facture. Ce processus débute par une phase de réduction du bruit, indispensable pour supprimer les artefacts susceptibles de perturber les étapes suivantes de traitement. Étant donné que les factures présentent généralement une palette de couleurs limitée, une opération de binarisation globale ou adaptative est réalisée. Cette conversion en noir et blanc a pour objectif de maximiser le contraste entre le texte et l'arrière-plan, tout en normalisant les variations locales de luminosité.

FIGURE 16 – Image d'origine

FIGURE 17 – Image noir et blanc

Après avoir transformé l'image en binaire, la détection des contours est effectuée à l'aide d'un détecteur de bords basé sur le calcul de gradients d'intensité, tel que l'algorithme de Canny. Ce détecteur permet d'extraire efficacement les structures fines présentes dans l'image, en particulier les zones de texte et les montants numériques, en minimisant les détections erronées dues au bruit résiduel.

FIGURE 18 – Image floutée

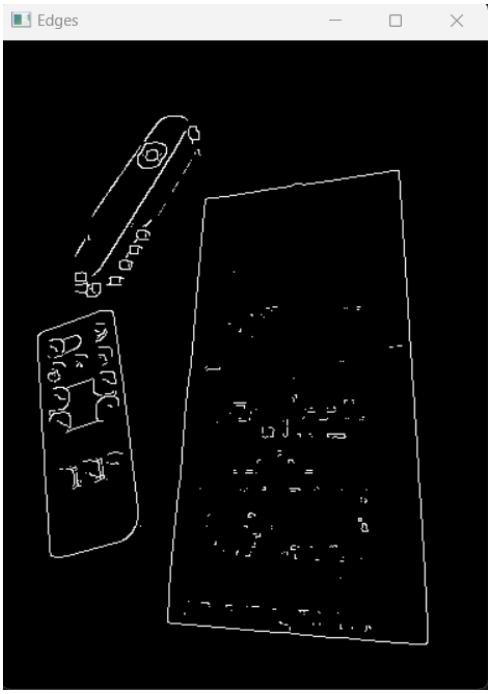

FIGURE 19 – Image avec filtre Canny

Afin de renforcer la continuité des contours et d'agréger les fragments textuels proches, une opération morphologique de dilatation est ensuite appliquée. Cette opération augmente l'épaisseur des structures détectées, facilitant ainsi la fusion des composantes liées au montant total.

Enfin, une phase d'analyse des contours est engagée. Les contours extraits sont triés en fonction de critères géométriques tels que la surface, l'aspect ratio et la position relative dans l'image. L'objectif est d'isoler les régions de plus grande taille correspondant typiquement aux zones où est inscrit le montant total. Cette stratégie permet de limiter les erreurs d'extraction et de garantir une détection robuste dans des conditions de qualité variable.

FIGURE 20 – Image floue

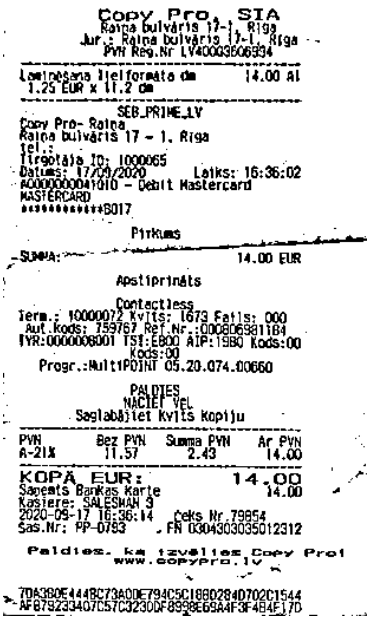

FIGURE 21 – Image recentrée

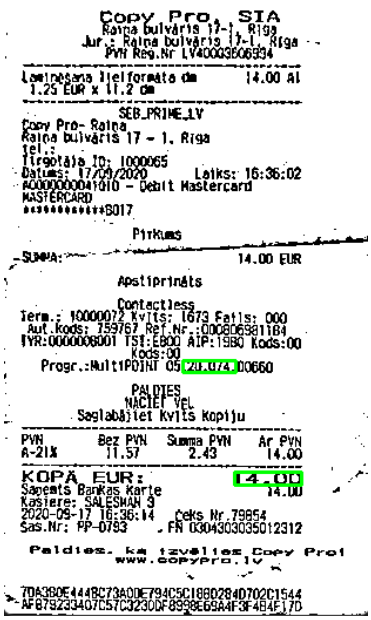

FIGURE 22 – Image contour montant

Nous avons donc ainsi le montant total qui semble bel et bien entouré comme nous le voulions. Ici, nous utilisons un **filtre Regex**

```
pattern_regex = re.compile(r'\d{1,3}(?:[.,]\d{2})')
```

L'expression régulière utilisée dans ce filtre a pour but de détecter des valeurs numériques au format décimal, où la partie entière est constituée de 1 à 3 chiffres, suivie obligatoirement d'un séparateur décimal (point ou virgule), et de deux chiffres après la décimale.

5.2 Application au dataset

Nous appliquons notre programme à un dataset composé de 53 images. Nous faisons deux tests de comparaison : le premier sans filtre (on utilise directement PyTesseract), le second en utilisant notre filtre.

FIGURE 23 – Détection sans filtre

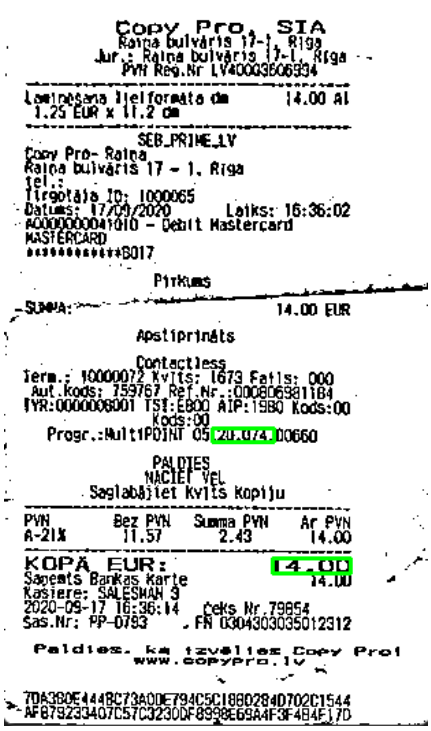

FIGURE 24 – Détection avec filtre

On enregistre le montant des deux tests et on le compare à une liste des valeurs réelles (valeurs mises à la main). De là, on peut comparer le taux de succès de notre filtre par rapport à une valeur de référence.

Métrique	**Avec filtre**	**Sans filtre**
Pourcentage de réussite (%)	40.385	0.000
Erreur absolue moyenne : MAE (€)	72.10	53.84

TABLE 1 – Comparaison des performances avec et sans filtre

Précision : avec notre filtre, 27 images sur les 53 n'ont pas leur contour qui est détecté.

5.3 Résultat et analyse des résultats

Nous obtenons alors le résultat suivant :

— Pour les images sans filtre, nous obtenons un pourcentage de succès à environ 40%, ce qui est un résultat correct. D'après les métriques obtenues, les erreurs obtenues sont cependant très grandes.

— Pour les images où nous utilisons notre filtre, nous avons un pourcentage nul. C'est-à-dire qu'aucun résultat obtenu avec notre filtre correspond à la valeur réelle.

— Sur les 53 images, notre filtre n'a pas réussi à dessiner correctement le contour de la facture correctement que pour 27 d'entre elles.

Ce résultat est certes décevant, mais nous pouvons l'analyser pour mieux le comprendre. Tout d'abord, un peu près la moitié de nos images ne sont pas reconnus par notre filtre à cause d'erreurs de contours. Voici une image type où notre code n'arrive pas à placer les quatre points nécessaire pour tracer un quadrilatère.

FIGURE 25 – Image où notre programme ne détecte pas le contour

Pour d'autres images, le problème vient de notre filtre Regex. Certaines factures sont différentes sur ce point et le nombre le plus grand ne correspond pas au montant total. Dans l'image suivante, le montant le plus grand serait ce que le client a rendu.

FIGURE 26 – Image montant total ($6.59) n'est pas le plus grand ($10).

6 Second Modèle : Hugging Face

6.1 Extraction du montant

Nous ferons un double test sur le modèle de Hugging Face. Le premier utilise seulement la détection du prix avec un pré-traitement sur l'image (agrandissement de l'image, conversion en niveau de gris, accentuation du contraste et de la netteté). Le second utilise en plus un "fallback" (solution de repli) utilisant PyTesseract si aucun montant n'est détecté.

FIGURE 27 – Résultat type

Nous obtenons ainsi les résultats suivants :

Métrique	**Avec fallback**	**Sans fallback**
Pourcentage de réussite (accuracy) (%)	59.62	62.00
MAE (€)	55.97	58.21
Nombre d'erreurs de traitement sur 53 images	1	3

TABLE 2 – Comparaison des performances avec et sans fallback

6.2 Résultat et analyse des résultats Hugging Face

Nous obtenons alors le résultat suivant :

— Le pourcentage de réussite avec et sans fallback reste significativement la même, même si avec le fallback, on a quelques images qui n'arrivent pas à être traité parmi celle où le résultat n'était pas correct. (Les images dont on ne trouvait pas une valeur correcte sans le fallback sont restés avec le fallback, qui cette fois-ci renvoie une erreur de traitement s'il y a une erreur dans le résultats.)

— Nos deux modèles ont des erreurs très fortes erreurs. Lorsqu'on regarde plus attentivement la répartition des erreurs, on s'aperçoit que des grosses erreurs apparaissent mais sont minoritaires, ce qui fait gonfler les erreurs :

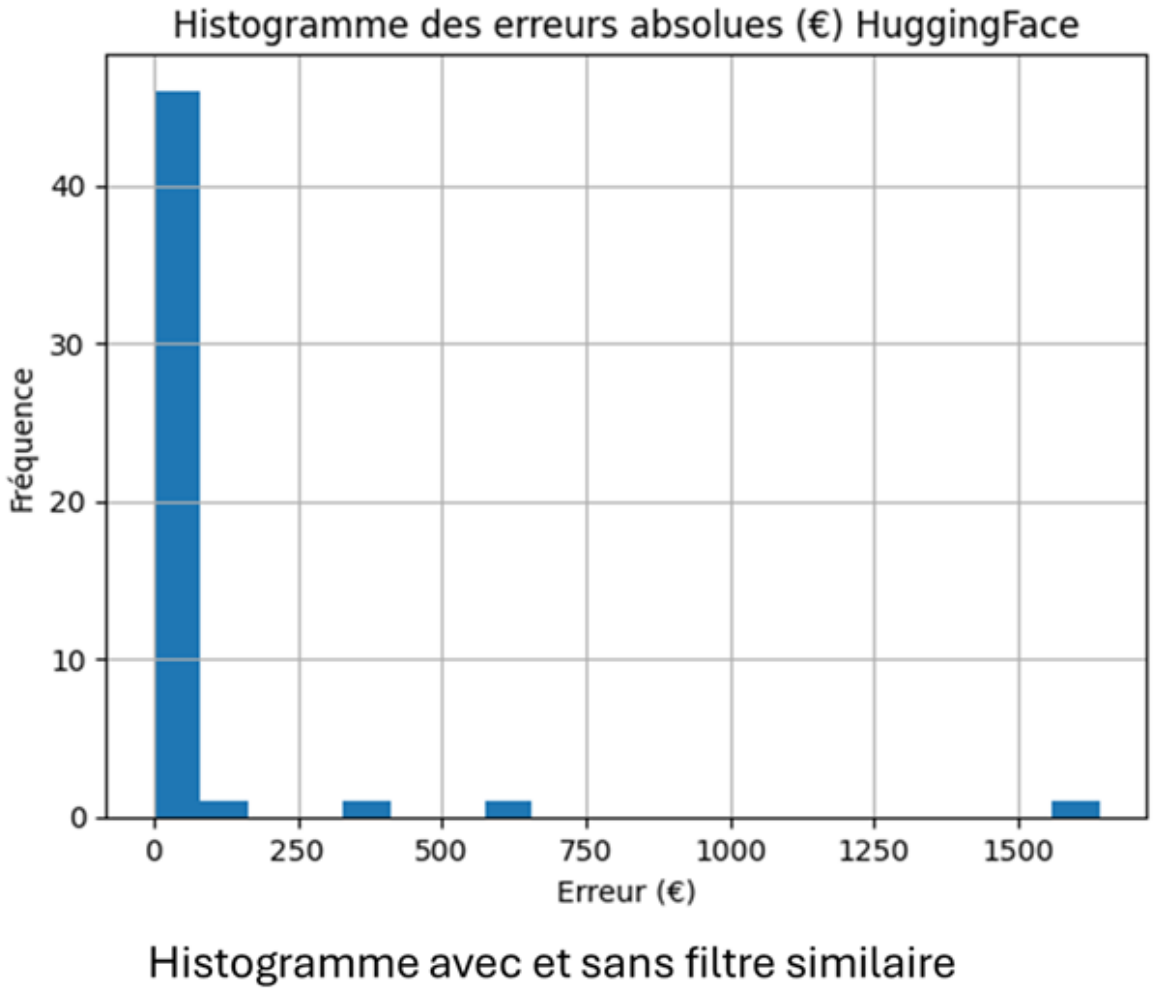

FIGURE 28 – Répartition des erreurs

7 Troisième Modèle : LLM et serveur Ollama

7.1 Utilisation du GPU et résultats

Lancer un LLM pour faire des requêtes demande énormément de puissance de calcul. En effet, un LLM contient plusieurs millions, voire de milliard de neurones et le CPU d'un PC n'est pas conçu pour faire tourner un LLM. Nous utiliserons donc un GPU (déjà présent sur notre PC) via une WSL pour pouvoir faire tourner notre modèle gemma3 :4b sur le serveur Ollama. Pour donner une comparaison, il a fallu environ 9 min à notre CPU pour traiter une demande, alors qu'avec notre GPU il n'a fallu que de 5 secondes pour la même demande.

Nous envoyons alors un message à notre lui donnant le rôle d'analyser une facture et de renvoyer le montant total avec 2 décimales.

Métrique	**Ollama**
Pourcentage de réussite %	90.57
Erreur absolue moyenne : MAE (€)	0.24
Nombre d'erreurs de traitement sur 53 images :	0

Table 3 – Résultats avec Ollama

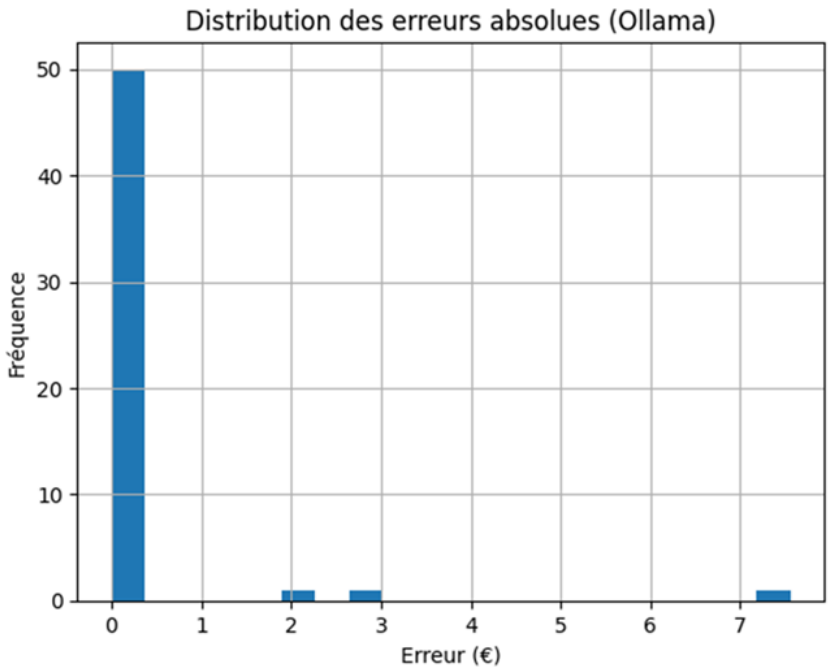

Figure 29 – Répartition des erreurs

Cependant, nous allons nuancer ces résultats. En effet, un LLM n'est pas un modèle de Machine Learning qui se base sur une "seed", un chiffre duquel on construit des nombres aléatoires. La source d'aléa est plus complexe et il est possible d'avoir des résultats différents avec les caractéristiques de départ (température, OS, librairies, données, ...). En relançant plusieurs fois notre code, nous pouvons nous rendre compte que le pourcentage de réussite varie légèrement, pouvant monter jusqu'à 96%.

7.2 Analyse des résultats

Le modèle avec Ollama est clairement le plus performant. Nous arrivons au résultat supérieur à 90% que nous souhaitions. Nous avons des erreurs faible (MAE=0.24). Néanmoins, cette méthode est coûteuse, que cela en énergie, en mémoire (3 GB pour installer le modèle) ou en puissant de calcul. De plus, le fait qu'un LLM ne soit pas déterministe pose un problème de taille.

8 Conclusion & Perspective

8.1 Conclusion

Au cours de notre projet, nous avons finalement fait un retour chronologique des différentes technologies utilisées pour détecter des éléments sur une image.

- Au départ, l'utilisation de bibliothèques Python OpenCV et PyTesseract permettant de filtrer et détecter du texte d'une facture.
- Puis, l'utilisation d'un modèle pré-entrainé provenant de la bibliothèque Hugging Face qui prédit les éléments d'une facture, dont le montant.
- Enfin, l'utilisation du serveur Ollama pour utiliser un LLM multimodal beaucoup plus poussé.

Nous avons pu voir que nos méthodes de détection du montant de notre facture deviennent plus performantes au fur et à mesure que nos modèles deviennent complexe. Cette complexité peut cependant avoir un coût au niveau de la mémoire, notamment pour le LLM Ollama.

8.2 Perspective

Il existe d'autres modèles qui pourraient répondre à notre projet. Un LayoutLLM tel que DocLLM pourrait être une solution envisageable même s'il consommerait des ressources comme notre LLM.

Une autre modèle serait de créer notre propre modèle avec un NPL qu'on entraînerait avec plusieurs autres datasets. L'avantage serait que le NPL serait moins coûteux.

Nous devons aussi rappeler que nous avons fait nos tests sur un petit dataset de donnée qui est certes représentatif des possibles images de factures que nous aurions, mais nous pourrions utiliser un dataset plus large pour avoir une deuxième intéprétation.

Références

[1] IONOS. *Qu'est-ce qu'un wrapper ? 14-Sept.-2020.*
https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-quun-wrapper/

[2] Notre projet sur Github. *Benjamin333Sz, "Lecture automatique de Facture", Github. 11-Apr.-2025.*
https://github.com/benjamin333sz/Lecture_automatique_facture/

[3] Explication de Tesseract. *UB-Mannheim, "Tesseract at UB Mannheim", Github. 8-Déc.-2015*
https://github.com/UB-Mannheim/tesseract/wiki

[4] GitHub encadrant. *Wajd Meskini, "receipt-ocr-project", Github*
https://github.com/atracordis/receipt-ocr-project/

[5] OpenCV *Documentation sur OpenCV 8-July-2016.*
https://opencv24-python-tutorials.readthedocs.io/en/latest/

[6] Hugging Face *Présentation Hugging Face, GeeKanJi, 27-Jan.-2025.*
https://cosmo-games.com/hugging-face/

[7] Hugging Face Transformers *Bibliothèque Transformers de HuggingFace, O. R, 13-Apr.-2025.*
https://www.lebigdata.fr/hugging-face-transformers

[8] Framework Datascientest *Présentation de ce qu'est un Framework, 17-June-2021.*
https://datascientest.com/quest-ce-quun-framework

[9] Détection de facture *Projet Hugging Face détection facture*
https://huggingface.co/Theivaprakasham/layoutlmv3-finetuned-invoice

[10] LayoutLM *Première version LayoutLM conception*
https://huggingface.co/docs/transformers/main/en/model_doc/layoutlm

[11] LayoutLM2 *Seconde version LayoutLM conception*
https://huggingface.co/docs/transformers/main/en/model_doc/layoutlmv2

[12] LayoutLM3 *Troisième version LayoutLM conception*
https://huggingface.co/docs/transformers/main/en/model_doc/layoutlmv3

[13] NLP et LLM *difference entre NLP et LLM*
https://huggingface.co/learn/llm-course/en/chapter1/2?fw=pt

[14] Gemini *Origine de Gemini et autres informations, A. Leeno, 30-Apr.-2024.*
https://www.lesnumeriques.com/societe-numerique/tout-savoir-sur-google-gemini-a221039.html

[15] Informations relatives au modèle gemma3 :4b de ollama *gemma3 :4b*
https://ollama.com/library/gemma3:4b

[16] DocLLM *Document officiel explicatif du LLM DocLLM avec son préentraînement*
https://arxiv.org/pdf/2401.00908