UPPSALA
UNIVERSITET

Evaluating Visual-Language Models for Handwritten Text Recognition in Historical Swedish Manuscripts

Hoang Ha Pham

Uppsala University
Department of Linguistics and Philology
Master Programme in Language Technology
Master's Thesis in Language Technology, 30 ECTS credits

May 22, 2025

Supervisors:
Fredrik Wahlberg, Uppsala University
Erik Lenas, The Swedish National Archives

Abstract

Optical Character Recognition (OCR)—and by extension, Handwritten Text Recognition (HTR)—has a long history of research and development. While OCR for printed text has become a mature and widely adopted technology, HTR remains a challenging task due to the high variability in human handwriting. Over time, the field has shifted from rule-based systems to more adaptable neural network-based approaches. Recently, the emergence of large language models and, subsequently, visual-language models (VLMs) has introduced new possibilities for tackling the HTR problem.

This thesis investigates the viability of VLMs for HTR, using Microsoft's Florence-2 as a case study. The model is evaluated against traditional computer vision models across individual tasks, such as text region detection and segmentation (using YOLO) and text recognition (using TrOCR). Additionally, end-to-end HTR pipelines composed of these models are compared on text recognition accuracy.

The findings show that Florence-2 performs comparably to, and in some cases surpasses, task-specific models in isolated tasks. Furthermore, a Florence-based two-step pipeline (line detection followed by text recognition) outperforms the traditional three-step pipeline (region detection, line segmentation, and text recognition) in standard text recognition metrics.

However, the study also reveals that Florence-2 demands significantly greater computational resources than conventional models. As a result, while VLMs such as Florence-2 demonstrate strong performance, at the current stage, they are not well-suited for large-scale text annotation projects.

Contents

1. Introduction

At present, the problem of Optical Character Recognition (OCR) on printed text has matured and is generally considered a solved problem. In contrast, Handwritten Text Recognition (HTR)—a closely related task—continues to pose significant challenges. Unlike OCR, the target of HTR is text produced naturally by humans. The combination of writing habits and linguistic differences gives rise to near-infinite variations, making the task inherently challenging. Despite these difficulties, handwritten documents contain a wealth of valuable information that warrants attention. This is particularly true for historical manuscripts which hold important cultural and historical insights relevant to both researchers and the general public.

The recent emergence of large language models (LLMs) capable of handling various language tasks has prompted a shift in how the HTR problem is approached. Drawing on techniques from multi-task learning in LLMs, researchers have developed visual-language models that combine computer vision and natural language processing components into a unified architecture. This integration also enables novel forms of interaction between the system and end-users that are unseen before.

In this context, the present thesis investigates how well a VLM performs the HTR task on historical Swedish manuscripts. The project is conducted in collaboration with the Swedish National Archives (Riksarkivet). The complete code base for this project is publicly accessible via my GitHub page.[1]

1.1. Research questions

This work aims to address the following research questions:

1. Can large VLMs perform Handwritten Text Recognition (HTR) on Swedish historical manuscripts?
2. How does the HTR performance of VLMs compare to specialized HTR pipelines?
3. How does the computational efficiency of VLMs compare to specialized HTR pipelines?
4. How does fine-tuning on multiple tasks affect the performance of VLMs?

1.2. Outline

Following this introductory chapter, Chapter 2 provides an overview of the OCR and HTR problems. Here, I formally define the HTR problem, review prior approaches, and discuss its application in the context of historical manuscript annotation. Chapter 3 outlines the experimental design, including the evaluated pipelines and evaluation metrics. Chapter 4 presents the quantitative results along with a qualitative analysis to offer an insight into Florence-2's successful and failure cases. Chapter 5 discusses the limitations of the current work and highlights areas that remain unexplored. Chapter 6 summarizes the findings and provides answers to the research questions. Finally, Chapter 7 proposes potential directions for future research.

[1] https://github.com/hoanghapham/visual-language-models-htr

2. Background

2.1. Overview

This background chapter provides a theoretical foundation for the project. For clarity, in this Background chapter, the term Optical Character Recognition (OCR) will specifically refer to the recognition of printed text, while Handwritten Text Recognition (HTR) will denote recognition applied to handwritten documents. It is worth noting that in other literature, the term OCR is used as an umbrella term that includes HTR. In other parts of the thesis, the terms HTR and OCR may be used interchangeably depending on the context.

2.1.1. Optical Character Recognition (OCR) - A Brief History

Optical Character Recognition (OCR) refers to the task of using electronic or mechanical systems to convert images of typed or printed text into machine-encoded text, so that they can be easily processed, analyzed, and searched. The source material can be physical (such as books, documents, or billboards) or digital (such as scanned documents or photographs). OCR has been an active research field for more than a century, and today it underpins a wide range of applications that, not long ago, would have been considered science fiction.

The origins of OCR can be traced back to early efforts to assist visually impaired individuals with reading (Schantz, 1982). Between 1912 and 1914, the first functional OCR products were independently developed by Emmanuel Goldberg and Edmund Fournier d'Albe. Goldberg's device was capable of converting printed text into standard telegraph code, while d'Albe's invention—the Optophone—was designed specifically to aid the blind by generating tones corresponding to letters as the device was moved across a printed page (d'Albe, 1914).

Between the 1930s and 1950s, OCR attracted significant interest from the business sector as a solution for electronic data processing, yet much of the research was not made public. For example, IBM had been filing patents since the 1940s, but did not release a commercial product until the 1960s. The situation changed in the 1950s, when David Hammond Shepard developed the Gismo system capable of reading Morse code, musical notation, and printed texts (Norman, 2025). Gismo was the first OCR system produced outside corporate or government research facilities, thereby bringing substantial public attention to the technology. In the 1970s and 1990s, OCR continued to mature as a practical data entry tool and an assistive technology for the visually impaired (Hauger, 1995). During this period, Raymond Kurzweil pioneered the development of reading machines that synthesized speech from printed text, and also introduced an OCR system capable of recognizing text in virtually any standard font.

Finally, the release of Tesseract in 2005 was a breakthrough that propelled the public adoption of OCR technology. Originally developed by Hewlett-Packard (HP) in the 1980s as proprietary software, Tesseract was later open-sourced by HP and the University of Nevada, Las Vegas in 2005. In 2006, its development received further support from Google (Vincent, 2006). At the time, it was regarded as the most accurate open-source OCR engine available (Willis, 2006). Since then, OCR technology has

advanced rapidly, becoming accessible to the general public through online services, mobile applications, and even native integrations in smartphone camera software.

2.1.2. Handwritten Text Recognition (HTR) - Introduction

Handwritten Text Recognition (HTR) refers to the process of converting handwritten text into machine-readable characters. This technology has a wide array of applications, including historical document preservation, automated data entry, digital note-taking, and assisting the visually impaired.

As of today, the problem of OCR for printed text is largely solved. However, HTR continues to pose significant challenges due to the non-standard nature of handwriting. Unlike standardized fonts (or even fonts developed specifically for OCR use-cases like OCR-A and OCR-B in financial and banking institutions (*Character set for optical character recognition (OCR-A)* 1981; *Character set for optical character recognition (OCR-A)* 1982)), handwritten text is influenced by the unique characteristics of the author, as well as the context in which it was produced. Factors such as personal writing style, educational background, cultural influences, the type of pen or ink used, as well as the writer's health and emotional state, all contribute to this variability. As a result, techniques effective in OCR do not easily translate to HTR.

In the following sections, I will formally define the HTR problem and explore the unique solutions that have been developed to address it.

2.2. HTR Problem Formulation

2.2.1. Mathematical Model

The goal of HTR is to find a sequence of characters y that most likely represents the text in an image x. Garrido-Munoz et al. (2025) expressed this as an equation:

$$y^* = \underset{y \in \Sigma^*}{\operatorname{argmax}} P(y|x) \tag{2.1}$$

where Σ is a predefined alphabet, Σ^* represents a set of all possible sequences composed of characters in that alphabet, and y is a sequence of characters $(y_1, y_2, ..., y_N)$ with $y_i \in \Sigma$.

This problem of identifying the most probable character sequence is known to be NP-hard (De la Higuera and Oncina, 2014). In other words, there is no known polynomial-time deterministic algorithm that can solve it efficiently. Therefore, current HTR methods rely on approximations to find solutions.

We can use Bayes' theorem to reformulate Equation 2.1 so that the answer can be approximated using data:

$$\hat{y} = \underset{y \in \Sigma^*}{\operatorname{argmax}} \frac{P(x|y)P(y)}{P(x)} \tag{2.2}$$

where $P(x)$ is the probability of image x, $P(y)$ is the probability of the transcription y, and $P(x|y)$ is the probability of observing image x given an observed transcription y. In practice, y is often searched within a lexicon $L \subset \Sigma^*$ which drastically reduces the complexity of the problem. Finally, because the images are independent from the transcriptions, $P(x)$ can be eliminated from Equation 2.2. The final equation is:

$$\hat{y} = \underset{y \in L}{\operatorname{argmax}} P(x|y)P(y) \tag{2.3}$$

In summary, this formula illustrates that the solution to the HTR problem can be approximated using a statistical model to estimate the likelihood of the image, $P(x|y)$, and a language model to estimate $P(y)$.

2.2.2. The Complexity of HTR Problem

The complexity of the HTR depends on the type of input images that need to be transcribed. Garrido-Munoz et al. divide HTR problem into two large categories: **up-to-line level** (which deals with transcribing characters, words, lines), and **beyond-line level** (which deals with transcribing paragraphs and full documents).

At the lowest tier of the up-to-line level is the character recognition problem, which can be framed as a basic image-character classification task. As we move beyond to words and lines levels, the task not only requires classification, but also the modeling of reading order (RO), which refers to the sequence in which characters, words, lines, paragraphs and figures should be read to preserve meaning and logical flow.

In general, words and lines follow a straightforward RO, albeit with line transcriptions, we need to deal with more text and word delimiters (such as spaces, commas, and periods). On the other hand, at the beyond-line level, paragraph text recognition involves two ROs: one to read text within lines and one to traverse between lines. Finally, page level has the highest complexity, where a page may contain multiple paragraphs and other types of contents (images, tables, charts, etc.) arranged in arbitrary layouts. Examples of these cases can be found in Appendix D.

In short, each level of complexity requires a different approach. Solutions designed for up-to-line level transcription differ significantly from those tackling beyond-line tasks, such as transcribing entire paragraphs or full pages in a single pass. In this work, I will primarily focus on solutions that address line-level transcription.

2.2.3. High-level HTR Pipeline

At a high level, an HTR pipeline consists of two main steps: **feature extraction** step in which images are transformed into numerical representations, and **alignment** step in which extracted features are mapped to a corresponding text sequence.

Image processing and feature extraction belong to the broader field of computer vision, with a long history of development. In earlier approaches, this stage involved extensive manual work and hand-crafted features. However, in the past decade, the rise of neural networks—particularly Convolutional Neural Networks—has automated much of this process, allowing the model to learn features directly from the data. Alignment techniques have also evolved significantly and currently receive the most attention of researchers. While traditional approaches such as Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) required substantial design effort and domain-specific knowledge, recent neural network-based solutions incorporate less inductive bias and have higher performance.

In this work, I concentrate specifically on the alignment process, with an emphasis on neural network-based methods.

2.3. Traditional HTR

In the early days, HTR solutions focused on word recognition and used hand-crafted pipelines. Vinciarelli (2002) outlined a pipeline for cursive word recognition, but this should apply to HTR in general.

1. **Preprocessing**: Remove background or other irrelevant elements in the image, and convert the image to grayscale. Binarization may also be performed to remove the text (foreground) from the background.
2. **Normalization**: Reduce individual writer variation with skew correction, slant correction, denoising, image scaling, etc.
3. **Segmentation**: Cut the image into smaller units which can be characters (explicit segmentation) or arbitrary regions (implicit segmentation). The type of segmentation will affect how alignment is done.
4. **Feature extraction**: Calculate feature vectors from the segments. Typically, features are selected manually, and they are categorized into low-level (letter fragments), mid-level (letters), and high-level (whole words).
5. **Recognition / Alignment**: Use the extracted features to find the best match between image segments and transcriptions. Early approaches include dynamic programming and the shortest-path-in-graph method (Garrido-Munoz et al., 2025). Later, Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs) became prominent as they can inherently model sequential data (Feng et al., 2006; Kundu et al., 1989).
6. **Lexicon reduction**: Reduce the number of possible interpretations with a predefined lexicon to improve system accuracy.

An example of this traditional pipeline can be found in Appendx B. It is easy to see that this is a laborious process with a high possibility of cascading failures. For example, image segmentation errors can drastically affect the final text alignment step, and it is not always easy to segment handwritten text into individual characters. Moreover, even though HMMs and CRFs represent a significant advancement, according to Graves et al. (2006), they are limited by the need for hand-crafted design choices. For instance, HMMs require the user to design state models, and they rely on explicit assumptions to make inference tractable, such as the assumption that observations are independent.

2.4. Neural Networks-based HTR

2.4.1. High-level NN-based Pipeline

Since the early 2010s, the popularization of Neural Networks (NNs) has significantly advanced OCR and HTR technologies. NN-based approaches eliminate the need for elaborate hand-crafted feature engineering pipelines, learn features automatically, and offer a computationally efficient framework that can leverage larger training datasets.

Figure 2.1 illustrates a typical modern encoder-decoder HTR solution. Each element of this architecture can be modified, or even eliminated (for example, DTrOCR (Fujitake, 2024) is a decoder-only model). There are also different ways to perform alignment, for example, Connectionist Temporal Classification, or sequence-to-sequence.

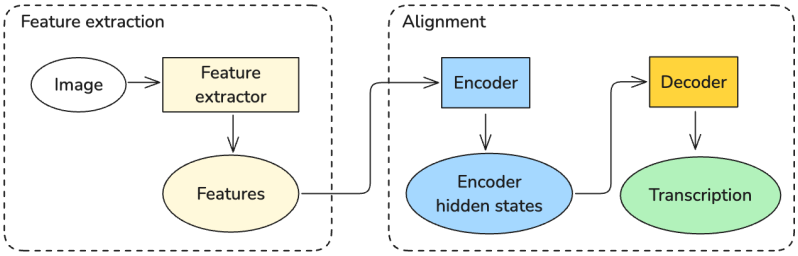


Figure 2.1.: High-level architecture of encoder-decoder HTR solution.

2.4.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of neural network commonly used to extract features from images, replacing the use of raw RGB pixel values. A CNN processes an image by sliding filters (or kernels) of a certain size across it and computing the dot product between the filter and the pixel values at each position. This process is illustrated in Figure 2.2.

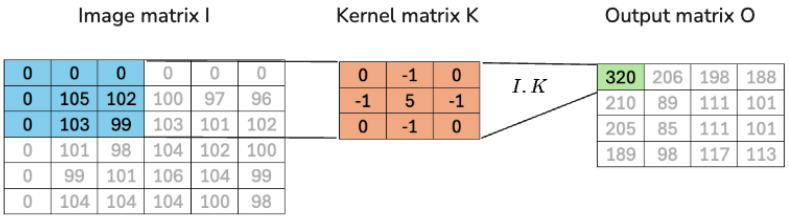


Figure 2.2.: An example of how CNN extract features from an image. The image is first converted into a matrix. If the image uses RGB color model, the dimension of the matrix I is $3 \times W \times H$, and each cell represents the intensity value of each color channel (red, green, blue). A kernel matrix K is used as a filter sliding across the images. At each position, we take dot product $I.K$ and save the result to an output matrix. In this example, only the operation on one channel is shown.

Compared to raw pixel values, CNN features offer several advantages: they preserve the two-dimensional structure of images, encode spatial hierarchies, and learn relevant features automatically. This eliminates the need for manually crafted and hand-selected features, making CNNs highly effective for image-based tasks (LeCun et al., 1989).

2.4.3. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are designed to process sequential data by feeding inputs into the network iteratively, and the output from one time step is used as the input for the next. This architecture is well-suited for modeling sequential data, where the probability of each symbol depends on the previous ones. RNNs were popularized by Mikolov et al. (2010), and have since become fundamental in many sequence-to-sequence modeling tasks. RNNs are naturally suited for transcribing text from images, where the spatial layout of the characters follows a consistent reading order. Figure 2.3 illustrates how RNN is used in an encoder-decoder solution for such a task.

One major challenge in training RNNs is the vanishing gradient problem, where gradients diminish as they are backpropagated through many time steps, making it difficult to learn long-range dependencies Pascanu et al. (2013). To address this issue, variants such as Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) networks were developed. These introduce gating mechanisms that help preserve and regulate information across longer sequences.

Despite improvements like GRUs and LSTMs, RNNs still face limitations when handling long sequences. During the encoding process, the network accumulates input information into a fixed-size hidden state vector. In practice, this representation becomes a bottleneck, as it is unrealistic to expect one vector to fully capture all information in long sequences such as paragraphs or documents.

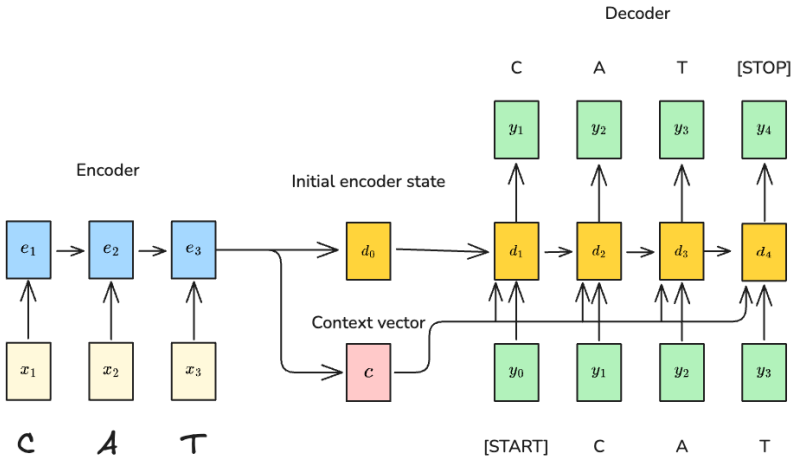


Figure 2.3.: Example of how RNN is used in an encoder-decoder setup. The encoder ingests visual features one by one to produce hidden states. The final encoder hidden state e_3 is used to predict the initial decoder state d_0, and is also used as the context vector c that is **shared for all decoder timesteps** ($c = e_3$). At decoder timestep t, context vector c and the prediction y_{t-1} of the previous step are used to produce new prediction y_t. y_0 and y_4 correspond to special tokens that mark the start and end of the decoded sequence.

2.4.4. Attention Mechanism and Transformer

To address the limitations of RNNs, the attention mechanism was introduced by Bahdanau et al. (2014). Rather than relying solely on a single fixed-size vector to encode the entire input sequence, attention allows the decoder to selectively focus on different parts of the input sequence (which are represented by the encoder's hidden state vectors). This makes it easier to capture long-range dependencies and improves performance on tasks involving long inputs.

There are three core concepts in the attention mechanism: Query, Value, and Attention. Firstly, the *Query* (Q) represents the current hidden state of the decoder at a given time step, while the *Values* (V) are the encoder's hidden states, which contain information about the input sequence. The *Attention* weights are calculated as the similarity between a query and various values (typically via a dot product) followed by a softmax function to produce a probability distribution. These weights determine which parts of the input the decoder should attend to at each decoding time step.

Initially, attention mechanisms were used in conjunction with RNN-based encoder-decoder architectures for sequence-to-sequence tasks such as machine translation (Bahdanau et al., 2014). Figure 2.4 illustrates such a setup.

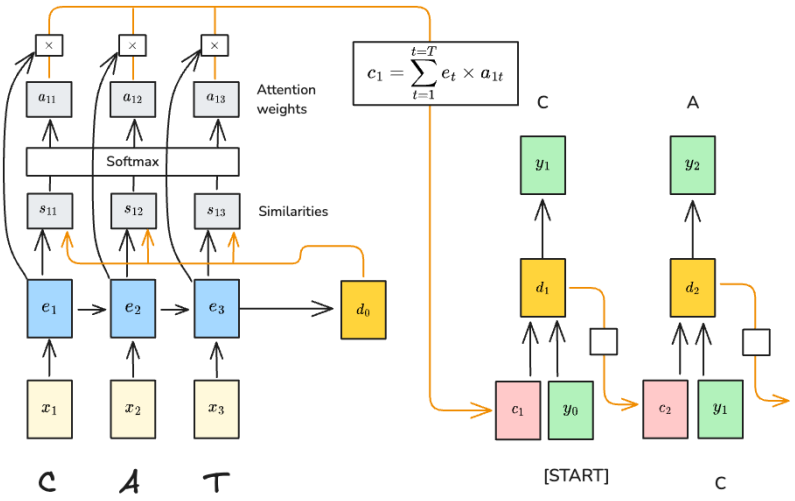


Figure 2.4.: An example of how Attention is used in an RNN encoder-decoder setup. The final encoder hidden state is used to produce initial decoder state d_0. At decoding time step 1, d_0 is compared with each encoder state e_t to produce similarity score s_{1t}, and then this is normalized with softmax to produce attention weights a_{1t} so that $\sum a_{1t} = 1$. Then the context vector c_1 is calculated as the weighted sum between the encoder hidden states e_1 and corresponding attention weights $a_1 1$. Then c_1 and y_0 are used to produce the prediction y_1. This process is repeated for all decoding time steps, **each time producing a different context vector** c_y.

Later, Vaswani et al. (2017) introduced the Transformer architecture, which discards recurrence entirely and relies solely on stacked attention layers and positional encodings to model sequential data. This architecture has since become foundational in modern language as well as multi-modal models.

There are different variations of attention mechanisms suited to different types of tasks. For example, in machine translation, the input and output belong to two distinct languages. In this case, we use **cross-attention**, where tokens in one sequence (e.g., the target language) attend to tokens in another sequence (e.g., the source language). In contrast, **self-attention** is used when the input and output belong to the same dataset. A common use case is next-word prediction, where each word in a sequence attends to some previous words within the same sequence (Vaswani et al., 2017).

2.4.5. Connectionist Temporal Classification (CTC) Alignment

CTC was introduced by Graves et al. (2006) as an RNN objective function to label sequence data with no clear segmentation, such as speech or handwriting. The unique feature of CTC is the ability to output a text sequence without requiring explicit segmentation of the input.

In audio or image transcription, the input sequence can be significantly longer than the output sequence which may consist of just a few words or characters. This length mismatch makes it impossible for typical RNN-based models to produce alignments, as RNNs operate with a fixed number of time steps.

CTC addresses this issue by allowing the model to predict repeated labels and a special "blank" symbol at each time step, resulting in an intermediate sequence of length T (the number of decoder time steps). These intermediate sequences are then collapsed by removing consecutive duplicate labels (that are not separated by blanks) and eliminating all blanks, yielding the final transcription. The probability of this transcription is the sum of the probabilities of all intermediate sequences that reduce to it. Figure 2.5 illustrates this process.

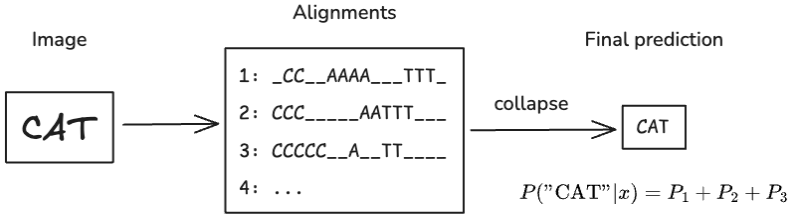


Figure 2.5.: An Example of how CTC gives a prediction. Several possible alignment sequences are produced, then collapse into a final output. The probability of the transcription "cat" is the sum of probabilities of the three corresponding sequences.

2.4.6. Classical Sequence-to-sequence (seq2seq) Alignment

Traditionally, seq2seq models have the encoder-decoder architecture (Figure 2.3): the *encoder* accepts input and produces a representation, and the *decoder* outputs the sequence from that representation (Sutskever et al., 2014). In the context of HTR, the encoder's inputs are text images, and the outputs are transcribed texts. Typically, RNN networks like LSTM are used for both the encoder and decoder.

Seq2seq models calculate the probability of a label sequence y given input x by taking the product of the conditional probabilities of tokens at time step t:

$$P(y|x) = \prod_{t=-1}^{T} P(y_t|y_1, y_2, ..., y_{t-1}, x) \tag{2.4}$$

Shkarupa et al. (2016) were among the first to apply seq2seq learning to the HTR problem. In their proposed approach, the encoder receives 1-pixel-wide vertical slices of the handwriting image as input, encodes the entire sequence of slices, and then passes the final hidden state of the encoder as the initial hidden state to the decoder LSTM. Sueiras et al. (2018) improves on this approach in two ways: firstly, they add a convolutional layer to extract visual features from handwriting images (instead of using pixel values directly), and secondly, they incorporate the attention mechanism into the architecture.

To improve the accuracy of the transcription, Kang et al. (2021) proposed a "candidate fusion" approach: a language model is first pre-trained on an external corpus and then integrated into the image-to-text recognizer to be jointly fine-tuned on the corpus of the handwritten dataset. During fine-tuning, at each time step, the character predicted by the recognizer is fed into the language model. In the decoding phase, at each time step, the recognizer generates an initial prediction y_t^o. This output is then refined by the language model to produce a corrected prediction y_t^{lm}. Finally, both predictions, along with the context vector c_t, are used as input for the next time step.

2.4.7. Transformer-based Alignment

Unlike classical seq2seq models that rely on recurrent decoding, the Transformer architecture eliminates recurrence and instead predicts all output tokens in parallel using the attention mechanism. To preserve the autoregressive nature of decoding, a masking strategy is applied in the self-attention layer. Specifically, when predicting the i-th character, all subsequent positions ($i + 1$ and onward) are masked out, ensuring that each prediction is conditioned only on the previous tokens.

Kang et al. (2022) were also among the first to apply the Transformer architecture to the HTR task. In their proposed system, a CNN backbone (specifically ResNet; He et al. (2016)) is used to extract visual features from the input images. Positional encoding is then added to preserve the order of characters seen on the image, and

then these features are passed through a self-attention module to generate the final visual representations with context.

More recently, Li et al. (2023) extended the idea further by proposing TrOCR, a model that leverages pre-trained vision and language models without relying on CNNs for visual feature extraction (Figure 2.6). The image encoder is initialized with weights from Vision Transformer models such as DeiT (Touvron et al., 2021) and BEiT (Bao et al., 2021), while the text decoder is initialized with pre-trained language models like RoBERTa (Liu et al., 2021) and MiniLM (Wang et al., 2020). The model uses a standard Transformer architecture for both encoding and decoding. At the time of writing, TrOCR is regarded as a state-of-the-art solution for both OCR and HTR tasks.

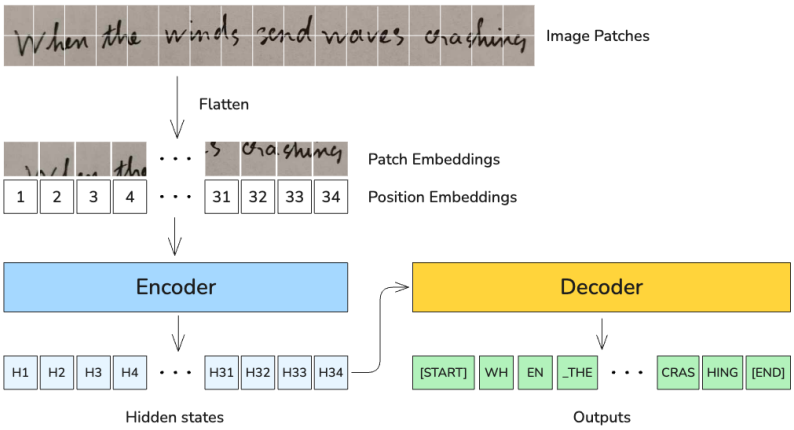


Figure 2.6.: Illustration of TrOCR's architecture, adapted from Li et al. (2023). The encoder is a pre-trained image Transformer, and the decoder is a pre-trained text Transformer. First, the image is divided into patches, then flattened and combined with positional embeddings before being fed to the encoder. The decoder uses the encoder's resulting hidden states to generate textual outputs.

2.5. Visual Language Models (VLMs)

In recent years, the release of large language models (LLMs) has revolutionized the field of AI and captured widespread public attention. Models such as LLaMA (Touvron et al., 2023) and, especially, GPT-3 (Brown et al., 2020) have impressed with their remarkable capabilities across a wide range of natural language processing tasks. Naturally, researchers have begun exploring how to apply these powerful tools to other domains, giving rise to Visual-Language Models (VLMs).

On the highest level, a VLM consists of a vision encoder and a text encoder. The vision encoder transforms visual input (such as images or video frames) into visual embeddings, while the text encoder converts textual input into text embeddings. These embeddings are then projected into a shared latent space and can be trained jointly.

Early VLMs use the *two-tower* architecture which has separate encoders for images and text. The vision encoder is typically a CNN such as ResNet (He et al., 2016) or a Vision Transformer (Dosovitskiy et al., 2020), while the text encoder is based on the Transformer architecture (Vaswani et al., 2017) and its variants, such as BERT (Devlin et al., 2019). The most well-known example of this architecture is CLIP (Radford et al., 2021). CLIP encodes images and texts into separate vectors, projects them into a shared latent space, and trains the model to maximize the similarity of correct image-text pairs while minimizing the similarity of mismatched pairs (Figure 2.7). This training strategy is known as *contrastive learning* (Oord et al., 2018).

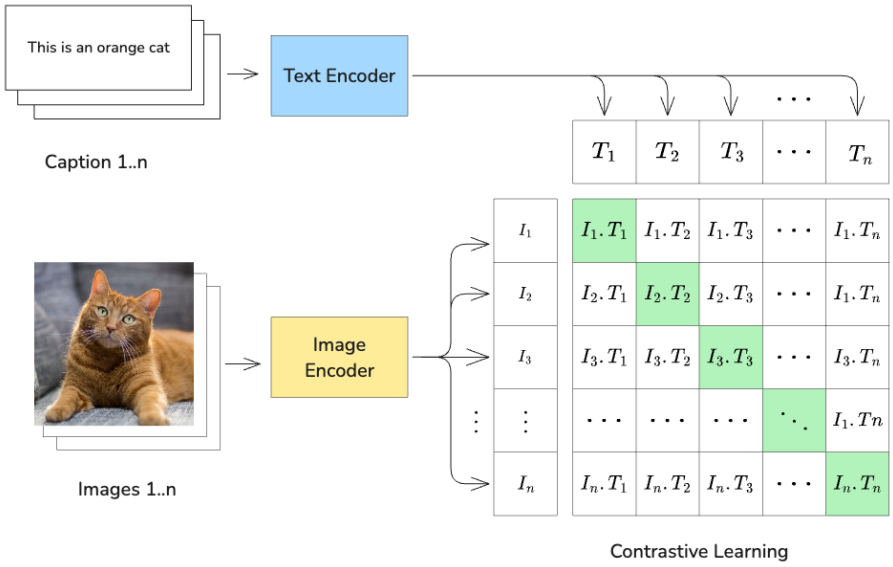


Figure 2.7.: Illustration of CLIP's two-tower architecture and contrastive learning framework, adapted from Radford et al. (2021). Image and text embeddings are project into to a shared embedding space. The model tries to maximize the similarity between matching image-text pairs, while minimizing that of mismatched pairs.

Later approaches extend the two-tower architecture into the *two-leg* architecture where image and text are still encoded separately using distinct encoders, but an additional multimodal encoder (typically a Transformer) is used to fuse their representations. In this design, the hidden states from both the image and text encoders are projected into a shared space, then concatenated and fed into the multimodal encoder. Cross-attention mechanism is used in the encoder to capture the relationship between the image and text modalities. Figure 2.8 illustrates this setup in FLAVA (Singh et al., 2022), a representative example of this architecture.

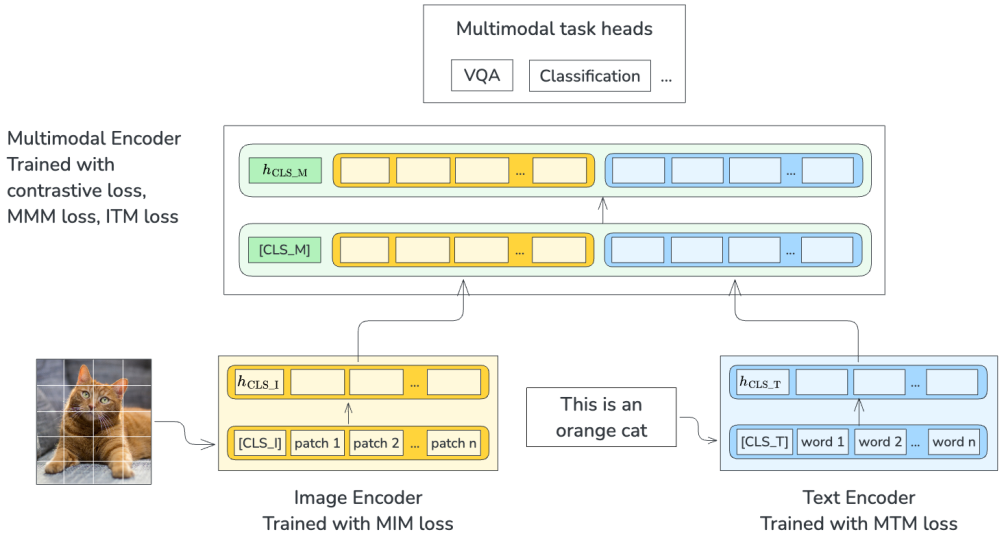


Figure 2.8.: Illustration of FLAVA's two-leg architecture, adapted from Singh et al. (2022). The image is divided into patches and processed by the image encoder. Likewise, the input text is tokenized and processed by the text encoder. Hidden vectors from these encoders are concatenated and input to a multimodal encoder, producing a fused representation $h_{\text{CLS\_M}}$. Task-specific heads use this multimodal representation for downstream tasks.

Florence-2—the VLM examined in this work—follows this two-leg architecture, but with distinct modifications. Unlike FLAVA whose multimodal component consists solely of an encoder and needs task-specific heads, Florence uses both an encoder and a decoder to process the combined text and image representations. The model uses pure text as a unified output format for both visual and language tasks. Further architectural details of Florence are provided in Section 3.3.1.

2.6. Application of HTR in Historical Records Transcription

There is a vast amount of information in historical handwritten text which holds great value for both researchers and the public. Consequently, applying HTR technologies to convert these documents into searchable text is an active area of research.

The current standard pipeline for recognizing historical handwritten text consists of two main steps: text line detection and text recognition. Text line detection is a crucial factor that affects the quality of text recognition (Boillet et al., 2022a), so an effective detection solution is essential. The text line detection process itself can be broken down into two smaller steps: text region detection and text line segmentation. Figure 2.9 demonstrates the output of such a pipeline.

Figure 2.9.: Example output of the HTRFlow pipeline (AI-Riksarkivet, 2023) trained to transcribe historical Swedish manuscript. The pipeline consists of a YOLO model to detect text regions, another YOLO model to segment lines within regions, and a TrOCR model to recognize text.

In practice, the text region detection and line segmentation steps can be performed using standard computer vision models, such as YOLO (Redmon et al., 2016). For the text recognition task, the current state-of-the-art model is TrOCR (Li et al., 2023) and DTrOCR (Fujitake, 2024). A well-known commercial solution that follows this approach is Transkribus (Colutto et al., 2019). Open-source alternatives include OCR4all (Reul et al., 2019) and HTRFlow (AI-Riksarkivet, 2023).

Recent VLMs have demonstrated strong capabilities in performing OCR and HTR tasks, achieving impressive accuracy. According to Li (2024), multimodal models such as GPT-4v (Achiam et al., 2023) and Gemini (Team et al., 2023) have been evaluated for recognizing contemporary handwritten text, showing initial success. However, to the best of my knowledge, these VLMs have not been thoroughly investigated in the context of historical handwritten text.

3. Methodology

In this chapter, I outline my approach to evaluating a traditional HTR pipeline and a VLM on the task of HTR for Swedish historical text. The first part is the experimental setup, where I discuss the data sources, and two different train - validation - test split schemes. Next, I introduce the models to be used in each pipeline, with the rationale for choosing them. Then, I go through the transformations to turn data into the correct format for each model and task. The final section presents the metrics used to evaluate the models and pipelines.

3.1. Experimental Design

3.1.1. Datasets

The datasets used in this work are Swedish historical manuscripts, all publicly available on the Hugging Face page of the Swedish National Archives.[1] All images are scans that feature either a beige or gray background. Each image is paired with an XML file in the PAGE-XML format (Pletschacher and Antonacopoulos, 2010), which provides detailed annotations including text regions, lines, reading order, and text content.

Below is the list of datasets:

Table 3.1.: Overview of datasets used

Index	Dataset name	Background	Images
1	bergskollegium_advokatfiskalskontoret_seg	gray	53
2	bergskollegium_relationer_och_skrivelser_seg	beige	1,497
3	frihetstidens_utskottshandlingar	gray	243
4	gota_hovratt_seg	beige	51
5	jonkopings_radhusratts_och_magistrat_seg	gray	39
6	krigshovrattens_dombocker_seg	gray	344
7	goteborgs_poliskammare_fore_1900	beige	5,408
8	svea_hovratt_seg	gray	1,243
9	trolldomskommissionen_seg	gray	766
	Total		**9,644**

3.1.2. Train - Validation - Test Splits

The data is divided into train, validation, and test splits using two schemes (Table 3.2). The first scheme is to mix all images and divide them, which is a common approach when building proof-of-concept models. In reality, models are often trained on as much data as available, then deployed to process new document images. These documents may have layouts unseen from the training data. To mimic this scenario, I use another

[1] https://huggingface.co/Riksarkivet

scheme in which certain datasets are grouped into the test split, while the remaining datasets are mixed and then divided into train and validation subsets.

Table 3.2.: Specification of two schemes: "Mix then split" (**mixed**) and "Split by source" (**sbs**).

Scheme	Split	Images	Percentage	Source Datasets
mixed	Train	7,811	81.00%	All
	Validation	868	9.00%	All
	Test	965	10.00%	All
sbs	Train	8,290	85.96%	2, 7, 8, 9
	Validation	624	6.47%	2, 7, 8, 9
	Test	740	7.57%	1, 3, 4, 5, 6

3.1.3. Tasks

Model's performance is assessed under the following comparison schemes:

- **At the isolated task level**: I compare specialized computer vision models (vanilla and fine-tuned) with three forms of VLM: vanilla form, **fine-tuned for a specific task**, and **fine-tuned for several tasks**. The tasks to be evaluated include: text region detection, text line detection, line segmentation, and OCR.
- **At the pipeline level (chain of tasks)**: I compare pipelines comprised of traditional models, pipelines of different VLMs fine-tuned for a specific task, and pipelines of a single VLM fine-tuned for several tasks. The pipelines will perform end-to-end OCR, and are evaluated using OCR metrics.

All experiments are done with both **mixed** and **sbs** schemes mentioned in Section 3.1.2. To ensure the models have not seen the test data, all models are retrained from the vanilla checkpoints obtained from the official release channels. Details of the training procedures can be found in Appendix A.

3.1.4. Hardware

All models are trained using the Snowy cluster provided by Uppsala University, with the following specification: CPU: 2x Xeon E5-2660 2.2 GHz, 16 cores; Memory: 128GB; GPU: Tesla T4 (16GB VRAM).

3.2. Traditional Pipeline

3.2.1. Model Details

In the traditional pipeline, I use Ultralytics's YOLO11 (Jocher and Qiu, 2024) for text region detection, line detection, and line segmentation. To be precise, the text region detection and line detection models are the `yolo11m` variant (20.1 million parameters), trained for the object detection task. The inference output of the model on one image is a list of bounding boxes. The line segmentation model is the `yolo11m-seg` variant (22.4 million parameters), trained for the instance segmentation task. The inference output of this model is a list of segmentation masks.

TrOCR (Li et al., 2023) is used for the OCR task. The checkpoint to be used can be obtained from Microsoft's Hugging Face page,[2] which is a variant fine-tuned for

[2] https://huggingface.co/microsoft/trocr-base-handwritten

handwritten text recognition and has 333 million parameters. The input images are single text-line images.

Due to computational resource constraints, all of the chosen models are medium or base variants, which I believe represent a good trade-off between performance and computational cost.

3.2.2. Pipeline Design

Figure 3.1 illustrates the basic "traditional pipeline" for processing a page image. The pipeline consists of three models, each responsible for a specific task, and a post-processing step to determine the reading order of the text in the image. The output of each model is processed and passed as input to the next step.

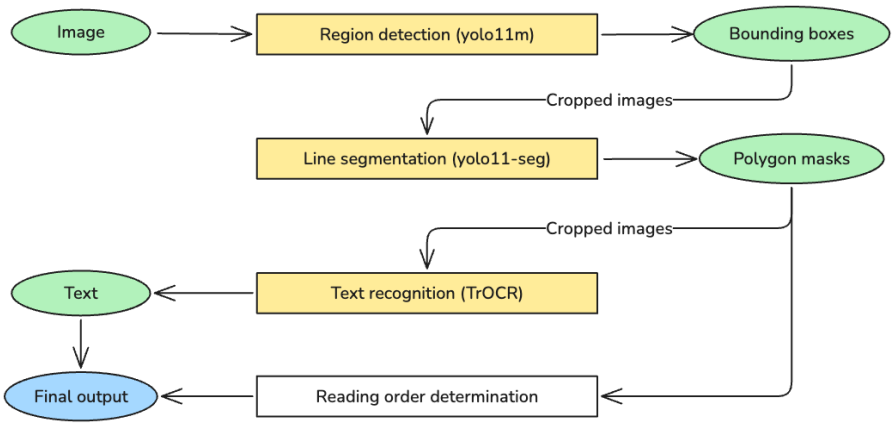


Figure 3.1.: The "traditional" HTR pipeline. First, `yolo11m` detects text regions in the image and outputs bounding boxes. The image is then cropped using the box coordinates and passed to the segmentation model. `yolo11-seg` outputs segmentation polygons, which are used to further crop the regions into lines. These line images are then fed into `TrOCR` for text recognition. The segmentation information is also used to determine the reading order using heuristic rules.

A variant pipeline with only two steps (text line detection and text recognition, without the intermediary text region segmentation step) is also evaluated.

3.3. VLM Pipeline

3.3.1. Model Architecture

For the VLM pipeline, I use Florence-2 (Xiao et al., 2023), specifically the base variant that has already been fine-tuned for a collection of tasks, such as object detection, segmentation, and OCR. This variant can be obtained from Microsoft's Hugging Face page.[3] It has 0.23 billion parameters.

Florence-2 uses DaViT (Ding et al., 2022) as its image encoder and BART (Lewis et al., 2019) as its text encoder. The embeddings produced by these encoders are concatenated and fed into a multimodal encoder-decoder transformer. All vision and language tasks are formulated as text generation problems: given an input image and a task-specific prompt, the model generates a **pure text output response**. This unified task formulation enables training using standard cross-entropy loss, as in typical language modeling. Figure 3.2 illustrates the overall architecture of the model.

[3] https://huggingface.co/microsoft/Florence-2-base-ft

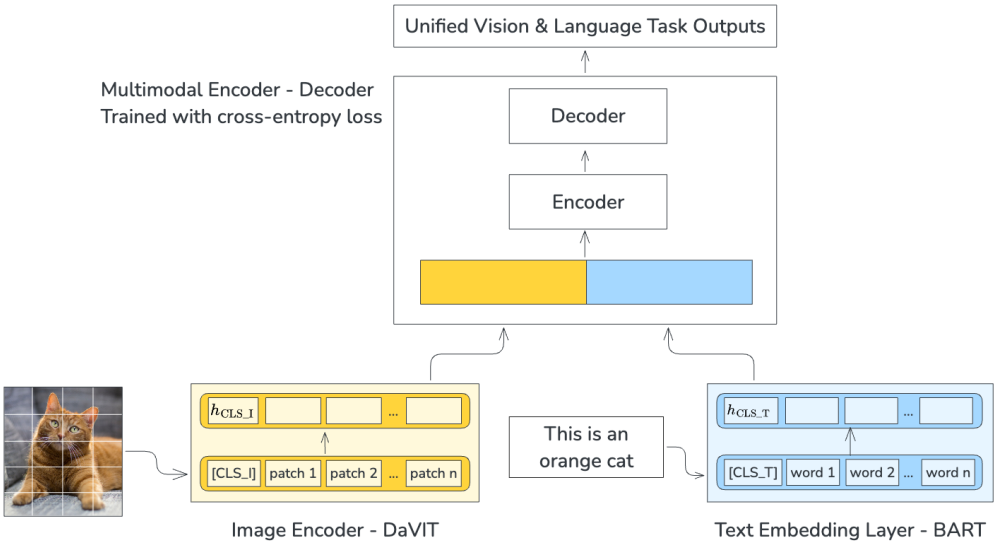


Figure 3.2.: Overview of Florence-2's two-leg architecture. In contrast to FLAVA (Figure 2.8) which uses only a multimodal encoder followed by task-specific heads, Florence-2 uses a full encoder-decoder transformer to process combined image and text inputs, producing a unified text output for visual and language tasks.

Initially, PaliGemma (Steiner et al., 2024) and Sa2VA (Yuan et al., 2025) were also considered as candidates for the experiments, since they are both capable of handling multiple tasks via user prompts. However, the smallest variant of PaliGemma has 3 billion parameters, making it unsuitable for the available hardware. Additionally, its range of supported tasks is not as diverse as that of Florence-2, and its prompt interface is less well-structured. Sa2VA's smallest variant has 1 billion parameters, but it requires the Flash Attention mechanism (Dao, 2024) for efficient training, which is not supported by the hardware.

On the other hand, Florence-2 is a suitable candidate due to its size and functionality. With only 0.23 billion parameters, the model is lightweight enough to be trainable with the university's hardware. In addition, the fine-tuned variant of the model supports several visual and language tasks out of the box, such as object detection, object segmentation, and OCR.

3.3.2. Model Input and Output

Florence receives an image and a special *task prompt* as input. For example, the task prompt for captioning task is `<CAPTION>`, object detection task corresponds to the `<OD>` task prompt, and OCR corresponds to the `<OCR>` task prompt.

For tasks that require a user's natural language prompt along with the task prompt (such as *referring expression segmentation*, where the user asks the model to segment an object that they refer to), the task and user prompt are concatenated to form the text input. For example, the full input corresponding to this task is as follows:

```
task_prompt = "<REFERRING_EXPRESSION_SEGMENTATION>"
user_prompt = "A person"
text_input = "<REFERRING_EXPRESSION_SEGMENTATION>A person"
```

As mentioned, Florence produces raw textual output. For language tasks such as image captioning, the output is a simple sentence. For visual tasks like object detection, the output is a specially formatted string containing "location tokens" that represent *quantized* coordinates. Specifically, Florence-2 quantizes the image dimensions by dividing the width and height into 1000 discrete bins, represented by tokens `<loc1>` to

<loc1000>, effectively creating a 1000 × 1000 grid. Each pixel in the image is assigned to one of these grid cells. A post-processing step then converts the raw text output into structured predictions, including decoding the quantized location tokens into pixel-level coordinates. Below is an example output for an object detection task:

```
# Raw text output:
"</s><s>person<loc_439><loc_104><loc_774><loc_919>
        skis<loc_327><loc_861><loc_962><loc_938></s>"
# Parsed output:
{'<OD>': {'bboxes': [[281.2799, 44.4125, 495.6799, 390.7875],
                     [209.5999, 366.1375, 616.0..., 398.8625]],
            'labels': ['person', 'skis']}}
```

A visual example of Florence's output for visual tasks can be found in Appendix C.

While Florence-2 can generate bounding boxes for multiple objects in a single inference, it is limited to only a single segmentation mask per inference. This constraint likely stems from the model's maximum output length of 1024 tokens. Representing a bounding box requires only four location tokens: two points (top-left and bottom-right) times two tokens per point (location for x and y). In contrast, a segmentation mask requires hundreds of location tokens to represent the polygon outline of a single object. Consequently, the 1024-token output limit becomes a bottleneck when dealing with tasks that involve many segmented objects, such as text line detection. Comparing to YOLO which can output several masks at once, this is a major drawback of Florence-2, and affects the design of the end-to-end HTR pipeline.

3.3.3. Pipeline Design

Figure 3.3 illustrates a pipeline that utilizes Florence-2 models fine-tuned for each task. As mentioned, Florence-2 cannot output multiple segmentation masks simultaneously, so the pipeline needs to iterate through each detected line to perform text region segmentation, which adds more overhead to the pipeline.

In the experiments with Florence, I will also compare this two-step pipeline with a three-step pipeline (text region detection, then line detection within region, and finally text recognition).

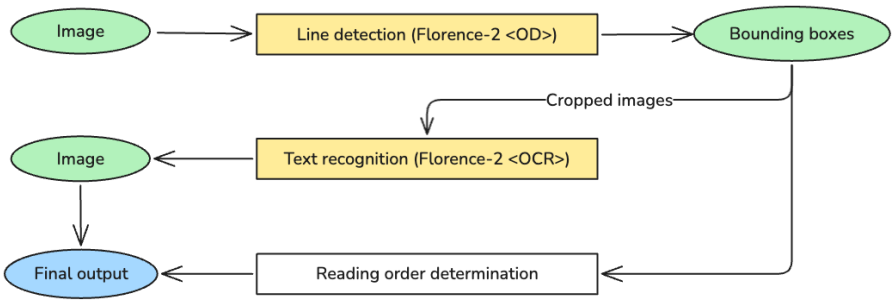


Figure 3.3.: The two-step Florence-2 HTR pipeline. First, a model uses <OD> task to detect lines, and output bounding boxes. The box coordinates is used to crop images into lines, and then another model (or the same model) use the <OCR> task to recognize text. Line bounding boxes coordinates are used to determine the reading order using heuristic rules.

3.4. Data Processing

In the text object detection task, the model receives an image and outputs a list of rectangular bounding boxes indicating areas that contain text (see Figure 3.4). The

detection targets can be either entire text regions or individual lines. For this task, the input image does not require any transformation.

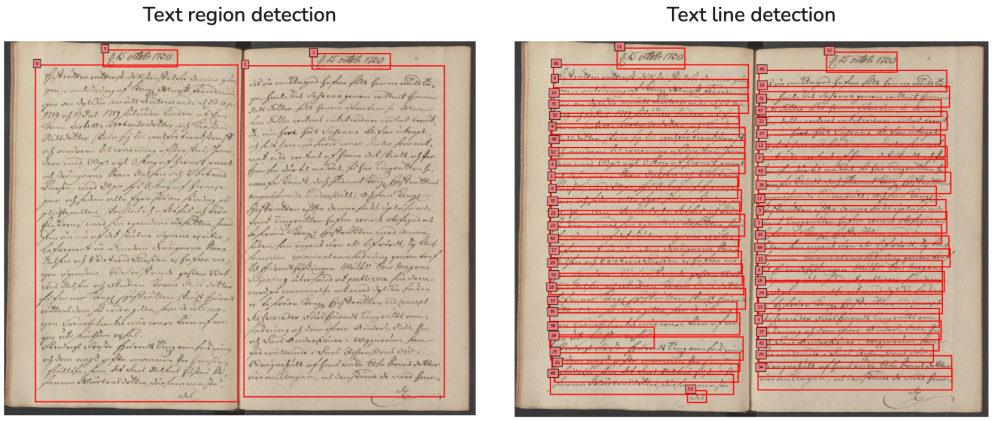


Figure 3.4.: Text region and text line detection

In the line segmentation task, the model receives an image cropped with the region bounding box, and produces a list of several segmentation masks covering the lines (see Figure 3.5). In this work, only YOLO trained specifically for segmentation can perform this task.

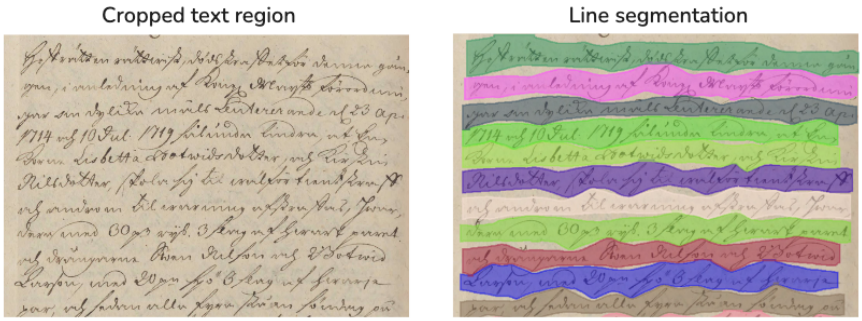


Figure 3.5.: Line segmentation done by YOLO

Finally, for the text recognition task, the model receives a cropped line image and outputs a text transcription. A basic approach is to crop the line using its rectangular bounding box, but the segmentation mask can also be applied to remove overlapping text fragments from neighboring lines (see Figure 3.6).

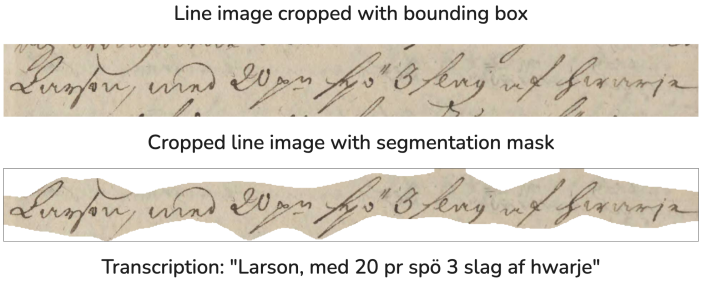


Figure 3.6.: Line cropping to prepare for text recognition

3.5. Evaluation

3.5.1. Object-level evaluation

Traditionally, **pixel-level** metrics are used to evaluate detection and segmentation performance. However, pixel-level metrics give no information about the number of correctly predicted, missed, or split objects, which makes them inadequate (Boillet et al., 2022b). Therefore, detection and segmentation metrics at the **object-level** are also important, as proposed by Tarride et al. (2019).

3.5.2. Region and Line Detection Metrics

Object detection involves two sub-tasks: proposing regions (i.e., generating bounding boxes that enclose potential objects) and classifying those regions. Standard evaluation procedures compute region proposal and classification metrics for each class separately, and then average the results to obtain overall performance. In my setup, however, each task targets only a single object class: *region* in text region detection, and *line* in text line detection. As a result, object classification metrics are not included.

To compute object-level detection metrics, first predicted boxes are aligned with ground truth boxes: for each pair of a predicted box and a ground truth box, I calculate the **Intersection over Union** (IoU) ratio, which is the area of overlap between the predicted box (A) and ground truth box (B), divided by the area of their union (see Equation 3.1).

$$\text{IoU} = \frac{\text{area}(A \cap B)}{\text{area}(A \cup B)} \tag{3.1}$$

Two boxes are considered a match if their IoU is at least 0.5. Once alignment is complete, the text detection problem can be treated as a binary classification task, allowing us to compute precision, recall, and F1-score. In this context, precision is defined as the number of matched predicted boxes divided by the total number of predicted boxes, recall as the number of matched predicted boxes divided by the total number of ground truth boxes, and F1-score as the harmonic mean of precision and recall. The matching is **one-on-one**: once a predicted box is matched, it is not used to match with other ground truth boxes.

3.5.3. Line Segmentation Metrics

As mentioned in Section 3.3.2, due to differences in how YOLO and Florence-2 approaches line segmentation, a fair comparison between the two is challenging. Nonetheless, the results of both models will still be presented for informative purposes.

Metrics for the segmentation task include: Intersection over Union (IoU), Dice coefficient, pixel accuracy (PA), mean pixel accuracy (MPA), and region coverage.

As mentioned earlier, IoU is computed as the ratio of the intersection area between the predicted and ground truth masks to the area of their union. The Dice coefficient (Equation 3.2) is a similar overlap measure, defined as twice the intersection area of predicted mask (A) and ground truth mask (B) divided by the total area of both masks:

$$\text{Dice} = \frac{2 \times \text{area}(A \cap B)}{\text{area}(A) + \text{area}(B)} \tag{3.2}$$

From here on let us denote the number of *correctly predicted mask pixels* as TP (true positive), the *correctly predicted background pixels* as TN (true negative), *ground truth mask* as P (positive), and *ground truth background* as N (negative). The total number of image pixels will be P + N.

Pixel accuracy (Equation 3.3) is calculated as the ratio of correctly classified pixels (both foreground (mask) and background) to the total number of pixels in the image.

$$\text{PA} = \frac{TP + TN}{P + N} \tag{3.3}$$

Mean pixel accuracy (Equation 3.4) treats the mask and background as two separate classes. It computes the classification accuracy for each class and averages them:

$$\text{MPA} = \frac{1}{2}\left(\frac{TP}{P} + \frac{TN}{N}\right) \tag{3.4}$$

Region coverage (Equation 3.5) is the area of intersection between the predicted mask (A) and ground truth mask (B), divided by the area of the ground truth mask:

$$\text{Region coverage} = \frac{\text{area}(A \cap B)}{\text{area}(B)} \tag{3.5}$$

For final model evaluation, these metrics are averaged across all segmented lines.

3.5.4. Line-level Text Recognition Metrics

All the following metrics are computed on a per-line basis and averaged across all lines.

The most fundamental metrics for line-level text recognition are Character Error Rate (CER) and Word Error Rate (WER). CER (Equation 3.6) quantifies the number of character-level errors in the predicted text relative to the ground truth, normalized by the total number of characters in the reference:

$$\text{CER} = \frac{S + D + I}{N} \tag{3.6}$$

where S is the number of substitutions, D is the number of deletions, I is the number of insertions, and N is the total number of characters in the ground truth. WER follows the same formula as CER, but considers words instead of characters:

$$\text{WER} = \frac{S + D + I}{N} \tag{3.7}$$

Two other metrics for word-level errors are Bag-of-Words (BoW) Hits and BoW Extras. Let A be the set of predicted words and B be the set of ground truth words. BoW Hits (Equation 3.8) is the ratio of correctly predicted words to the total number of ground truth words:

$$\text{BoW Hits} = \frac{|A \cap B|}{|B|} \tag{3.8}$$

BoW Extras (Equation 3.9) is the ratio of incorrectly predicted words (those not in the ground truth set) to the total number of predicted words:

$$\text{BoW Extras} = \frac{|A \setminus (A \cap B)|}{|A|} \tag{3.9}$$

Both BoW Hits and BoW Extras operate on sets of words, meaning they disregard word order and focus solely on word presence.

4. Results & Discussion

In this chapter, I compare the performance of Florence-2 with that of traditional models across the following tasks: text region detection, text line detection, text line segmentation, and text recognition. The chapter begins with evaluations of individual models on each task. Then, I assess the performance of complete HTR pipelines composed of these models, using only text recognition metrics. Qualitative analysis is also conducted to illustrate the strengths and weaknesses of the Florence pipeline.

4.1. Individual Tasks Performance

4.1.1. Text Region Detection

In object detection tasks, precision measures the proportion of predicted bounding boxes that correctly match ground truth boxes, while recall measures the proportion of ground truth boxes that are correctly detected. The F-score is the harmonic mean of precision and recall. Region coverage quantifies how much of the ground truth area is covered by the predicted boxes. Table 4.1 shows the performance of Florence-2 and YOLO before and after fine-tuning.

Table 4.1.: Text region detection performance of baseline vanilla (vnl) vs. fine-tuned (ft) models. Vanilla models predict the entire image as a single object, resulting in artificially high precision, low recall, and high region coverage. Vanilla YOLO seems to be a bit better than vanilla Florence. After fine-tuning, both models have comparable results, although Florence is a bit behind on Recall on the **sbs** scheme.

Variant	Split	Model	Precision	Recall	Fscore	Avg. Coverage
vnl	mixed	florence	1.0000	0.0023	0.0047	0.9996
		yolo11m	1.0000	0.0135	0.0265	0.9780
	sbs	florence	1.0000	0.0004	0.0007	0.9999
		yolo11m	1.0000	0.0387	0.0744	0.9766
ft	mixed	florence	1.0000	0.9513	0.9751	0.9813
		yolo11m	1.0000	0.9478	0.9732	0.9695
	sbs	florence	1.0000	0.7078	0.8289	0.9572
		yolo11m	1.0000	0.7424	0.8522	0.9406

Unsurprisingly, both models perform poorly in their vanilla form. They often predict the entire image as a single object, resulting in artificially high precision (as the oversized bounding box encompasses most ground truth regions), low recall (since only one or very few boxes are predicted), and high region coverage (due to the large overlap with ground truth areas). Although the difference is not substantial, the vanilla YOLO model slightly outperforms Florence-2.

Once fine-tuned, Florence-2 performs comparably to YOLO across both data split schemes, albeit Recall on the **sbs** scheme of Florence-2 is a bit lower. Performance under the **mixed** scheme is generally higher than under the **sbs** scheme. This is expected, as

the **mixed** scheme allows the training set to include images similar in characteristics to those in the test set. In contrast, the **sbs** scheme separates datasets by source, resulting in training and test sets that may differ significantly in page color, layout, and handwriting style.

4.1.2. Text Line Detection

In this task, model performance metrics are also precision, recall, F-score, and region coverage. As shown in Table 4.2, the performance of both vanilla Florence-2 and YOLO is similarly poor. This task requires the models to detect individual text lines directly from the full page, and since line bounding boxes are considerably smaller than region-level boxes, with IoU threshold of 0.5, oversized predicted bounding boxes will not be matched with line-level ground truth boxes.

Once fine-tuned, however, both models perform comparably well. Interestingly, the performance of line detection under the **sbs** scheme is only slightly lower than under the **mixed** scheme. For example, Florence's line detection performance on **sbs** is only 7% lower than on **mixed**, while its gap in region detection performance is about 20% (Table 4.1). Overall performance of line detection is also significantly better than text region detection. A possible explanation is that compared to text regions, individual text lines are more consistent visual units.

Table 4.2.: Text line detection performance of vanilla (vnl) vs. fine-tuned (ft) models. Vanilla models tend to produce oversized bounding boxes that encompass the whole image, so with IoU threshold of 0.5, the predicted boxes are not matched with very small line-level boxes. This results in zero precision and recall, despite a very high Avg. Region Coverage. After fine-tuning, both models perform comparably well.

Variant	Split	Model	Precision	Recall	Fscore	Avg. Coverage
vnl	mixed	florence	0.0000	0.0000	0.0000	0.9996
		yolo11m	0.0000	0.0000	0.0000	0.9820
	sbs	florence	0.0000	0.0000	0.0000	1.0000
		yolo11m	0.0000	0.0000	0.0000	0.9697
ft	mixed	florence	1.0000	0.9623	0.9808	0.9745
		yolo11m	1.0000	0.9833	0.9916	0.9819
	sbs	florence	1.0000	0.8987	0.9466	0.9682
		yolo11m	1.0000	0.9217	0.9592	0.9725

4.1.3. Line Segmentation

In this section, the performance of fine-tuned YOLO and Florence-2 are reported separately, as the approach for this task differs between the two models. With YOLO, the input is a cropped image of a detected text region, and the model returns segmentation masks for all lines within that region. Table 4.3 presents the performance metrics for this setup.

Table 4.3.: YOLO's performance on line segmentation within region performance

Split	IoU	Dice	Pixel Acc.	Mean Pixel Acc.	Coverage
mixed	0.7580	0.8605	0.9900	0.9309	0.8687
sbs	0.6577	0.7878	0.9650	0.8703	0.7819

In contrast, for Florence-2, the input is a line image cropped with its bounding box, and the model generates a segmentation mask for the text region within that line. As shown in Table 4.4, the model has good performance in this setup.

Table 4.4.: Florence's performance on text region segmentation within cropped line image

Split	IoU	Dice	Pixel Acc.	Mean Pixel Acc.	Coverage
mixed	0.7831	0.8757	0.8688	0.8619	0.8754
	0.7961	0.8838	0.8732	0.8638	0.9012

4.1.4. Text Recognition

This section reports the performance of Florence-2 and TrOCR on the text recognition task using text line images. These input images are either cropped by bounding boxes (**line_bbox**) or segmentation masks (**line_seg**), as shown in Figure 3.6. Therefore, OCR metrics reported in this section are calculated on **line** level and average across all line images in the test set. In contrast, metrics in Section 4.2 will be reported on **page** level.

Performance on the Same Input Type

Table 4.5 shows the performance of vanilla and fine-tuned models on the **line_bbox** image type. As with other tasks, both vanilla models perform poorly out of the box, with high CER and WER, low BoW Hits, and high BoW Extras. After fine-tuning, both models show substantial improvement across all metrics. Across two split schemes, Florence achieves lower CER and WER than TrOCR, especially in **sbs**, it also has higher BoW Hits (0.6432 vs. 0.6136) and lower BoW Extras (0.4336 vs. 0.4677).

Table 4.5.: Performance of models on **line_bbox** data. Vanilla models perform poorly, while fine-tuned models are much better. Florence slightly outperforms TrOCR: lower CER, WER and BoW Extras, and higher BoW Hits. The better metric value within a cell is highlighted for fine-tuned models.

Variant	Split	Model	CER	WER	BoW Hits	BoW Extras
vnl	mixed	florence	0.7611	0.9829	0.1680	0.9813
		trocr	0.6395	0.9810	0.1728	0.9752
	sbs	florence	0.8048	0.9934	0.2037	0.9930
		trocr	0.7107	0.9953	0.1643	0.9931
ft	mixed	florence	**0.0974**	0.3360	**0.7648**	0.3289
		trocr	0.1424	0.3372	0.8230	0.3309
	sbs	florence	**0.1438**	**0.4426**	**0.6432**	**0.4336**
		trocr	0.1926	0.4797	0.6136	0.4677

General patterns on **line_seg** data is similar to that of **line_bbox** data: poor performance with vanilla models, much better performance when fine-tuned, and Florence-2 performs slightly better than TrOCR.

Table 4.6.: Performance of models on **line_seg** data. Similarly, vanilla models perform poorly, while fine-tuned models are much better. Florence slightly outperforms TrOCR. The better metric value within a cell is highlighted for fine-tuned models.

Variant	Split	Model	CER	WER	BoW Hits	BoW Extras
vnl	mixed	florence	0.7365	0.9827	0.1776	0.9817
		trocr	0.6176	0.9774	0.1734	0.9720
	sbs	florence	0.7959	0.9938	0.2021	0.9934
		trocr	0.6841	0.9926	0.1637	0.9901
ft	mixed	florence	**0.0843**	**0.3042**	**0.8257**	**0.2976**
		trocr	0.1406	0.3364	0.8142	0.3298
	sbs	florence	**0.1584**	**0.4662**	**0.6103**	**0.4565**
		trocr	0.1896	0.4767	0.6083	0.4615

Effects of Image Types on Text Recognition

Figure 4.1 shows that in the **mixed** scheme, segmentation slightly improves Florence-2's results, with lower CER, WER, BoW Extras, and higher BoW Hits.

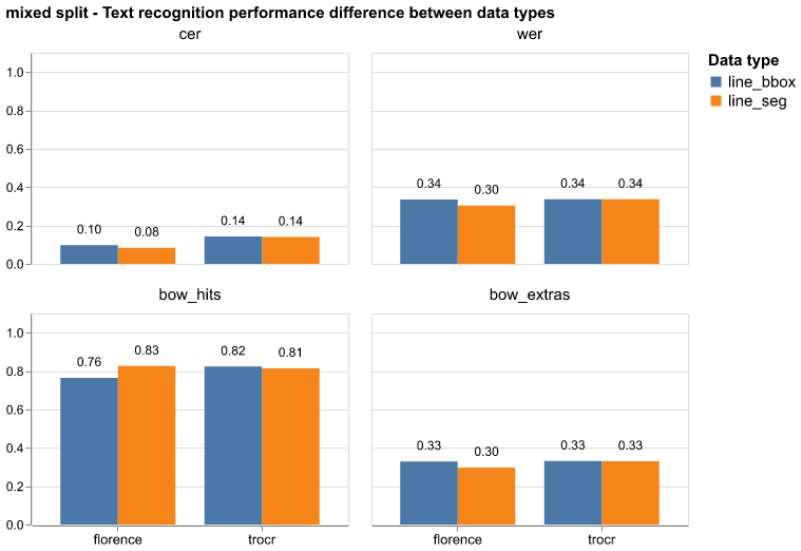


Figure 4.1.: Comparing performances on two data types in the **mixed** scheme.

However, as shown in Figure 4.2, under the **sbs** scheme, performance is slightly worse with segmentation, although not by a significant margin. This suggests that for Florence-2, line segmentation may not help when handling manuscript images with unseen layouts or writing styles.

In contrast to Florence, TrOCR shows consistent performance across both data types and split schemes. This may suggest that the text region detection step in the traditional pipeline could be skipped, allowing us to go directly from line detection to text recognition (provided that the line detection step is good enough).

Overall, I conclude that after fine-tuning, Florence achieves comparable performance to YOLO and TrOCR on individual tasks. In the next section, complete end-to-end pipelines will be evaluated with HTR metrics.

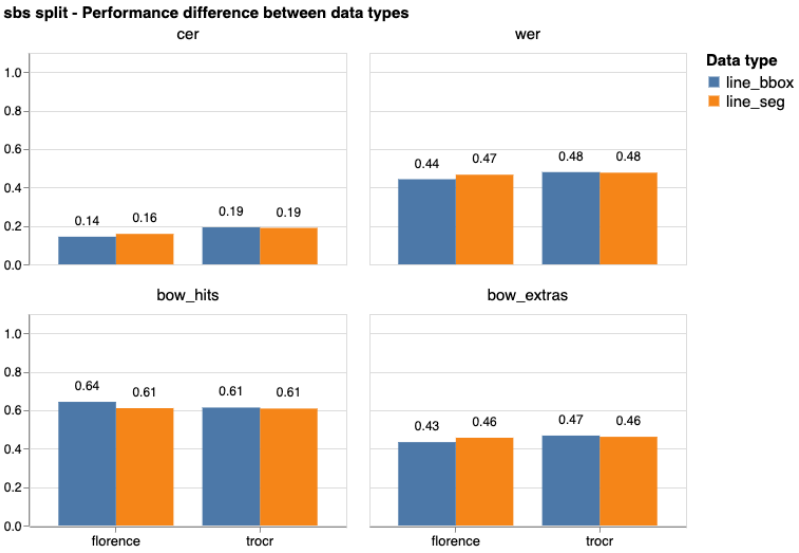


Figure 4.2.: Comparing performances on two data types in the **sbs** scheme.

4.1.5. Effect of Sequential Fine-tuning

To construct an end-to-end HTR pipeline using a single Florence model, I employed a sequential fine-tuning approach. The model was first fine-tuned on the line detection task, followed by additional fine-tuning on the OCR task. The hypothesis was that the model would retain its capability for line detection while acquiring acceptable performance in OCR. However, experimental results did not support this assumption. As shown in Table 4.7, the performance of this model on the OCR task is significantly inferior to the model trained exclusively for OCR.

Split Type	CER	WER	BoW Hits	BoW Extras
mixed	0.4480	0.3382	0.7226	0.3325
sbs	0.4852	0.5361	0.5176	0.5251

Table 4.7.: Performance of the Florence model on the OCR task after sequential fine-tuning for line detection followed by OCR. Overall, the performance is lower compared to a model trained exclusively for OCR.

Figure 4.3 also shows that the model gradually loses its ability to perform line detection, as indicated by the steadily decreasing recall metric. This suggests that the model tends to predict oversized bounding boxes. Given the poor performance of this approach, I decided not to proceed further with the one-model setup.

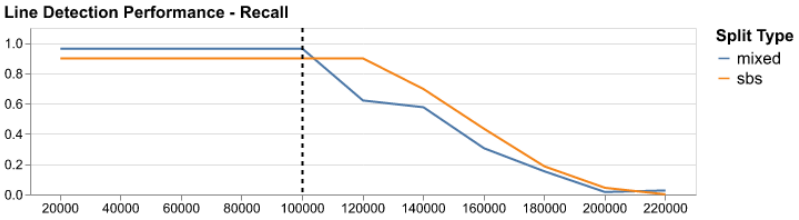


Figure 4.3.: Line detection performance of the Florence model after OCR fine-tuning. After the 100,000th step, Recall drops sharply, indicating detection performance has degraded.

4.2. Full Pipeline Performance

To thoroughly examine how different steps interact and affect overall pipeline performance, several pipeline variants are evaluated. The list of pipelines is as follows:

Table 4.8.: Pipeline list

Type	Steps	Steps Notation
Traditional	Region detection, Line segmentation, Text recognition	region_od, line_seg, ocr
	Region detection, Line detection, Text recognition	region_od, line_od, ocr
	Line detection, Text recognition	line_od, ocr
Florence	Region detection, Line detection, Text recognition	region_od, line_od, ocr
	Line detection, Text region segmentation, Text recognition	line_od, line_seg, ocr
	Line detection, Text recognition	line_od, ocr

After the object detection steps, detection results are sorted using a simple heuristic. First, the image is guessed to be a two-page spread or not. Detected objects (regions, text lines) are determined as belonging to the left page or the right page. Finally, the objects are sorted in this order: top margin, main text, bottom margin, and side margin. An example of this heuristic can be found in Appendix B.

4.2.1. Traditional Pipelines

Table 4.9 presents the results for the traditional pipelines. Overall, performance on the **sbs** scheme remains lower than on the **mixed** scheme, consistent with the trends observed in the single-task evaluations. In Section 4.1.4, I hypothesized that the region detection step could be skipped, i.e. going directly from line detection to text recognition. The result here seems to confirm this is possible: all metrics of the **line_od → ocr** pipeline is comparable if not better than the full pipeline (**region_od → line_seg → ocr**). On the other hand, substituting line detection for line segmentation in the three-step pipeline seems to degrade the performance.

Table 4.9.: Performance of traditional pipelines. Among three variants, the **line_od → ocr** pipeline performs the best.

Pipeline Steps	Split	CER	WER	BoW Hits	BoW Extras
region_od, line_seg, ocr	mixed	0.1722	0.3078	0.7672	0.2158
	sbs	0.2693	0.5121	0.5938	0.3962
region_od, line_od, ocr	mixed	0.3426	0.4716	0.7640	0.2171
	sbs	0.3716	0.5946	0.5903	0.4218
line_od, ocr	mixed	0.1556	0.2796	0.7922	0.1883
	sbs	0.2644	0.5094	0.5911	0.3957

4.2.2. Florence-based Pipelines

Table 4.10 presents the results for Florence-based pipelines. Overall, performance is strong under the **mixed** split scheme but declines under the **sbs** scheme, which is consistent with the trend. The additional text region segmentation step added before text recognition offers no performance gain, while increasing computational cost. Adding a region detection step before line detection only degrades the performance.

Table 4.10.: Among three Florence pipelines, the **line_od → ocr** one performs the best.

Pipeline Steps	Split	CER	WER	BoW Hits	BoW Extras
region_od, line_od, ocr	mixed	0.1646	0.2656	0.8133	0.1486
	sbs	0.5729	0.7052	0.4813	0.3470
line_od, line_seg, ocr	mixed	0.1195	0.2485	0.8103	0.1793
	sbs	0.2332	0.4900	0.6003	0.3882
line_od, ocr	mixed	0.1105	0.2261	0.8342	0.1591
	sbs	0.2219	0.4649	0.6259	0.3581

4.2.3. Pipelines Starting with Line-level Detection

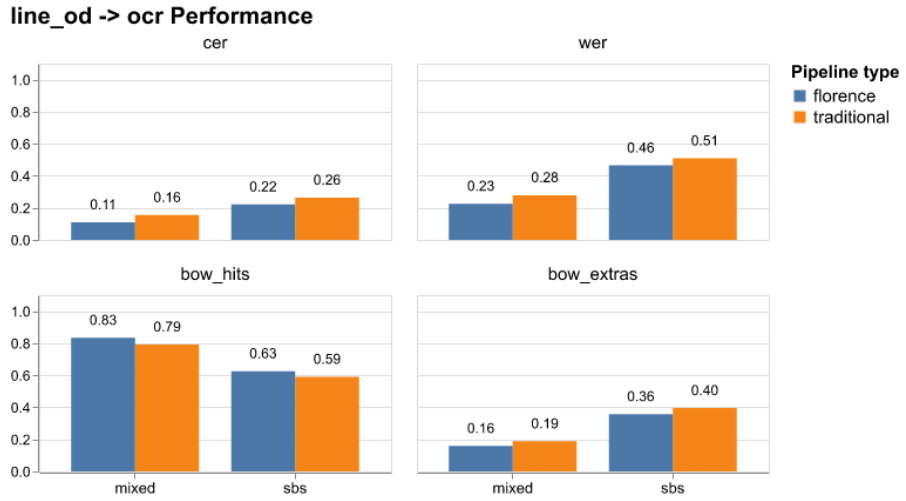


Figure 4.4.: Comparing pipelines with line_od → ocr setup. Florence outperforms the traditional pipeline, with lower CER, WER, BoW Extras, and higher BoW Hits.

Figure 4.4 shows that in a set up with only two steps (line detection on the page followed by text recognition) Florence-2 outperforms the traditional pipeline.

4.2.4. Pipelines Starting with Region-level Detection

Figure 4.5 shows that for the **mixed** split, Florence-based pipeline achieves performance comparable to that of the traditional pipeline with line segmentation, which is the strongest among the traditional approaches. On the other hand, the performance of Florence on the **sbs** split is the lowest among all pipelines (Figure 4.6).

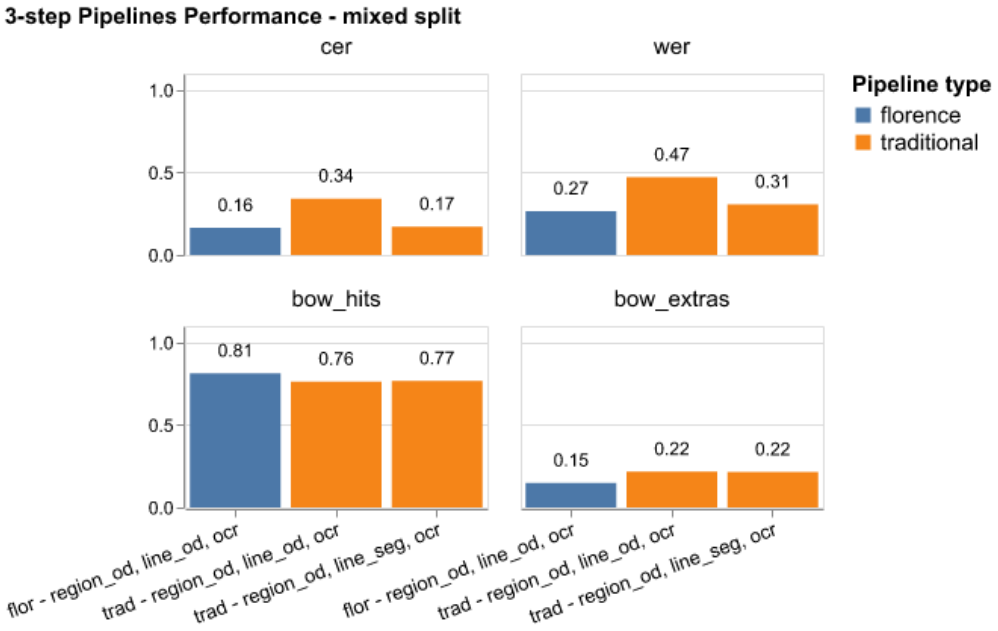


Figure 4.5.: Performance of 3-step pipelines on mixed split. Florence pipeline matches the performance of the best traditional pipeline.

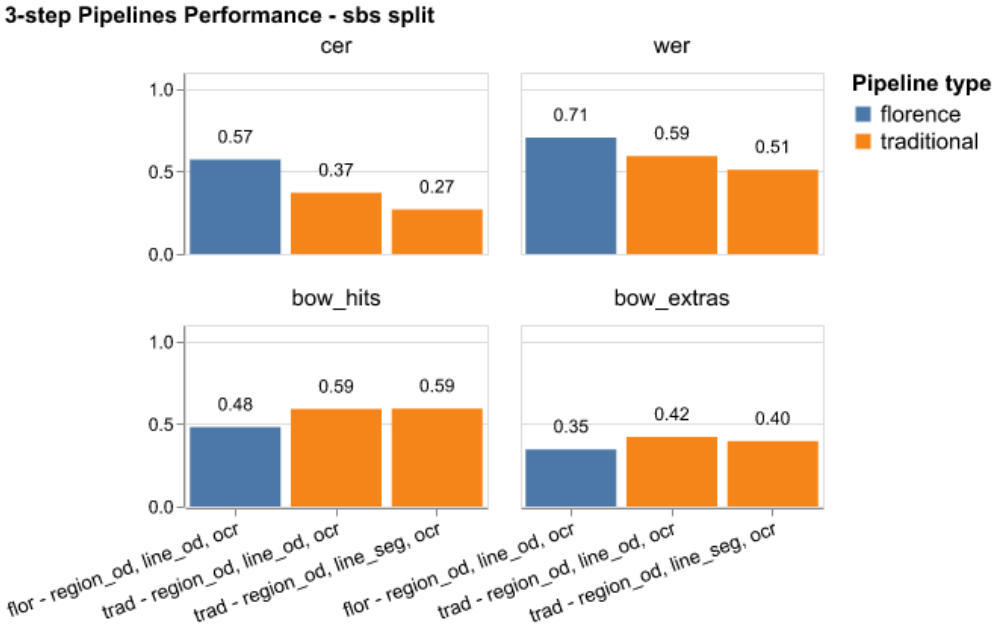


Figure 4.6.: Performance of 3-step pipelines on sbs split. Florence's performance is the lowest among all pipelines.

4.2.5. Comparing Best Pipelines

Based on the analyses presented, I conclude that the best Florence-based pipeline is the **line_od → ocr**, while the best traditional pipeline is **region_od → line_seg → ocr**.

Figure 4.7 shows that the two-step Florence pipeline outperforms the three-step traditional pipeline when considering quantitative OCR metrics such as CER, WER, BoW Hits, and BoW Extras. In Section 4.3, I will provide a further discussion on the differences between the three-step and two-step pipelines.

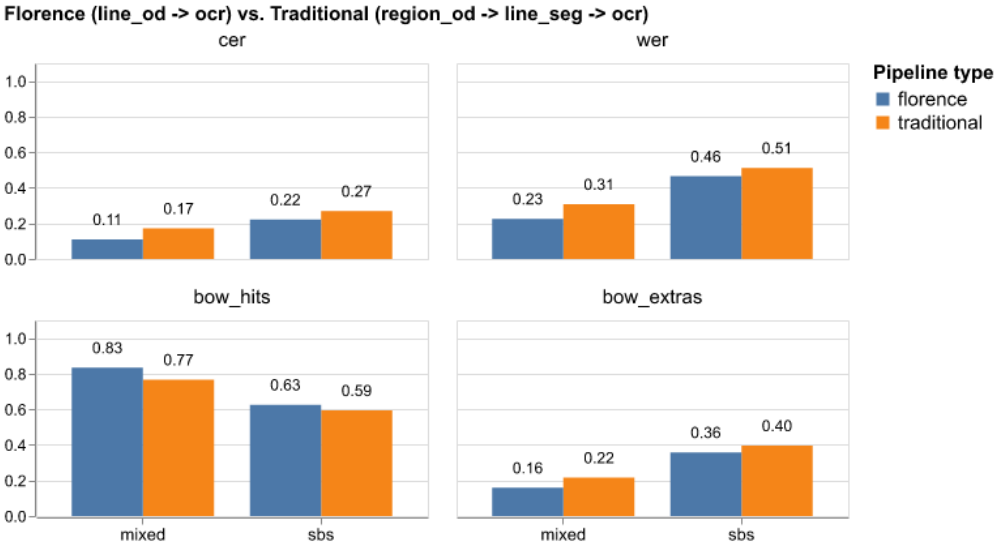


Figure 4.7.: Comparing Florence's 2-step pipeline and traditional's 3-step pipeline. Florence outperforms the traditional pipeline.

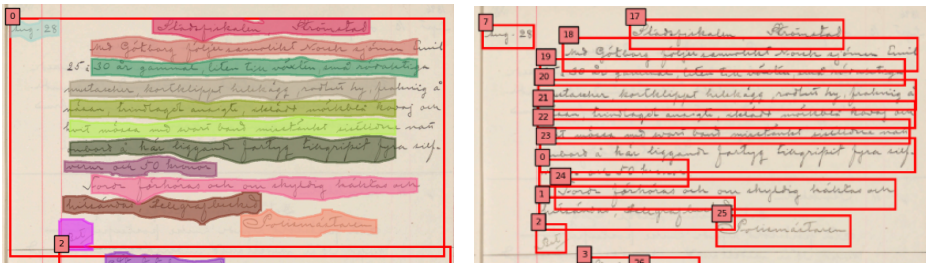
4.3. Qualitative Analysis

4.3.1. Success Cases

The Florence pipeline generally performs well on clean, well-structured page images with straightforward layouts and legible handwriting. In such cases, its robust OCR capability can surpass that of the TrOCR model. Additionally, the simpler two-step pipeline—which begins with line detection—introduces fewer potential failure points.

In some scenarios, the traditional pipeline with segmentation may generate incorrect masks and send flawed input to the OCR stage, while the Florence pipeline can be more successful (see Figure 4.8). It is also possible that using bounding boxes to identify and crop text lines offers more consistency than segmentation-based methods.

More illustrations of cases where Florence succeeds when the three-step pipeline fails can be found in Appendix E.



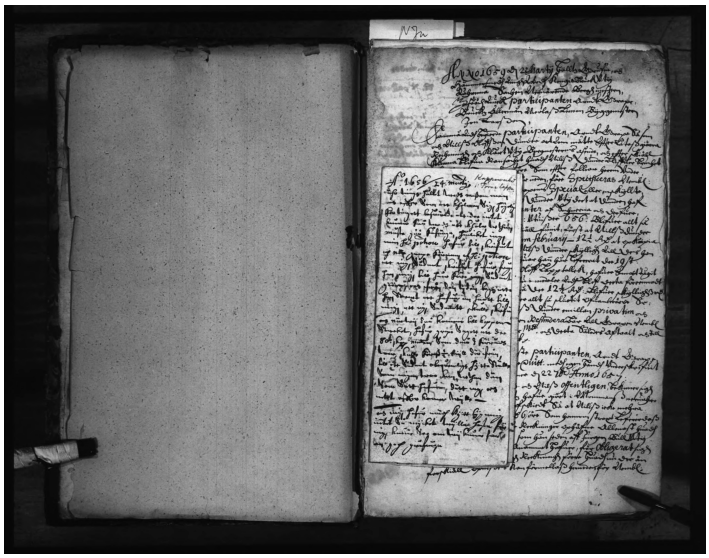
(a) Traditional pipeline's prediction **(b)** Florence pipeline's prediction

Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.3171	0.4794	0.6726	0.3314
Traditional	0.5961	0.8876	0.5238	0.4094

Figure 4.8.: Example of cascading failure in the traditional pipeline. In Image 4.8b, the segmentation masks do not fully cover some lines. As a result, the OCR step has to operate on incomplete line images. A pipeline with line-level detection circumvents this issue by cropping lines with bounding boxes, which is more robust.

4.3.2. Failure Cases

When an image is extremely noisy (for example, the image is dirty, has complex layouts, or is a patchwork of multiple small paper fragments), both the traditional and Florence pipelines will fail (Figure 4.9).

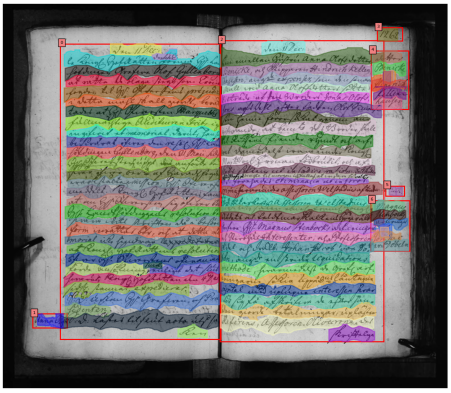


Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.6848	0.9771	0.0914	0.8974
Traditional	0.6862	0.9862	0.1343	0.8694

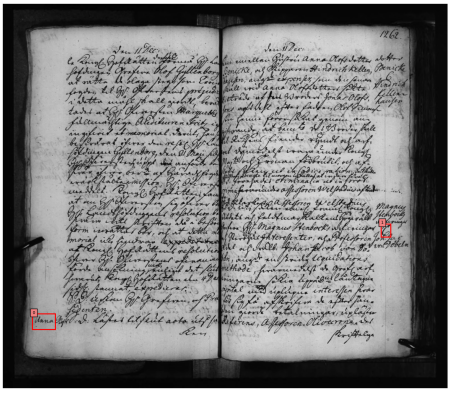
Figure 4.9.: A failure case for both pipelines. The layout in this image is complex: a patch of paper is embedded within the main page, and some text lines span both the patch and the main page. The image quality is poor, with dark regions obscuring portions of the text, and the writing is dense. This image also confuses the sorting heuristic, as the right page is mistakenly interpreted as two separate pages (due to the vertical dark line at the center).

A common source of failure in both pipelines is the reading order determination step. As described in Section 4.2, the current heuristic is relatively simple and cannot capture layouts that deviate from the predetermined reading order. For instance, in some documents, the main body text may appear on the outer sides of the page, while marginal notes are placed at the center.

In some cases, Florence fails entirely. In the case illustrated in Figure 4.10, the traditional method successfully detects regions, segments lines, and recognizes text. Florence, however, fails at line detection, resulting in no input for the OCR step. The exact cause of the failure is unclear. This may be due to the image being underrepresented in the training set (e.g., grayscale, with show-through from the reverse side). More illustrations of failure cases can be found in Appendix F.



(a) Traditional pipeline's prediction



(b) Florence pipeline's prediction

Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.9967	0.9969	0.0031	0.5000
Traditional	0.1495	0.3960	0.6748	0.3491

Figure 4.10.: An example where the traditional pipeline succeeds, while Florence fails right at line detection.

4.3.3. The Cases with Low CER but High WER

A typical performance behavior of a success case is low CER, low WER, high BoW Hits, and low BoW Extras. However, there are cases with low CER, but high WER, indicating that these documents may have longer words where the pipelines can easily make mistakes in one character and invalidate the whole word.

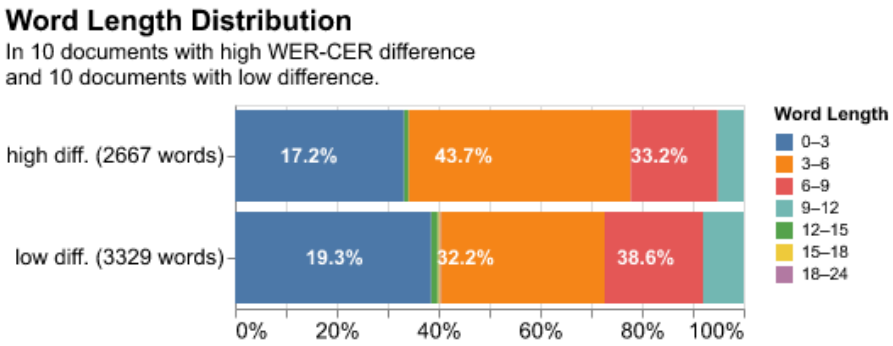


Figure 4.11.: Word length distribution in samples with large vs. small WER-CER gaps. High WER-CER difference cases tend to contain longer words (around 66.8% of words have at least 3 characters), while low-difference cases have shorter words (only about 61.4% are 3 characters or longer).

4.3.4. Training and Inference Time on Single Task

This section provides an approximate estimate of the training and inference time costs for the two pipelines. All measurements were obtained using the hardware setup described in Section 3.1.4.

Overall, Florence-based pipelines require significantly more computational resources for both training and inference compared to the traditional pipeline. Table 4.11 summarizes the task-specific training time for Florence and the corresponding models in the traditional pipeline on the **mixed** split.

Table 4.11.: Estimated training time per epoch on the **mixed** split. For the text object detection task, Florence is significantly slower than YOLO, requiring approximately nine times longer per epoch. In the text recognition task, Florence also takes more than twice as long to train compared to TrOCR.

Task	Images	Model	Time per Epoch (hours)
Region detection on page	7,709	YOLO11m	0.3333
		Florence	3.0000
Line detection on page	7,709	YOLO1m	0.3333
		Florence	3.0000
Text recognition on line	441,408	TrOCR	48.0000
		Florence	132.0000

Table 4.12 shows the total inference time of Florence compared to dedicated models in the traditional pipeline, evaluated on the **mixed** split.

Table 4.12.: Estimated inference time of each task on the **mixed** split. Overall, Florence takes approximately twice as long compared to traditional models.

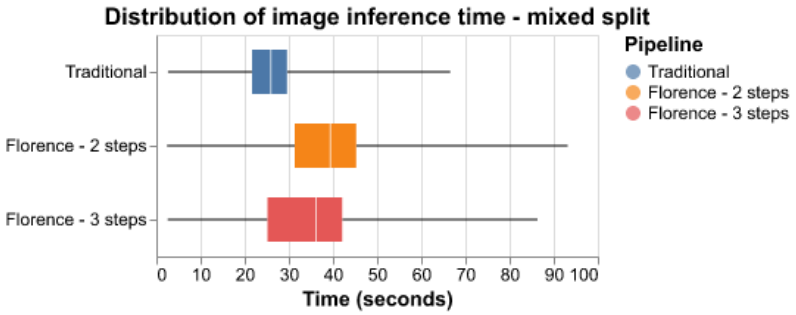
Task	Images	Model	Time per Image (secs)
Region detection on page	951	YOLO11m	0.6940
		Florence	1.2618
Line detection on page	951	YOLO11m	0.6940
		Florence	1.2618
Text recognition on line	54,569	TrOCR	0.2507
		Florence	0.4816

4.3.5. Full Pipeline Inference Time

Figure 4.12 presents the overall mean and median inference time for three pipelines: traditional (region_od, line_seg, ocr), two-step Florence (region_od, ocr) and three-step Florence (region_od, line_od, ocr). At first glance, we can see that the traditional pipeline, even with three steps, is the fastest.

At first glance, the two-step Florence pipeline appears slower than the three-step variant. However, the inference time histogram in Figure 4.13 reveals a subset of images with unusually low inference times, which bring down the overall mean and median.

To investigate, I examined around 100 images with inference times between 8 and 16 seconds in the three-step pipeline. These were failure cases where the pipeline failed to detect any text regions, thus, the subsequent line detection and text recognition steps were skipped. The performance metrics for these cases were: CER 0.9468, WER 0.9647, BoW Hits 0.1157, and BoW Extras 0.3725.



Pipeline	Mean (secs)	Median (secs)
Traditional	25.4996	25.9280
Florence - 2 Steps	38.2361	39.4540
Florence - 3 Steps	33.2889	36.1390

Figure 4.12.: Inference time statistics for full pipelines on the **mixed** split. The traditional pipeline, even with three steps, is still faster than both Florence pipelines. Interestingly, the two-step Florence pipeline appears to be slower than the three-step variant. The reason will be explored.

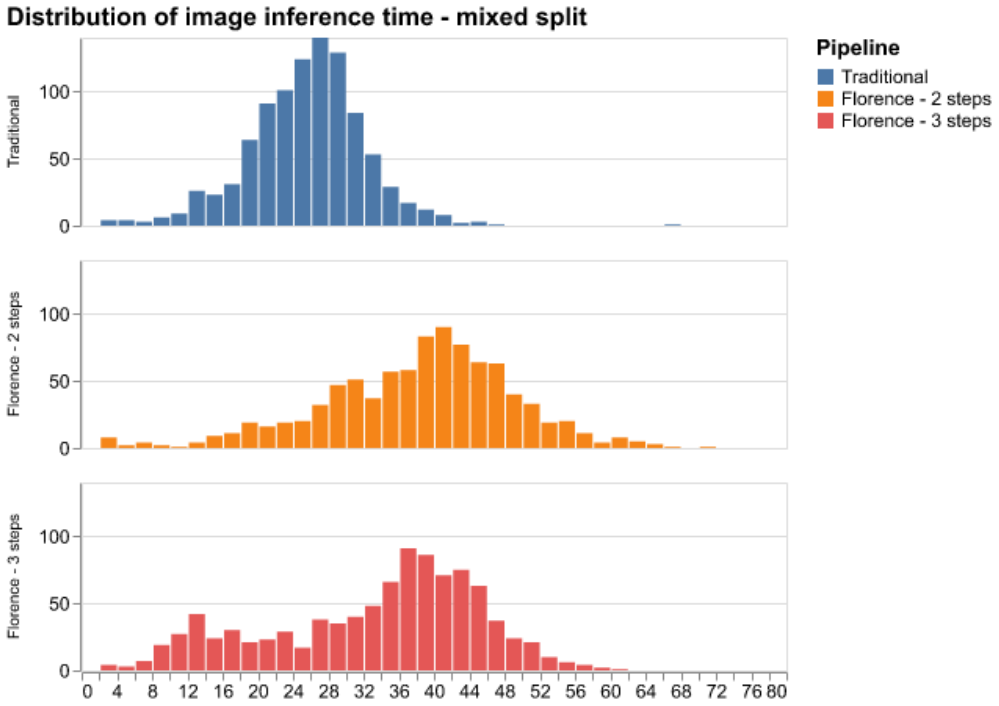


Figure 4.13.: The three-step Florence pipeline seems to be faster than the two-step variant because there is a subset of images with very low inference times, which reduced the overall mean and median. These are failure cases where the three-step pipeline failed at the first region detection step.

In summary, the findings in this section highlight a key drawback of Florence: although its performance in detection and text recognition is on par with state-of-the-art specialized models, it comes with significantly higher computational costs, as well as higher training and inference times. This limits its practicality for large-scale applications.

5. Limitations

5.1. Data Augmentation

As shown in Table 3.1, 72% of the total images used in this work have beige backgrounds, while only 28% are in grayscale. This imbalance may contribute to Florence's failure cases (Section 4.3.2). One possible remedy is to diversify the training set by data augmentation. For example, beige images could be converted to grayscale to add more grayscale samples. The models could also be trained exclusively on grayscale inputs, and at inference time, input images can be turned to grayscale for consistency.

5.2. Evaluation Process

The evaluation metrics used in this study are entirely dependent on the quality of the ground truth annotations. Notably, 428 out of 9,644 annotation files were found to contain no text, which can create misleadingly low performance scores. There are also cases where the transcriptions are incomprehensible. A large portion of the annotations have been generated using Transkribus (a widely adopted commercial HTR solution among historical scholars) and reviewed by experts. It is possible that some files were overlooked during the verification process.

Another limitation of this work is its reliance solely on automated, metric-based evaluation. Since an HTR system is ultimately intended for end users, a live user testing with a demo would offer valuable insights into the system's utility.

5.3. Fine-tuning for Multiple Tasks

In theory, a single Florence model can be fine-tuned to perform multiple tasks. However, this sequential approach requires completing the training for one task before starting the next. For instance, training a model to perform both line detection and OCR would take about a week. As observed in Section 4.1.5, fine-tuning for OCR degraded the object detection performance. As a result, I did not pursue this direction in the current work. A promising next step would be to explore techniques for fine-tuning Florence on different tasks without compromising performance on others.

5.4. Integration with Linguistic Knowledge

As discussed in Section 4.3.2, the reading order determination step is a major source of low performance in the HTR pipeline. Both the traditional and Florence-based pipelines evaluated in this work rely on simple heuristics for determining reading order, without leveraging any linguistic knowledge. Although Florence is described as a "visual language model," its OCR task does not incorporate language understanding when recognizing Swedish handwritten text, as it was trained exclusively on English data. A promising direction for future work would be to integrate linguistic knowledge into the pipeline. This could involve incorporating a large language model (LLM) trained on Swedish as a post-processing component, or retraining Florence-2 using a historical Swedish corpus to enable more linguistically informed predictions.

6. Conclusion

This thesis explored the potential of visual language models for handwritten text recognition (HTR) on historical Swedish manuscripts. By extensively comparing Florence-2—a recently released VLM—with a traditional HTR pipeline composed of specialized computer vision models (YOLO) and a text recognition model (TrOCR), this research aimed to assess whether a VLM-based pipeline is ready for large-scale historical text annotation tasks at the Swedish National Archives (Riksarkivet).

To thoroughly investigate the capabilities of Florence-2 (which supports multiple visual and language tasks out of the box), it was fine-tuned for specific tasks: text region detection, text line detection, text region segmentation, and text recognition. The fine-tuned Florence models were then compared to dedicated models performing the same tasks. Subsequently, end-to-end HTR pipelines composed of two or three models were assembled and evaluated against each other using standard text recognition metrics.

The quantitative and qualitative evaluations addressed the research questions posed in Chapter 1 as follows:

1. **Can large VLMs perform Handwritten Text Recognition (HTR) on Swedish historical manuscripts?**

   Results from the individual-task evaluations show that Florence-2 performs on par with, and in many cases outperforms, specialized models in object detection and text recognition tasks. This demonstrates that VLMs can successfully perform tasks required for HTR.

2. **How does the HTR performance of VLMs compare to specialized HTR pipelines?**

   Quantitative results from the end-to-end pipeline evaluation indicate that Florence pipelines can match or surpass the performance of traditional pipelines. Thus, a VLM-based HTR pipeline is feasible.

3. **How does the computational efficiency of VLMs compare to dedicated HTR pipelines?**

   Qualitative analysis reveals that Florence is significantly more computationally demanding. Both training and inference times are notably higher compared to traditional models, posing a challenge for large-scale use cases.

4. **How does fine-tuning on multiple tasks affect the performance of VLMs?**

   Experiments show that after fine-tuning Florence for object detection, subsequent fine-tuning for OCR degrades its object detection performance. Due to time constraints, this issue was not further explored and is left for future work.

7. Future Works

Aside from the quick extensions mentioned in the Chapter 5, several new research directions can be pursued to address the shortcomings of this work.

First, layer freezing during fine-tuning could be examined as a strategy to prevent degrading previously learned tasks. Second, to better integrate linguistic knowledge into the text recognition and reading order determination steps, future work could explore retraining Florence's language model component with a historical Swedish text corpus. This approach can also be compared to using a large language model (LLM) trained on historical texts as a post-processing module.

Finally, although the overarching theme of this thesis is to explore the capabilities of visual language models in general, only a single model—namely, Florence—was investigated. Future work should explore additional models, either those with a larger number of parameters or more efficient architectures that can be trained within reasonable time and hardware constraints.

Bibliography

Achiam, Josh, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. (2023). "Gpt-4 technical report". *arXiv preprint arXiv:2303.08774*.

AI-Riksarkivet (2023). *htr-flow: A framework for Handwritten Text Recognition workflows*. https://github.com/AI-Riksarkivet/htrflow. Accessed: 2025-04-15.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural Machine Translation by Jointly Learning to Align and Translate". *CoRR* abs/1409.0473. URL: https://api.semanticscholar.org/CorpusID:11212020.

Bao, Hangbo, Li Dong, Songhao Piao, and Furu Wei (2021). "Beit: Bert pre-training of image transformers". *arXiv preprint arXiv:2106.08254*.

Boillet, Mélodie, Christopher Kermorvant, and Thierry Paquet (2022a). "Robust text line detection in historical documents: learning and evaluation methods". *International Journal on Document Analysis and Recognition (IJDAR)* 25.2, pp. 95–114.

Boillet, Mélodie, Christopher Kermorvant, and Thierry Paquet (2022b). "Robust text line detection in historical documents: learning and evaluation methods". *International Journal on Document Analysis and Recognition (IJDAR)* 25.2, pp. 95–114.

Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. (2020). "Language models are few-shot learners". *Advances in neural information processing systems* 33, pp. 1877–1901.

Character set for optical character recognition (OCR-A) (1981). American National Standards Institute. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub32-1-1981.pdf.

Character set for optical character recognition (OCR-A) (1982). American National Standards Institute. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub32-1-1975.pdf.

Colutto, Sebastian, Philip Kahle, Hackl Guenter, and Guenter Muehlberger (2019). "Transkribus. A Platform for Automated Text Recognition and Searching of Historical Documents". In: *2019 15th International Conference on eScience (eScience)*, pp. 463–466. DOI: 10.1109/eScience.2019.00060.

d'Albe, EE Fournier (1914). "On a type-reading optophone". *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 90.619, pp. 373–375.

Dao, Tri (2024). "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: *International Conference on Learning Representations (ICLR)*.

De la Higuera, Colin and Jose Oncina (2014). "The most probable string: an algorithmic study". *Journal of Logic and Computation* 24.2, pp. 311–330.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186.

Ding, Mingyu, Bin Xiao, Noel Codella, Ping Luo, Jingdong Wang, and Lu Yuan (2022). "Davit: Dual attention vision transformers". In: *European conference on computer vision*. Springer, pp. 74–92.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. (2020). "An image is worth 16x16 words: Transformers for image recognition at scale". *arXiv preprint arXiv:2010.11929*.

Feng, Shaolei, R Manmatha, and Andrew McCallum (2006). "Exploring the use of conditional random field models and HMMs for historical handwritten document recognition". In: *Second International Conference on Document Image Analysis for Libraries (DIAL'06)*. IEEE, 8–pp.

Fujitake, Masato (2024). "Dtrocr: Decoder-only transformer for optical character recognition". In: *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pp. 8025–8035.

Garrido-Munoz, Carlos, Antonio Rios-Vila, and Jorge Calvo-Zaragoza (2025). "Handwritten Text Recognition: A Survey". *arXiv preprint arXiv:2502.08417*.

Graves, Alex, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber (2006). "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd international conference on Machine learning*, pp. 369–376.

Hauger, James Scott (1995). *Reading machines for the blind: A study of federally supported technology development and innovation*. Virginia Polytechnic Institute and State University.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Jocher, Glenn and Jing Qiu (2024). *Ultralytics YOLO11*. Version 11.0.0. URL: https://github.com/ultralytics/ultralytics.

Kang, Lei, Pau Riba, Marçal Rusiñol, Alicia Fornés, and Mauricio Villegas (2022). "Pay attention to what you read: Non-recurrent handwritten text-Line recognition". *Pattern Recognition* 129, p. 108766. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2022.108766. URL: https://www.sciencedirect.com/science/article/pii/S0031320322002473.

Kang, Lei, Pau Riba, Mauricio Villegas, Alicia Fornés, and Marçal Rusiñol (2021). "Candidate fusion: Integrating language modelling into a sequence-to-sequence handwritten word recognition architecture". *Pattern Recognition* 112, p. 107790.

Kundu, Amlan, Yang He, and Paramvir Bahl (1989). "Recognition of handwritten word: first and second order hidden Markov model based approach". *Pattern recognition* 22.3, pp. 283–297.

LeCun, Yann, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel (1989). "Handwritten digit recognition with a back-propagation network". *Advances in neural information processing systems* 2.

Lewis, Mike, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer (2019). "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension". *arXiv preprint arXiv:1910.13461*.

Li, Lucian (2024). "Handwriting Recognition in Historical Documents with Multimodal LLM". *arXiv preprint arXiv:2410.24034*.

Li, Minghao, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei (2023). "Trocr: Transformer-based optical character recognition with pre-trained models". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 37. 11, pp. 13094–13102.

Liu, Zhuang, Wayne Lin, Ya Shi, and Jun Zhao (2021). "A robustly optimized BERT pre-training approach with post-training". In: *China national conference on Chinese computational linguistics*. Springer, pp. 471–484.

Mikolov, Tomas, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur (2010). "Recurrent neural network based language model." In: *Interspeech*. Vol. 2. 3. Makuhari, pp. 1045–1048.

Norman, Jeremy M. (2025). *David Shepard Invents the First OCR System 'GISMO'*. Accessed: 2025-04-16. URL: https://www.historyofinformation.com/detail.php?entryid=885.

Oord, Aaron van den, Yazhe Li, and Oriol Vinyals (2018). "Representation learning with contrastive predictive coding". *arXiv preprint arXiv:1807.03748*.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. Pmlr, pp. 1310–1318.

Pletschacher, Stefan and Apostolos Antonacopoulos (2010). "The page (page analysis and ground-truth elements) format framework". In: *2010 20th International Conference on Pattern Recognition*. IEEE, pp. 257–260.

Radford, Alec, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. (2021). "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PmLR, pp. 8748–8763.

Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi (2016). "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

Reul, Christian, Dennis Christ, Alexander Hartelt, Nico Balbach, Maximilian Wehner, Uwe Springmann, Christoph Wick, Christine Grundig, Andreas Büttner, and Frank Puppe (2019). "OCR4all—An open-source tool providing a (semi-) automatic OCR workflow for historical printings". *Applied Sciences* 9.22, p. 4853.

Schantz, Herbert F. (1982). *History of OCR, Optical Character Recognition*. Recognition Technologies Users Association. ISBN: 0943072018.

Shkarupa, Yaroslav, Roberts Mencis, and Matthia Sabatelli (2016). "Offline Handwriting Recognition Using LSTM Recurrent Neural Networks". In: vol. 1. Nov. 2016, p. 88.

Singh, Amanpreet, Ronghang Hu, Vedanuj Goswami, Guillaume Couairon, Wojciech Galuba, Marcus Rohrbach, and Douwe Kiela (2022). "Flava: A foundational language and vision alignment model". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 15638–15650.

Steiner, Andreas, André Susano Pinto, Michael Tschannen, Daniel Keysers, Xiao Wang, Yonatan Bitton, Alexey Gritsenko, Matthias Minderer, Anthony Sherbondy, Shangbang Long, et al. (2024). "Paligemma 2: A family of versatile vlms for transfer". *arXiv preprint arXiv:2412.03555*.

Sueiras, Jorge, Victoria Ruiz, Angel Sanchez, and Jose F Velez (2018). "Offline continuous handwriting recognition using sequence to sequence neural networks". *Neurocomputing* 289, pp. 119–128.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". *Advances in neural information processing systems* 27.

Tarride, Solène, Aurélie Lemaitre, Bertrand Coüasnon, and Sophie Tardivel (2019). "Signature detection as a way to recognise historical parish register structure". In: *Proceedings of the 5th International Workshop on Historical Document Imaging and Processing*, pp. 54–59.

Team, Gemini, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. (2023). "Gemini: a family of highly capable multimodal models". *arXiv preprint arXiv:2312.11805*.

Touvron, Hugo, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou (2021). "Training data-efficient image transformers & distil-

lation through attention". In: *International conference on machine learning*. PMLR, pp. 10347–10357.

Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. (2023). "Llama 2: Open foundation and fine-tuned chat models". *arXiv preprint arXiv:2307.09288*.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). "Attention is all you need". *Advances in neural information processing systems* 30.

Vincent, Luc (2006). *Announcing Tesseract OCR*. URL: https://googlecode.blogspot.com/2006/08/announcing-tesseract-ocr.html (visited on 2025-04-01).

Vinciarelli, Alessandro (2002). "A survey on off-line cursive word recognition". *Pattern recognition* 35.7, pp. 1433–1446.

Wang, Wenhui, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou (2020). "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers". *Advances in neural information processing systems* 33, pp. 5776–5788.

Willis, Nathan (2006). *Google's Tesseract OCR engine is a quantum leap forward*. URL: https://www.linux.com/news/googles-tesseract-ocr-engine-quantum-leap-forward/ (visited on 2025-04-01).

Xiao, Bin, Haiping Wu, Weijian Xu, Xiyang Dai, Houdong Hu, Yumao Lu, Michael Zeng, Ce Liu, and Lu Yuan (2023). "Florence-2: Advancing a unified representation for a variety of vision tasks (2023)". *URL https://arxiv.org/abs/2311.06242*.

Yuan, Haobo, Xiangtai Li, Tao Zhang, Zilong Huang, Shilin Xu, Shunping Ji, Yunhai Tong, Lu Qi, Jiashi Feng, and Ming-Hsuan Yang (2025). "Sa2VA: Marrying SAM2 with LLaVA for Dense Grounded Understanding of Images and Videos". *arXiv preprint arXiv:2501.04001*.

A. Training Procedures

A.1. Task-specific Datasets

Each task operates on a different granularity of image data. For region-level tasks such as line detection or segmentation within a region, page images are cropped into smaller regions using region bounding boxes. Similarly, for line-level text recognition, pages are cropped into individual lines using either line bounding boxes or polygon annotations. The resulting region- and line-level images inherit the train, validation, or test split assignment of their corresponding source page.

Table A.1 summarizes the total number of images available for each task. The larger number of images in lower-level tasks results in increased training time, a point that will be further discussed in Section 4.3.4.

Task Description	**Total Images**
Page-level text region and line detection	9,644
Region-level line detection and segmentation	66,008
Line-level text recognition	543,930

Table A.1.: Total image counts for each task across all data splits. Region- and line-level images inherit the train, validation, test split assignment of their page-level source.

A.2. Annotation processing

In the PAGE-XML files of the raw dataset, both text regions and lines are annotated using boundary polygons represented as a list of `(x, y)` points in pixel coordinates. For consistency, I derive rectangular bounding boxes in the "top-left and bottom-right" format by taking the minimum and maximum values of the `x` and `y` dimensions from the polygons. These representations are then transformed into the formats required by each model.

For the YOLO object detection model, the output is a list of bounding boxes in the `(x, y, width, height)` format, with `(x, y)` being the box's center point. Both the coordinates and size measurements are normalized to the [0, 1] range by the image's actual width and height. YOLO segmentation model's output is a list of segmentation masks. Each mask is a series of `(x, y)` points, with `x`, `y` being pixel coordinates normalized by the image's width and height.

Florence-2's bounding box uses the top-left and bottom-right representation, and its segmentation masks consist of a series of `(x, y)` points. All raw coordinates are quantized, as mentioned in Section 3.3.2, which means they need to be converted back to pixel-based coordinates when doing performance evaluation for visual tasks.

The label for the text recognition task requires no special transformation, since they are simply lines of text.

A.3. Traditional Pipelines Hyperparameters

The `yolo11m` models are trained for 10 epochs with a batch size of 6 and an image size of 1280, using the default hyperparameters. The `yolo11m-seg` models follow the same training duration and image size but use a smaller batch size of 2, also with default hyperparameters.

The TrOCR models are trained for 80,000 steps (approximately one epoch) with a batch size of 6. I use the Adam optimizer with an initial learning rate of 2×10^{-5}, a weight decay factor of 0.0001, and an inverse square root learning rate scheduler with zero warm-up steps. These settings follow the default fine-tuning configuration listed on the model's official release page.

A.4. Florence-based Pipelines Hyperparameters

For tasks involving object detection (such as text region and line detection), Florence is trained for 40,000 steps (approximately 10 epochs) with a batch size of 2.

For tasks that operate on cropped line images (such as single-line segmentation and OCR), the models are trained for 220,000 steps (roughly equivalent to one epoch) with a batch size of 2. For optimization, I adopt the hyperparameters recommended on the model's official release page: Adam optimizer with a learning rate of 1×10^{-6}, and a linear learning rate scheduler with zero warm-up steps.

B. Pipelines

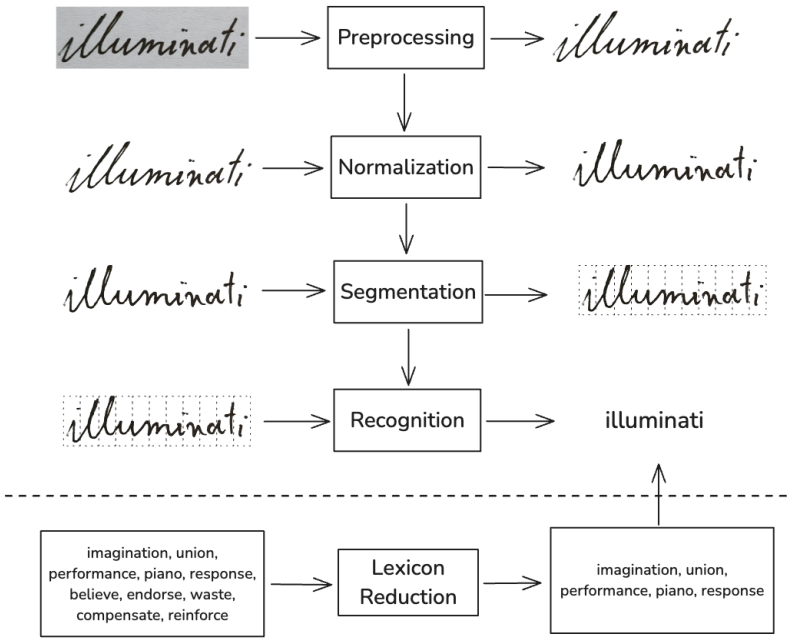


Figure B.1.: Illustration of a traditional HTR pipeline, adapted from Vinciarelli (2002).

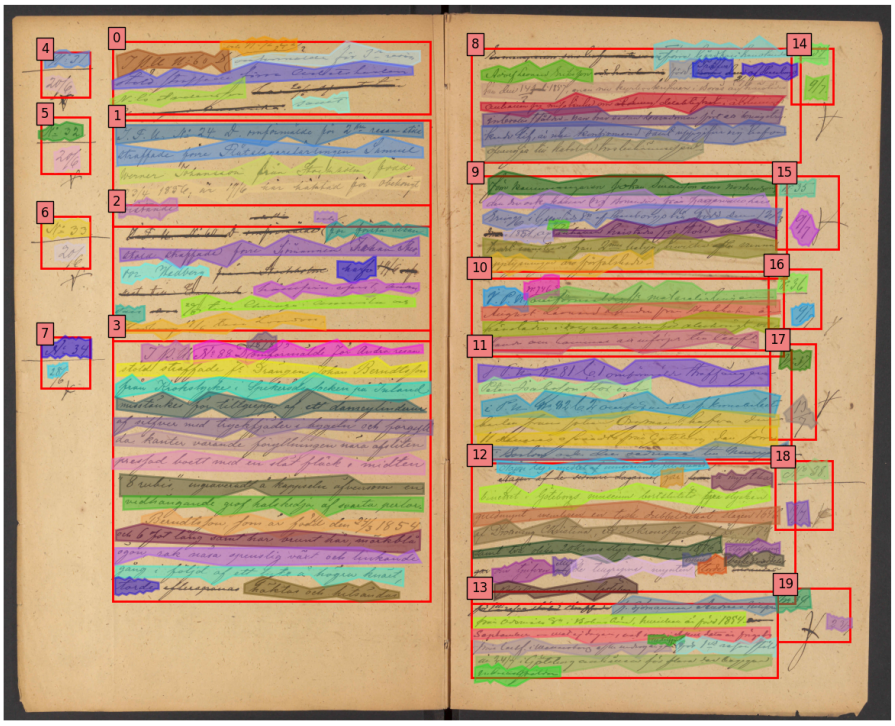


Figure B.2.: Heuristic to determine reading order. First, the image is guessed to be a two-page spread if there is a vertical division line in the middle. This division line is identified as the darkest column of pixels in the middle area of the image. Next, detected objects (regions, text lines) are determined as either belonging to the left page or the right page based on their position relative to the division line. Finally, the objects are sorted in this order: top margin, main text, bottom margin, and side margin. The numbers on the bounding boxes represent the reading order.

C. Input - Output

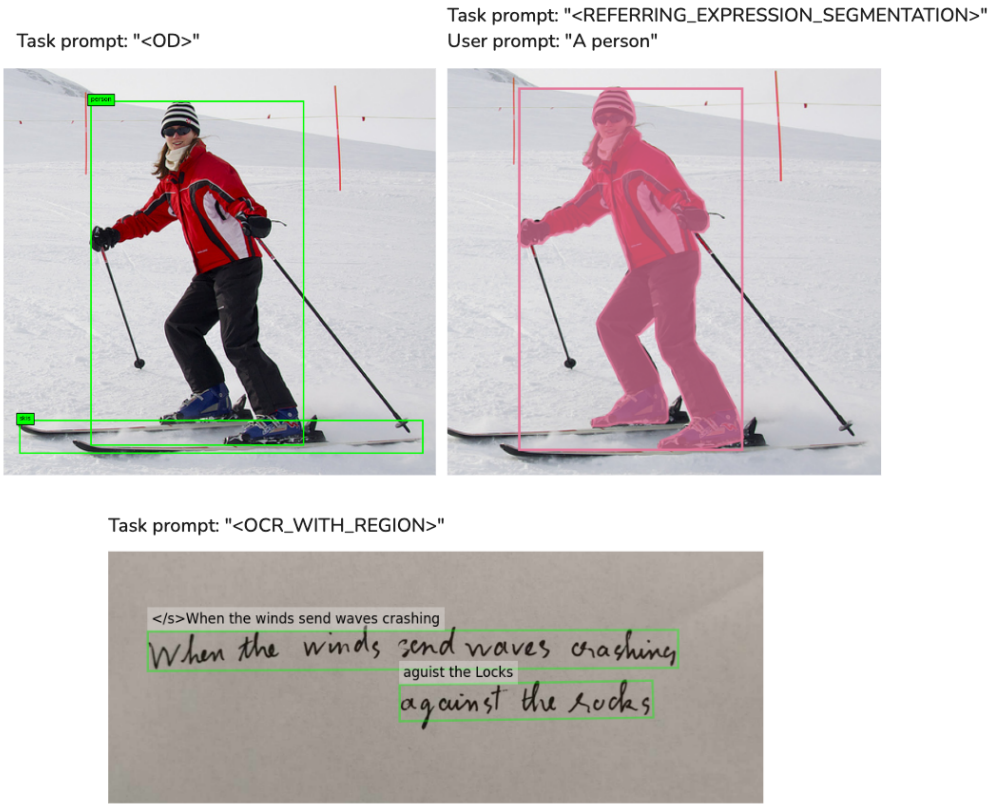


Figure C.1.: Example of Florence-2 multi-task capabilities. To perform different tasks, the user simply inputs the corresponding task prompts. The mask shown here requires 380 location tokens, representing a polygon of 190 points.

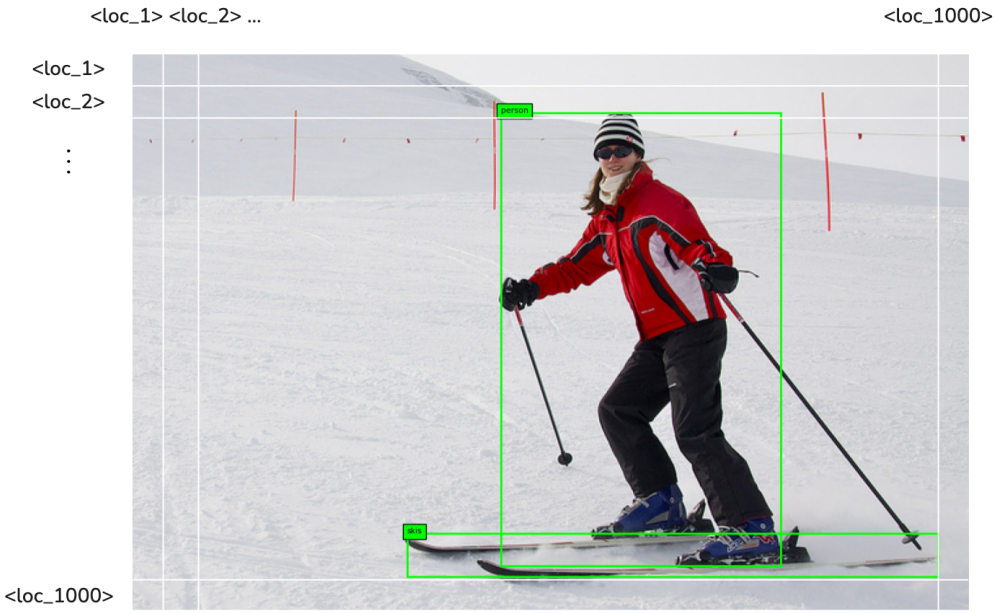


Figure C.2.: Example of Florence's raw and parsed output for the object detection task, discussed in Section 3.3.2.

D. Layout Examples

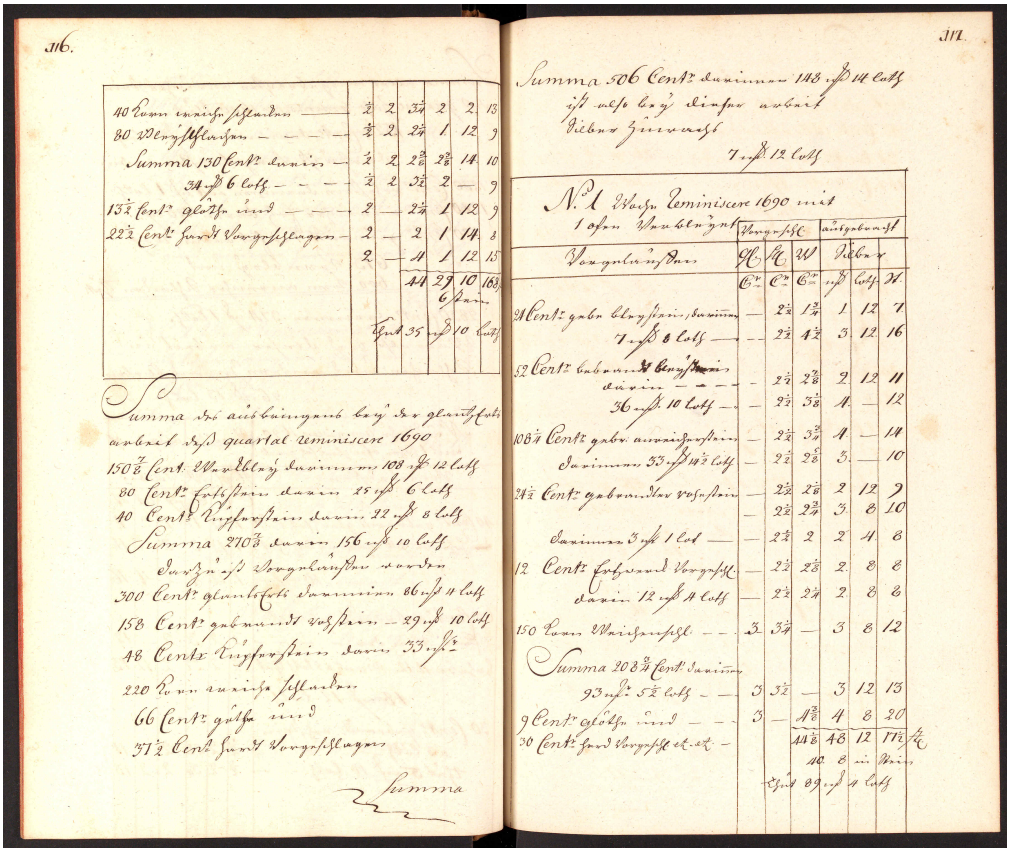
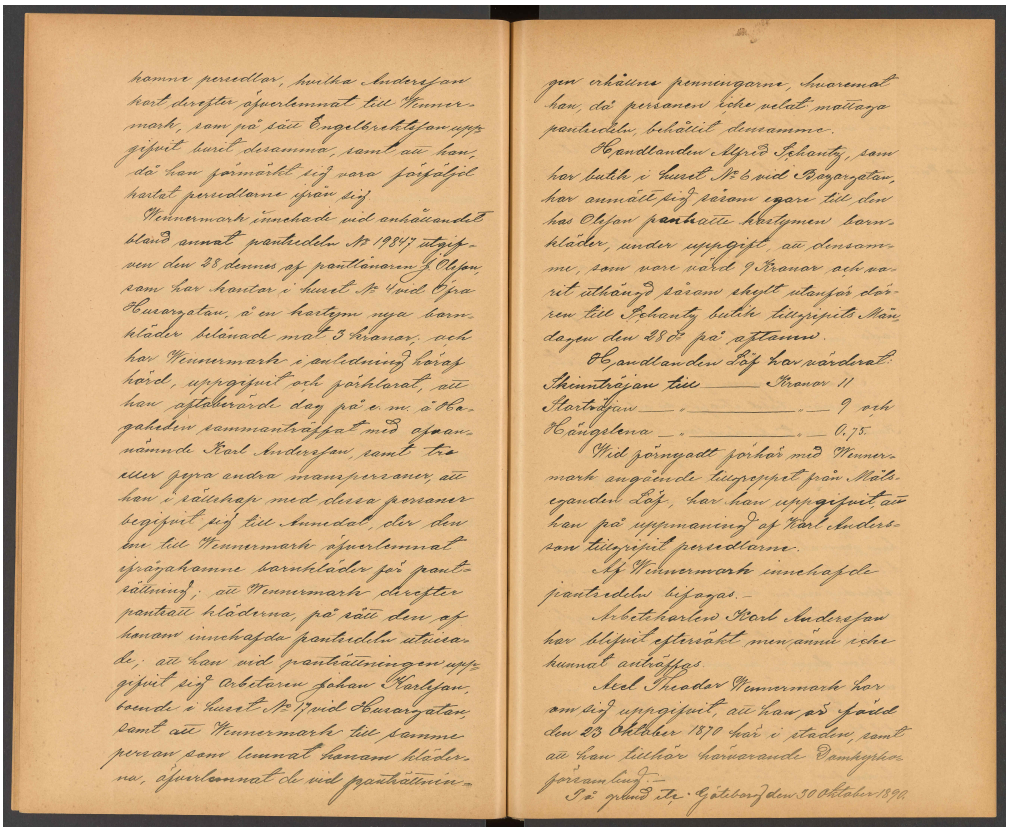


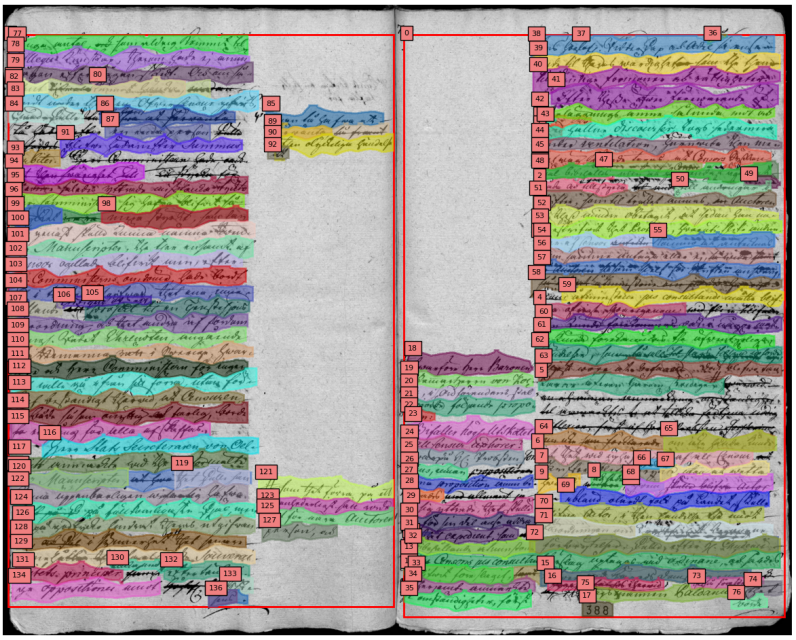
Figure D.1.: Example image of a page with mixed content type and arbitrary reading order.

E. Success Cases

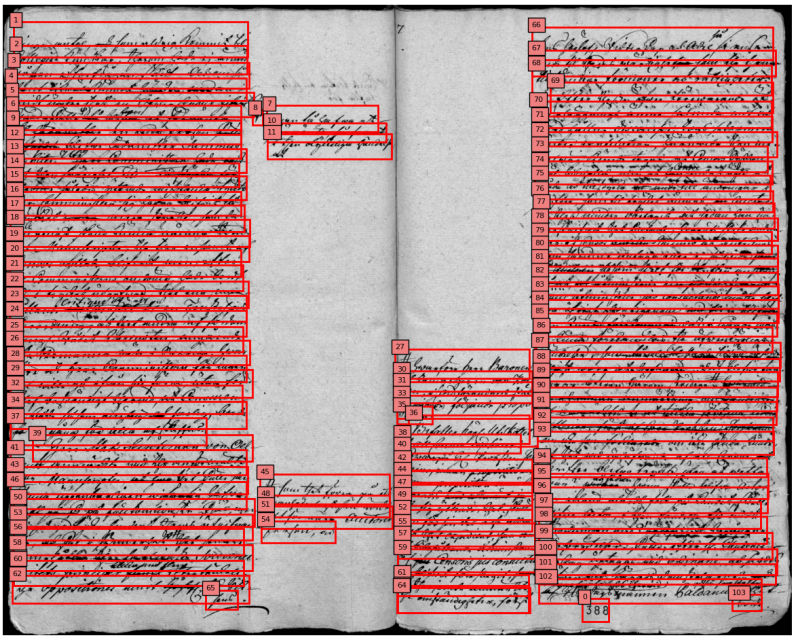


Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.0053	0.0309	0.9691	0.0309
Traditional	0.0091	0.0463	0.9537	0.0433

Figure E.1.: An example where both pipelines perform well.



(a) Traditional pipeline's prediction

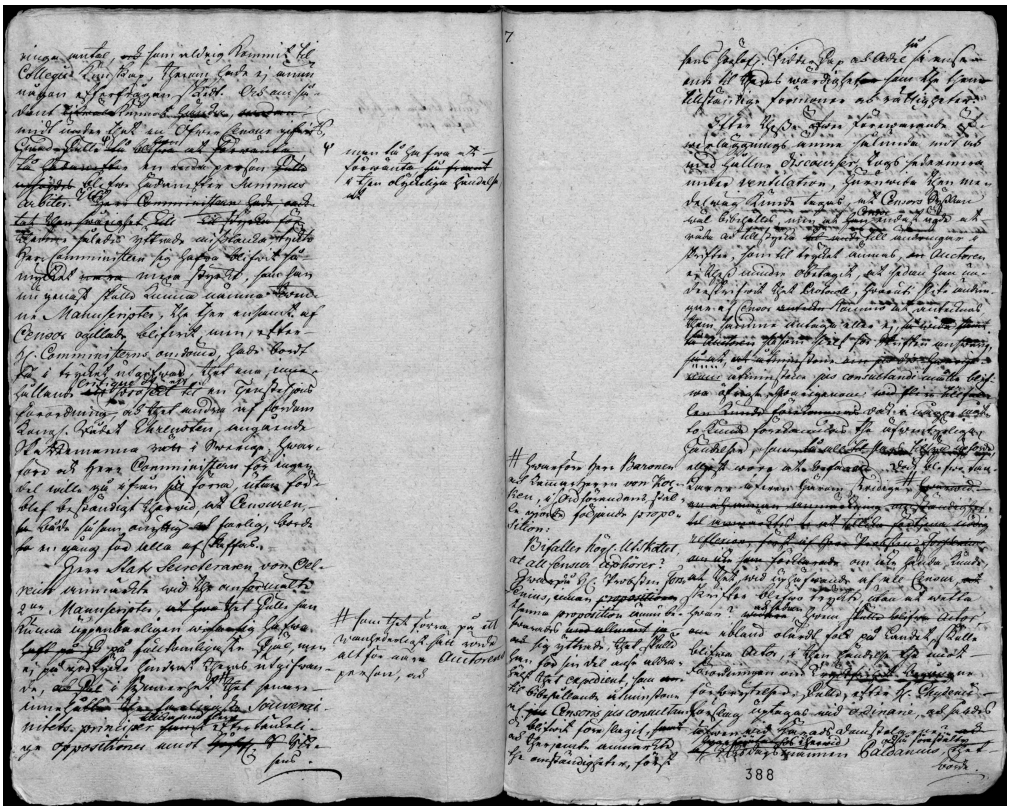


(b) Florence pipeline's prediction

Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.3584	0.5307	0.6927	0.2831
Traditional	0.7137	0.9610	0.5577	0.3472

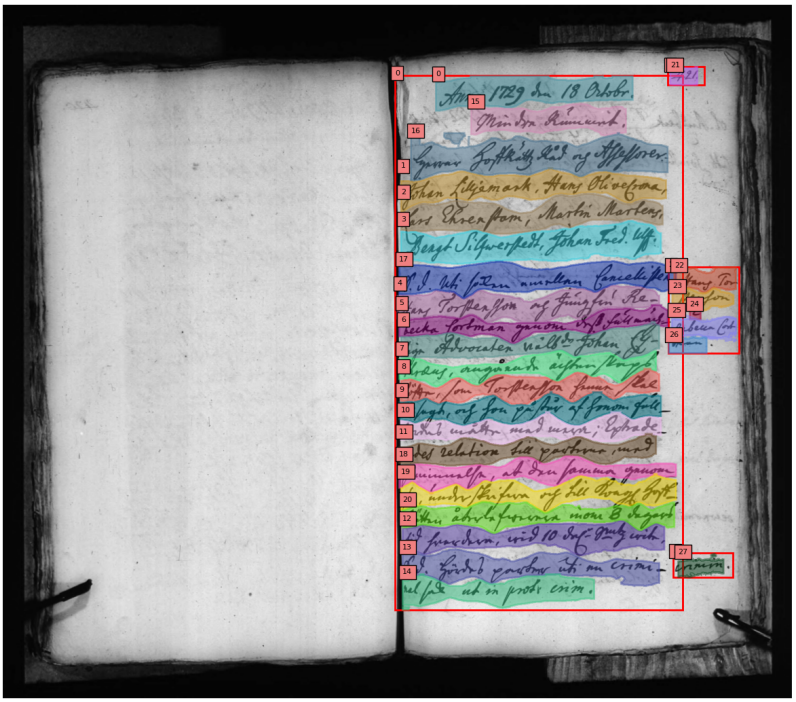
Figure E.2.: Another example where the traditional pipeline has cascading failures while Florence is more successful. In Figure E.2a, we can see there are short segments that start in the middle of a line, which means the lines are over-segmented. Arbitrary segmentation also confuses the sorting heuristic, further degrading overall performance. In contrast, the Florence pipeline identifies and crop lines using bounding boxes, providing more complete inputs to the OCR step compared to segmentation masks.

F. Failure Cases

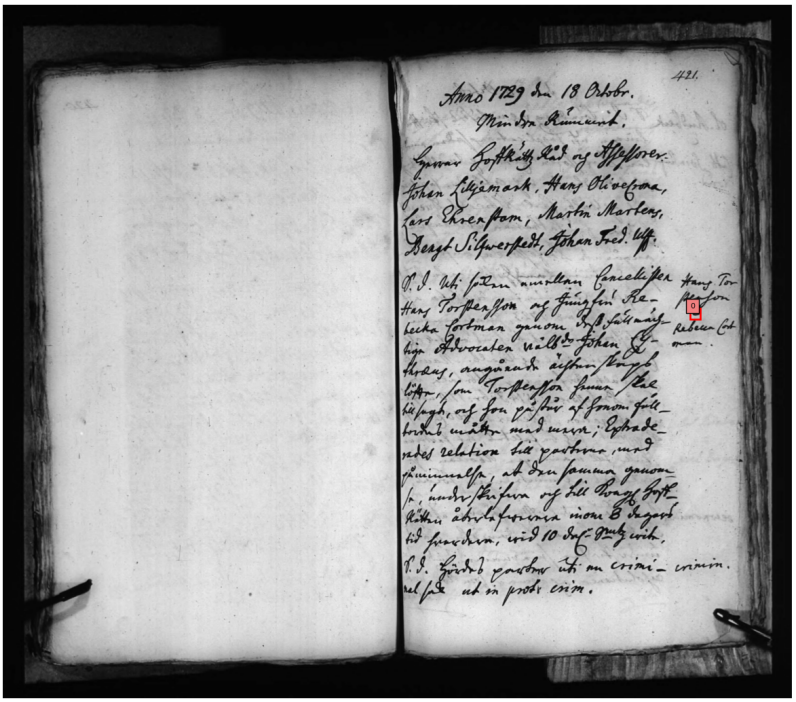


Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.3584	0.5307	0.6927	0.2831
Traditional	0.7143	0.9628	0.5595	0.3598

Figure F.1.: In this case, even when Florence is more successful than the traditional pipeline (Figure E.2), it is still a difficult case. Its layout is very different from the assumption of the sorting heuristic (Figure B.2): margin texts are placed in the center of the page instead of on the two sides. Wrong sorting means higher CER and WER since these metrics are based on edit distance, which emphasizes the order of letters or words in a string.



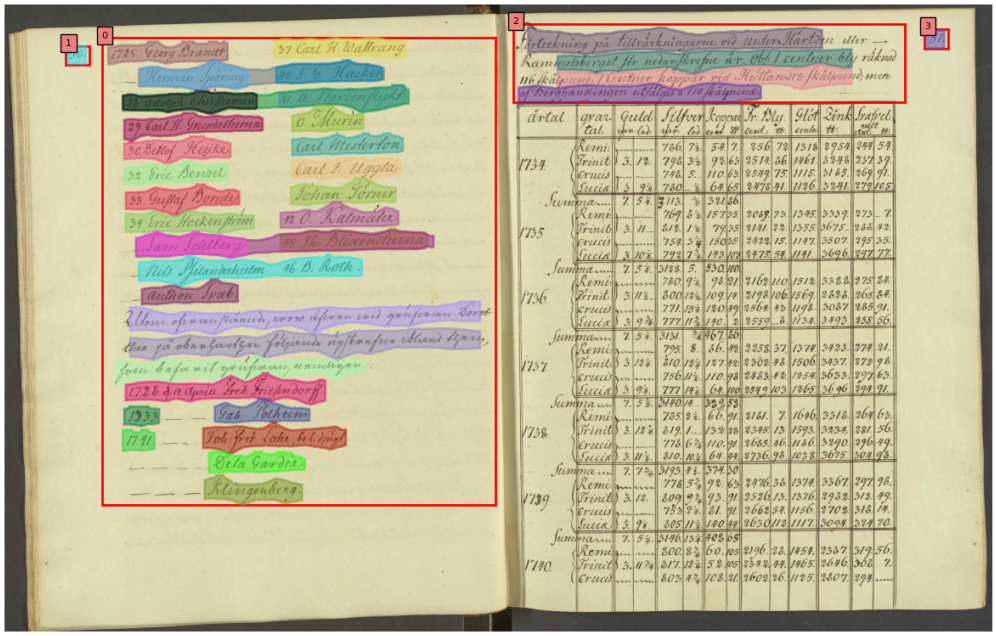
(a) Traditional pipeline's prediction



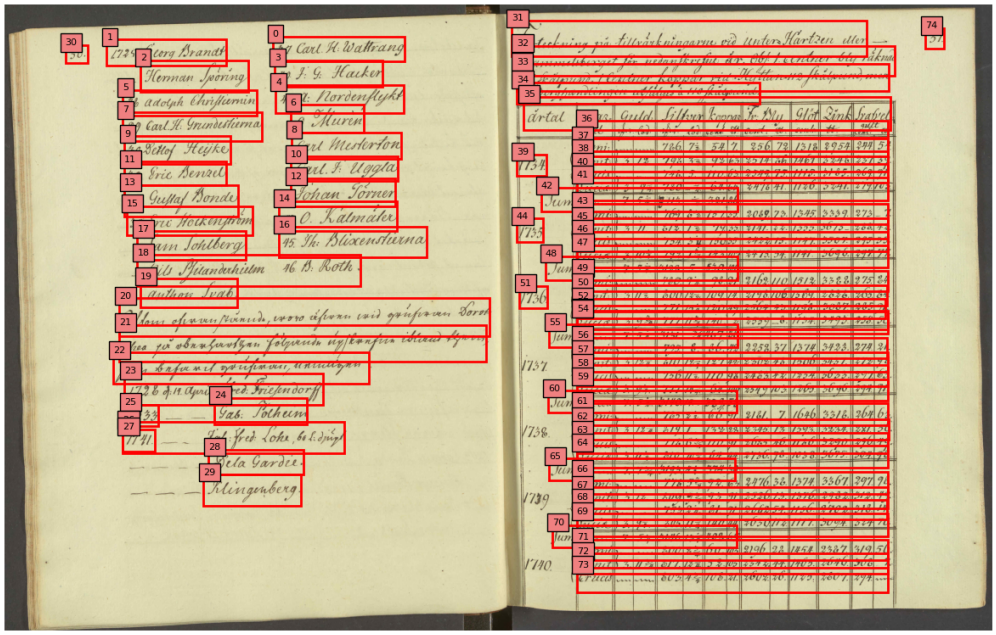
(b) Florence pipeline's prediction

Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.9986	0.9914	0.0086	0.0000
Traditional	0.4762	0.5973	0.7845	0.2155

Figure F.2.: An example where Florence performs poorly compared to the traditional pipeline. In this case, the traditional method successfully detects regions, segments lines, and recognizes text. Florence, however, fails at line detection, resulting in no input for the OCR step. Notably, a lot of these cases are in the **svea_hovratt_seg** dataset (Table 3.1).



(a) Traditional pipeline's prediction



(b) Florence pipeline's prediction

(c) Florence is able to guess more correct words than the traditional pipeline.

Pipeline	CER	WER	BoW Hits	BoW Extras
Florence	0.7862	0.8819	0.6894	0.8496
Traditional	0.4784	0.5938	0.5606	0.3934

Figure F.3.: Traditional and Florence pipelines both fail here, but for different reasons. While the traditional pipeline with region detection can detect the main text and ignore the table, it performs poorly at the OCR stage. In contrast, the Florence pipeline tends to overshoot since it detects line-level text, and also captures the text inside the table. As a result, it has high CER, WER, and BoW Extras, but also high BoW Hits due to better OCR performance.